Evaluation of textual variability languages in the context of complex systems

Alvarez Morales, Javier

Award date:
2020

Master thesis

Evaluation of textual variability languages in the context of complex systems

Author:

Javier Alvarez Morales

Supervisors:

Dr.Ir. L.G.W.A Cleophas
l.g.w.a.cleophas@tue.nl
Eindhoven University of Technology

Dr. G. Kahraman
g.kahraman@tue.nl
Eindhoven University of Technology

Dr.Ir. R.R.H Schiffelers
ramon.schiffelers@asml.com
ASML / Eindhoven University of Technology

July 30, 2020

Contents

Introduction

Modern software-intensive systems exhibit exponential growth in size and heterogeneity [1]. Increasingly powerful and versatile hardware, along with fast-moving and highly competitive markets demand innovative software engineering methodologies. Software product line engineering (SPLE) offers a systematic approach to reuse and manage software artefacts—e.g. requirements, architecture, code, test cases. A *software product line* (also known as *software product family*) can be defined as "a set of software-reliant systems that share a common managed set of features satisfying a particular market or mission area, and are built from a common set of core assets in a prescribed way" [2]. This approach of software development has proved to shorten development time and reduce costs while improving the overall quality by means of reuse [3].

Problem context

In the context of modern complex systems—such as the lithography machines designed and built by ASML—variability management methodologies face challenges that remain unaddressed by "traditional" SPLE techniques. The behaviour of complex systems is difficult to model due to the dependencies, relationships, or other types of interactions between their parts and with their environment [4]. These systems are usually also "variable-intensive" because they need to accommodate diverse application and deployment scenarios [1]. Furthermore, today's stakeholders and customers expect flexibility in even more dimensions (e.g. features, context awareness, processing power, etc.). The increasing system complexity and variability requirements in ASML's products demand innovative approaches to handling variability.

Lithography machines are heterogeneous systems, integrating mechanical, electrical/electronics and software systems. The system's precision level is achieved using dozens of interconnected computers running hundreds of parallel processes, meeting strict timing requirements and implementing advanced control techniques. Complex systems which integrate computation, physical processes and networks are called Cyber-physical systems (CPSs) [5]. A key element in the design of CPSs is the study of joint dynamics; computers, software, networks and physical processes are often modelled together using the so-called hybrid models [5]. The variability in application and deployment scenarios add to the already inherently complex software that controls CPSs.

Problem Description

This works focuses on the problem of variability representation, which is the first step in the variability management process. There is a need for formal description of variability among different systems to apply computer-assisted analysis techniques that will ultimately enable the systematic reuse of engineering artefacts. Reuse is a key factor in shortening engineering development cycles, reducing costs and improving product quality.

This work proposes the following path: systematic literature review, the application of a theoretical expressiveness evaluation framework, and the selection and practical evaluation of a variability modelling language at ASML. Firstly, we will analyze the comparison of recent textual variability languages based on the systematic literature review by Eichelberger et al. in [6] and expanded by ter Beek et. al in [7]; we further expanded these works by adding tool support as an evaluation criterion. Secondly, Asadi's et al. theoretical framework for variability (ATFV) will be used to assess the expressiveness of fourteen VMLs. Based on the capabilities in the literature survey, the expressiveness provided by using ATFV and the robustness of the toolset, one VML will be selected and evaluated using ASML's variability models. Thirdly, we will verify if the selected variability language has the necessary constructs to represent large variability models such as ASML's, and if tools are robust enough to deal with them.

Specifically this work aims to address the following (research) questions:

- RQ1: Can we effectively evaluate the existing variability modelling languages using the aforementioned approaches?
- RQ2: Is the selected variability modelling language expressive enough to represent ASML variability models?
- RQ3: Are the tools of the selected variability modelling language robust enough to deal with the variability models at ASML?

Thesis organization

Chapter 1 presents an overview of the variability management process in the context of SPLE and the importance of variability management tools for industry adoption. This context is important for understanding the role of variability modelling within the variability management process.

Next, the most relevant capabilities of fourteen textual variability languages are compared based on the work of Eichelberger et al. [6] and ter Beek et al. [7]. Tool support was added as an evaluation criterion due to its relevance for industry adoption in general, and for the practical aspect of our evaluation in particular. The recently proposed High-Level Variability Language (HLVL) was added for completeness and also because of it is presented as an intermediate representation for other variability languages. Finally, Eichelberger's et

al. and ter Beek's et al. assessment was revised; the results of all of this are presented in Chapter 2.

Furthermore, Asadi's et al. theoretical framework for variability (ATFV) is expanded and then used to provide an alternative and more detailed assessment of feature modelling. This mapping is then extrapolated to the fourteen VMLs considered in Chapter 2. The value of this proposal as an evaluation framework is discussed in Chapter 3

Chapter 4 then presents a model transformation from the ASML variability models to the chosen variability language. Missing constructs in the target language are identified through this process (RQ2). Furthermore, robustness of the chosen variability language tools is evaluated by performing analysis in the translated models to address RQ3.

Finally, the conclusions of our work and a definitive answer for RQ1 are presented in Chapter 5, where we address the implications for ASML and discuss future work opportunities.

Acknowledgements

I would like to thank to my thesis supervisors Dr.Ir. Loek G.W.A Cleophas, Dr. Gökhan Kahraman, and Dr.Ir. Ramon R.H Schiffelers for their feedback and support during this work. In particular to Dr.Ir. Cleophas, for patiently reading every draft of this work and providing not only helpful insights but encouraging words.

To ASML for giving me the opportunity of this challenge. Special thanks to Pieter Kunst from the configuration management team at ASML, who answered many questions related to the company's variability models; and to Michał Antkiewicz from the University of Waterloo, who always replied to my emails about the Clafer language and its toolset.

This work is specially dedicated to my wife Aileen, who is always there when I feel overwhelmed and doubtful. Thank you for proof-reading this document and asking the right questions to improve its contents.

Chapter 1

Variability management in SPLE

Identifying and managing commonalities and variability in software product lines is usually referred to as *variability management* [8]. Variability management comprises *domain* and *application* engineering processes [3]. The domain engineering process is responsible for defining commonality and variability of the product line, and for establishing reusable artefacts. The application engineering process is responsible for deriving product line applications from those reusable artefacts. Application engineers bind existing software components according to application-specific needs, maximizing the benefits of reusable artefacts [9].

1.1 An overview of the variability management process

A traditional software product line is composed of three main parts: *problem space*, *solution space* and *a mapping* between these two spaces [9]. The problem space is captured in the variability model. The actual artefacts used to build the product variants form the solution space. To enable software product derivation, the links between configurable elements in the model and the artefacts implementing those elements should be explicitly documented. This mapping information is frequently referred to as *configuration knowledge* [10]—dependencies and constraints among elements in the variability model are considered configuration information as well.

The variability modelling process defines and documents variabilities and commonalities among a set of products; although some approaches, such as decision modelling, focus only on documenting variability [11]. A variability model *represents* the variability between a set of products so that it provides the information at a system's level and facilitates comprehension of variability [12].

The variability modelling process needs to answer two basic questions: *1. What varies?* and *2. How does it vary?* [9]. The answer to the first question leads to a set of variable items or variable properties of an item called *variation points*. By answering the second question, the

possible instances of a variable item or a variable property, the *variants*, are identified.

Once the system's variation points and variants are defined, the domain engineering process identifies their dependencies and constraints. These constraints may be between two variants, between a variation point and a variant, or between variation points. Variation points, variants and constraints comprise a *variability model*, which is formally defined using a *variability modelling language* (VML).

The process of resolving all the variability within a variability model is known as *product derivation*. When a derived product satisfies all the variability model constraints, we say that it is a *valid product*—otherwise it is *invalid* [3].

Several variability modelling techniques have been developed over the last three decades [12]. Feature modelling and decision modelling have become the most popular [13]. The majority of existing variability languages support at least one of these approaches, with feature modelling being the most widely supported [14].

Feature modelling has its origins in Feature-Oriented Domain Analysis (FODA) [15]. Feature models capture features, which are defined as "end-user visible characteristics of systems in the domain" [13]. Features are organized hierarchically in a feature tree, and feature groups represent choices between multiple sub-features [12]. Constraints and dependencies between features can typically be expressed in distinct constraint formalisms. Feature models are used to model both the commonality and the variability of a set of products. Product derivation is performed by selecting features from the feature model.

Decision modelling focuses on decisions that distinguish the products of a product line; its main goal is to support product derivation. Decisions can be defined as differences among systems [13], i.e., what needs to be decided upon when configuring a system. Decision models do not document commonality—a characteristic that differentiates them from feature models. The mapping of decisions to reusable assets is, therefore, a central concept in decision modelling. Dependencies and constraints between decisions can be defined using constraint languages as in feature models.

Commonality and variability between software systems is modelled through variability models, forming the so-called *problem space*. The set of common artefacts implementing the functionality described in a variability model is the *solution space*. In order to automatically build products from the software artefacts using configurations derived from a variability model it is necessary to define a *mapping* between problem and solution space. The correct traceability between elements represented in a variability model and the implementation artefacts has been identified in the literature as one of the most important challenges in variability management [14].

After mapping the problem space to the solution space, a mechanism to build the product needs to be defined. This mechanism has three inputs: a product configuration derived from the variability model (problem space), a set of common artefacts (solution space) and the mapping information of configurable items to artefacts. Techniques such as *cloning, conditional compilation, conditional execution, polymorphism, module replacement* and *runtime*

reconfiguration are variability realization examples, each one with a different level of complexity. Factors like code granularity, change frequency and binding time should be considered when deciding which mechanism to use; no single approach is appropriate for every situation. These realization techniques are independent of the variability representation and can be combined to effectuate the variability specified in the model. For instance, two different SPLs could represent variability using feature models, yet one uses conditional compilation and module replacement while the other relies completely on runtime reconfiguration to effectuate the variability.

1.1.1 Variability management tools and industry adoption

A plethora of variability modelling languages and tools have been developed over the last three decades, however, these tools rarely go beyond basic proof-of-concept implementation and most of them have had minimal industry success [14]. Variability management tools used in the industry are frequently developed internally and thus meet companies' variability requirements, but rarely work in different contexts and domains [16, 17]. In some cases, custom tools do not completely follow a particular variability modelling approach (e.g. feature modelling). Moreover, variability models are often not used; instead, the configuration knowledge is described directly in the code, through simple XML files or DSLs [14]. These solutions might initially solve the variability management problem, but often fall short as the configuration knowledge increases.

The vast majority of open-source variability management tools have been developed as part of an academic project, given that internally developed custom tools are rarely shared by companies. Thus, their evaluation is usually limited to academic tools. Scientific papers seldom provide an in-depth analysis of the presented tools and their limitations; instead, toy examples are often used to highlight the advantages (and sometimes a few shortcomings) of the presented proposal. Quality attributes relevant to the practical use of tools such as *usability*, *scalability*, *integration*, and *performance* are usually not included in these studies; let alone feedback from end-users [14]. This situation is understandable; academic projects often have limited budgets and tight deadlines. With such limited resources, developing fully-fledged toolsets for their language proposals or providing a thorough evaluation of them is not feasible.

An example of a successful variability management tool, designed and developed outside academia, is KConfig—the Linux Kernel configuration management tool. The kernel developers designed the KConfig language as an explicit variability specification for the Linux Kernel [18]. The KConfig model closely resembles and can be represented as a feature model [19]. The Linux variability management system has proven to be effective in handling the Kernel's complexity and its exponential growth in size; the code base has more than 15,000 configuration options and an immense number of possible product configurations [20]. Although KConfig provides a working solution for the Linux kernel domain, it cannot be easily applied in other contexts. This variability language only incorporates simple nested configuration elements called *configs* and a basic constraint language to restrict the options on those configs, which is not expressive enough for different domains.

Three main challenges in variability management tools have been identified in the literature: *scalability of variability models, consistency and correctness verification* and *mapping problem and solutions space* [14]. Both the language and the tools must incorporate features that allow scaling the variability models in a principled manner. Otherwise, the variability management software easily becomes an ad-hoc collection of tools using an inscrutable patchwork of techniques. Applying automated reasoning techniques to ensure consistency and correctness of the variability models requires that the language provides well-defined semantics. When consistency and correctness of the models are not effectively verified, the tools quickly fall into disuse. The problem of mapping between problem and solution spaces is not even addressed by many tools [14]. For those that incorporate some mapping information, the used techniques vary.

1.1.2 Conclusions

This chapter provided a brief description of the variability management process in SPLE. Additionally, a few challenges on the adoption of variability management tools by the industry were described. The purpose of this chapter was to locate variability modelling within the wider context of variability management.

In the next chapter we focus on analysing the capabilities of fourteen variability modelling languages, which will provided us with an overview of the state recent variability modelling languages.

Chapter 2

Variability modelling language capabilities

Dozens of visual and textual variability modelling specifications have been proposed during the last three decades, each one offering distinctive capabilities as variability modelling needs evolve. This chapter will provide an overview and a comparison of those main characteristics using recent language proposals.

Textual representation of variability improves interoperability with other tools by facilitating model transformations—an important requirement when working with large and complex models. It also enables the use of existing tools for editing and manipulating textual statements [21]. Even though visual variability representations—like FODA—provide an important cognitive advantage, they become unpractical as the models grow. With an appropriate textual representation, visualization tools can be built on top, improving usability and enhancing the user's experience. Hence, only text-based variability languages are considered in the presented overview.

2.1 Contribution

Several published works evaluate the capabilities of different variability languages [6, 7, 22–24] based on different evaluation criteria. This chapter is the result of a literature study and provides a comparison of the capabilities offered by fourteen variability modelling languages proposed over the last two decades. The purpose of the information presented in this chapter is to provide a useful overview of the main characteristics offered by the included VMLs; it does not pretend by any means to give a detailed description of every language.

This chapter is based on the systematic literature review conducted by Eichelberger et al. in [6] and then updated by ter Beek et al. in [7]. These publications provide a succinct yet complete overview of the most relevant capabilities offered by the VMLs proposed in

recent years. Our contributions here, compared to [6] and [7], consist of: 1) incorporating the recently proposed High-Level Variability Language (HLVL); 2) adding tool support as a criterion for comparison; and 3) revising the support level granted to some VMLs in certain categories after a careful review of the language's references.

We argue that toolset support is a critical aspect for industry adoption because companies need to be certain that a variability language will work in their use case before integrating it as part of their workflow. Because companies rarely consider a new variability language if they need to develop tools from scratch to simply evaluate it, tool support provided by each variability language is presented in detail in this chapter.

2.2 Chapter organization

The remainder of this chapter is organized as follows:

- Section 2.3: presentation of the fourteen VMLs included in this evaluation.
- Section 2.4: description of capabilities selected in this evaluation.
- Section 2.5: comparison of the VMLs based on the considered capabilities.
- Section 2.6: introduction of other capabilities specifically relevant to cyber-physical systems.
- Section 2.7: chapter's conclusions and discussion of promising language proposal in the context of ASML.

2.3 VMLs considered

The list of fourteen textual variability languages is shown below. Items 1–11 were originally presented by Eichelberger et al. in [6], ter Beek et al. updated the evaluation of Clafer to consider its behavioural extension (item 11), added PyFML and VM (items 12–13) in [7], the HLVL (item 14) is incorporated in this work. It presents a brief description of the VMLs considered in this evaluation and includes the main references used in each case. When another publication was consulted, the appropriate reference is provided in the running text.

1. *Feature Description Language* (FDL) [25]: textual language that supports basic feature modelling.
2. *Forfamel* [26]: feature modelling language that is part of the *Kumbang* modelling framework [27].

3. *Batory's tree grammars*: as introduced by Batory in [28], it defines a cardinality-based feature modelling language.

4. Variability Specification Language (VSL) [29]: it is part of the Compositional Variability Management (CVM) framework which provides advanced constructs for composition of feature models.

5. *Simple XML Feature Model* (SXFM) [30]: an XML-based representation of feature models used in the software product line online tools website[1]

6. *FAMILIAR* [31]: a DSL that allows combining and analyzing feature models. Enables management of large scale variability models.

7. *Text-based Variability Language* (TVL) [32]: the first *textual* feature modelling language. It incorporates aspects that are not part of FODA [15] such as feature attributes, cardinalities and modularization.

8. *μTVL* [33]: a variation of TVL implemented for the ABS, a language for Abstract Behavioural Specification. It removes some characteristics present in TVL, but incorporates the possibility of working with multiple trees in a single model.

9. *VELVET* [34]: a TVL-inspired language with several extensions and reimplemented from scratch. It is integrated as part of Feature IDE.

10. *Integrated Variability Modeling Language* (IVML) [35, 36]: IVML is part of the *EASy-Producer* framework, which aims to provide a complete tool for managing SPLs. This framework has been developed by the *Software Systems Engineering group* at the University of Hildesheim in Germany. IVML follows the decision modeling rather than feature modelling paradigm.

11. *Clafer* [37, 38] is a language proposal combining behaviour, structure, and variability using an integrated notation. However, a subset of the language can be used to model feature and decision models. Clafer was designed by researchers from the University of Waterloo and the IT University of Copenhagen.

12. *PyFML* [39]: a simple textual feature modelling language based on and implemented in Python.

13. *Variability Modeling* (VM) [40]: a variability language designed and implemented with specific requirements for the video domain. This project was a joint effort between academia and industry.

14. *The High-Level Variability Language* (HLVL) [41]: aims to be a standard variability language that can be used as an intermediate representation of models specified in other variability languages.

[1] http://www.splot-research.org/

2.4 VML capabilities considered in this evaluation

The capabilities considered in this comparison are grouped in seven categories: *configurable elements*, *data types*, *constraint support*, *configuration support*, *language specification*, *composition support* and *tool support*. Each evaluation category is described in this section.

2.4.1 Configurable elements

Most of the variability languages support feature modelling, in which the basic configuration element is the *feature*, and features can be grouped in *feature groups*. Among these groups, *optional*, *alternative* and *multiple selection* is possible. Over time, more information was incorporated to variable items through cardinalities and attributes; new constructs such as variability extensions and references enable more expressive ways to relate different configurable elements in a variability model.

- *Optional*: A configurable element can be optionally selected.
- *Alternative*: Among a group of variable elements, one and only one must be selected.
- *Multiple selection*: More than one element can be selected from a group.
- *Extension*: It is possible to extend a base variability model.
- *Attributes*: Data attached to the basic variability unit
- *Cardinalities*: How many instances of a configurable element are allowed.
- *References*: Alias for an variable element that is not locally defined.

Similar to the concept of class extension in object-oriented programming, some variability languages allow to define a base variability model and then provide mechanisms to extend this base definition—sometimes in a different file or module. This capability enables working with open variability models and software ecosystems [42]. We encompassed all these mechanisms under the *extension* capability.

Attributes are often supported as *attached information* to the basic variability unit. These attributes usually use the numeric data types provided by the language (e.g. integers or reals), such numeric attributes are often used to model quantitative aspects of variability, enabling single- or multi-objective optimization. Sometimes strings are attached to variable elements to provide a brief description.

Cardinality-based feature models introduced the concept of *cardinalities* as an extension of feature modelling [43]. Cardinality denotes how many instances of a configurable element can be part of a configuration and it is expressed indicating a lower- and a upper-bound; lower-bound n and upper-bound m is specified as [**n**..**m**]. *Optional* ([0..1]) and *mandatory*

([1..1]) variability can be specified on atomic features (i.e. features that are not decomposed further). When applied to a feature group, alternative ([1..1]) and multiple selection (e.g.[0..*] or [1..*]) can be modelled using cardinalities. Cardinalities are more expressive than feature groups in feature modelling when dealing with multiple selection. For instance, only through using cardinalities is it possible to enforce the selection of at least two and at most seven elements in a group containing of say ten elements ([2..7]).

A number of languages support the notion of *references*. On a feature tree, references can be used to point to a feature defined in a different branch or even in a different tree (when multiple models are supported). Additionally, references allow the sharing of configurations between multiple variability models.

2.4.2 Data types

Typed variables are used to model variability (e.g. decision modelling) or to define attributes (e.g. feature models with attributes). Three different categories of data types are distinguished in [6]:

- *Predefined*: Basic data types predefined in the language. These data types are also found in most programming languages. E.g. *string*, *boolean*, *integer*, *float*.
- *Derived*: Data types derived from a basic type. E.g. *sets of predefined types*
- *User-defined*: A data type with a user-defined domain. E.g. *enumerations*.

Predefined data types are the most widely supported and mainly used to express attributes attached to variable items.

Eichelberger et al. does not provide a clear definition of the derived data types in [7], however, containers and constrained variables were identified as derived data types in their evaluation. A container is a set of configurable elements of the same type while constrained variables are those which their domain is reduced by a constraint—e.g. limit an integer to only positive numbers.

When the domain is entirely defined by the user, such as in enumerations, it is classified as a *user-defined* type. Compound data structures containing different predefined data types are considered user-defined as well.

2.4.3 Constraint support

Diagrammatic variability representations (such as feature modelling) often supports *requires* (element A *requires* element B) and *excludes* (element A *excludes* element B) constraints. Eichelberger et al. refers to these two constructs as *basic constraint support* [6]. This capability is considered supported if the language provide explicit construct to express them.

Propositional logic is supported by most of the textual VMLs to specify dependencies and constraints. Boolean values represent variants and these are related using *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$) and *implies* ($\Longrightarrow$) operators. A boolean variable appearing in a propositional formula indicates that its associated variant must be part of a configuration. On the contrary, when the same boolean variable is negated, it means that such variant must be excluded. Even though simple constraints can be represented using propositional logic, sometimes language designers include the *required* and *excludes* constructs in addition to propositional logic, just as syntactic sugar.

First-order logic incorporates the quantification operators *for all* ($\forall$) and *exists* ($\exists$) into the variability language. The former requires that a (sub-)constraint holds for *all* the configurable elements while the latter requires that such (sub-)constraint holds for *at least one* configuration.

As part of the constraint language, relational and arithmetic expressions might be supported. Relational expressions relate two variables using *equal* ($==$), *greater than* ($>$), *greater or equal than* ($\geq$), *less than* ($<$) and *less or equal than* ($\leq$) operators. Arithmetic expressions offer the possibility of deriving new values from an arithmetic formulae which might include operators such as addition or subtraction. Both relational and arithmetic expressions are only applicable to numeric or string data types and may be used in combination with propositional or first-order logic.

2.4.4 Configuration support

The basic configuration support offered in textual variability languages is the *value assignment*, i.e. relate a variable with one of its domain values. This support can be explicit (e.g. a statement to select features) or implicit (e.g. in terms of constraints).

Based on value setting operations, many languages incorporate the concept of *partial* and *complete configuration* into the language, allowing the designation of a range of value assignments as a configuration managed separately. Partial configurations enable the configuration of large variability systems in several stages (known as staged configurations [44]): the application engineer can load a configuration to set most of the variables and then fine-tune the remaining values manually. The final configuration can then be stored as a complete configuration and reused whenever needed.

Finally, it is possible to define *default values* in some VMLs. Default values can be overwritten during the model configuration, otherwise used as part of a final configuration. Similar to partial and complete configurations, default values simplify the configuration process.

2.4.5 Composition support

Scalability is essential for a variability modelling approach as the complexity of most real-world product lines will not be handled effectively if the solution does not scale [45]. This

category determines whether or not a variability language enables scalability of models by providing composition and modularization mechanisms.

2.4.6 Language specification

The degree of formality used for language specification is another evaluation criteria. Three different resources have been identified for that purpose: examples, grammars, and formal semantics [6]. Examples are the simplest way to *informally* explain how to build syntactically (structure) and semantically (intended meaning) correct models. A language grammar defines a set of rewrite rules for generating a syntactically valid program (or model). A grammar is oftentimes provided as part of the language specification to describe it more accurately than using examples only. Finally, formal semantics provide meaning to language constructs using a mathematical model.

When formal semantics are provided for a particular language, it is considered *formally specified*. If a complete grammar for the language is included in the consulted references, the formal specification is considered partial. Examples are not considered as a formal specification, and thus, marked as non-supported.

2.4.7 Tool support

As explained in Chapter 1, tool support is a crucial factor for the adoption of a new variability language, particularly in an industrial context. Most of the language specifications accompanied with tools, never go beyond the initial version while more robust solutions often are not maintained anymore after a certain point. It has been identified that tool support is crucial, particularly for large-scale systems [46]. Large-scale systems require scalable modelling approaches, which can be enabled only by the adequate tools [47]. Furthermore, this work includes a practical evaluation of a variability language using ASML's models, hence, its incorporation as an evaluation criterion.

The following four levels of tool support are considered:

- *Full support*: The following requirements shall be met
  - Syntax validation.
  - Configuration support.
  - Constraints evaluation.
- *Partial support*: Only some of the requirements defined in the previous bullet are supported.
- *Non-verifiable support* : The tools could not be installed following the provided instructions or throw critical errors when executed.

- *No support*: Only a language specification, but no tool support, is provided by the authors.

Full support is granted to a tool when three requirements are covered: *syntax validation*, *configuration support* and *constraints evaluation*. For an arbitrary model using any construct in the language specification, the tools shall determine whether or not it is syntactically correct. Syntax verification might be done by different means, e.g. live validation as part of an editor or during the compilation of the model using the command line. Configuration support means that the user can assign variants to variation points to resolve variability and derive a model instance (i.e. a product configuration). Finally, the tools shall evaluate the constraints and ensure that all the model instances satisfy them. These tree requirements ensures that an user can perform one of the elementary activities in variability management, namely, to generate a valid software configuration from a variability model.

2.5 Comparison between variability modelling languages

Table 2.1 shows the supported characteristics for each VML evaluated, the elements in green are those that were included or changed from the results presented by Beek et al. in [7]. The four levels of support and their associated notation are the following:

- *Direct support*(+): A language construct supporting the evaluated capability is clearly identified in the available literature.
- *Indirect support* (±): Certain characteristic cannot be one-to-one mapped to a language concept, yet it is supported by the language, e.g. *requires* and *excludes* constraints might not be part of the language concepts but still supported through propositional logic.
- *Unclear* (?): Indications that a characteristic is supported are found in the literature, but it is not completely clear given the lack of a detailed description. This case is common in those cases where language constructs are only explained by means of simple examples.
- *No support* (−): There is no indication that the considered aspect is currently supported by the language.

For the toolset evaluation, the notation is used as follows: Full (+), partial (±), non-verifiable (?) and no (−) support.

2.5.1 Forms of variation, extensibility and references

All the evaluated VMLs directly or indirectly support the three basic forms of variation (optional, alternative and multiple selection). Most of the specifications support cardinalities

VML	forms of variation							data types			constraint expressions					configurations						
	optional	alternative	multiple	extension	Attributes	cardinalities	references	predefined	derived	user-defined	simple	propositional	first-order	relational	arithmetic	default values	assign values	partial	complete	composition	formal spec.	tool support
FDL	+	+	+	-	-	-	-	-	-	-	+	-	-	-	-	+	±	-	-	-	±	-
Forfamel	+	+	+	+	+	+	+	-	-	+	-	+	+	+	+	-	+	-	+	-	±	-
Tree-grammars	+	+	+	-	-	+	-	-	-	-	-	+	-	-	-	-	±	-	-	-	-	-
VSL	+	+	+	+	+	+	+	+	-	+	+	?	?	-	?	+	+	+	+	+	-	?
SXFM	+	+	+	-	-	+	-	-	-	-	-	+	-	-	-	-	±	-	-	-	-	+
FAMILIAR	+	+	+	-	-	-	?	+	+	-	-	+	-	-	-	-	+	+	+	+	-	±
TVL	+	+	+	?	+	+	+	+	-	+	-	+	-	+	+	-	+	-	-	-	+	±
μTVL	+	+	±	+	+	+	-	+	-	-	+	+	-	+	+	-	?	+	+	-	+	+
Clafer	+	+	+	+	+	+	+	+	+	+	-	+	+	+	+	-	+	+	+	+	+	+
VELVET	±	+	+	+	+	±	+	+	-	-	-	+	-	+	-	-	+	+	±	+	-	-
IVML	+	+	+	+	+	±	+	+	+	+	-	+	+	+	+	+	+	+	±	+	-	+
PyFML	+	+	+	-	+	+	-	+	-	-	+	+	-	+	+	+	+	-	-	-	-	±
VM	+	+	+	-	+	+	-	+	-	-	+	+	-	+	+	+	+	+	±	+	-	?
HLVL	+	+	+	-	+	+	-	+	-	-	+	+	-	+	+	-	+	+	+	-	-	?

Table 2.1: *Variability language capabilities and supported characteristics based on Table 1 in [7]. Added or updated values are shown in green*

and attributes while around half of them incorporate variability extension mechanisms and references.

FDL supports optional, alternative and multiple selection explicitly. It incorporates the selection constraints *one-of* (one and only one) and *more-of* (one or more) for feature groups [25]. Eichelberger et al. state in [6] that these constraints on feature group constructs imply certain lower an upper bounds, hence, cardinalities are implicitly supported. However, these two constraints represent the cardinalities [1..1] and [1..*] only. We consider that cardinality is supported if it is possible to specify arbitrary bounds, which is not the case for this language, and hence unsupported. Neither data types nor references nor attached information are supported by FDL.

Forfamel supports the basic forms of variation using cardinalities. Features are represented using extendable types and can be referenced. Further, each feature has an arbitrary number of attributes, such attributes might be numeric or a user-defined set of values (i.e. enumeration).

Batory's tree grammars (abbreviated as *tree-grammars* from now on) explicitly express *optional* and *alternative* selection as well as cardinalities [28]. Multiple selection is indirectly supported using cardinalities. Data types, references and attached information are not supported. **SXFM** follows a rather similar approach, with the only difference being that it only supports feature groups with numeric cardinalities [30].

VSL supports cardinality based feature modelling; Optional and alternative selection are incorporated explicitly while multiple selection using cardinalities. A special kind of references referred to as *configuration links* are supported. Using these configuration links the config-

uration for a target feature model can be derived from a source configuration model; these links define a mapping between source and target models, allowing automatic configuration derivation [48].

FAMILIAR does not support cardinalities or attached information, only simple constraint constructs. When dealing with complex data types (like sets), that information is stored as a *reference* rather than values (akin to Java) [31], however, there is no clear indication that references to arbitrary data types are supported.

TVL supports *optional* and *alternative* selection explicitly and *multiple* selection using cardinalities attached to feature groups. According to the examples in [32] forward definitions are allowed using the "include" keyword, but it is not explicitly indicated if later extensions and refinements of feature models are supported.

μTVL offers almost the same support that TVL regarding the forms of variation. However, TVL specifies multiple selection through the `someOf` construct while μTVL indirectly supports it using cardinalities. Further, references are not supported in this language proposal.

Clafer supports all the forms of variability explicitly. Extension is possible through inheritance using *abstract* clafers, which can be used as an extensible class. Besides feature groups, cardinalities can be attached to abstract features, limiting the number of concrete clafers that can be instantiated. References can be defined to any arbitrary data type (including references to clafers). Integers and Strings can be used as attached information.

Features are by default *optional* in **VELVET**. The keyword `mandatory` must be used to denote non-optional features, thus, optional is considered as indirectly supported. The base feature model definition can be specified in a file, and refined in different files—allowing extensibility. Group selection might be constrained using *someof* ([1..*]) and *oneof* ([1..1]), but arbitrary lower and upper bounds are not supported. It is possible to defined named constraints in VELVET, which can be then referenced from other parts of the model.

IVML represents variability using typed decision variables. Decision variables are grouped using compounds—a container type similar to structs or records in programming languages. Extensibility is supported through compounds refinement. Using the `refines` keyword, additional elements can be added to a base compound creating an extended version—akin to subclassing in object-oriented languages [36]. Cardinalities are indirectly supported in containers (sequences and sets), where the number of selectable elements from a container can be restricted using attached constraints. Further, user-defined typed variables (meta-attributes) can be added to decision variable attributes. These meta-attributes allow the same level of expressiveness as decision variables and can be used to add any kind of variability information, e.g. binding time [7].

PyFML enables all the basic forms of variability through cardinalities, which are explicitly supported on basic features. Beek et al. assessed cardinalities as non-supported for PyFML in [7] without any rationale behind that decision. This evaluation was changed after consulting [39] where it is explicitly mentioned that instance cardinalities are supported. Arbitrary attributes can be attached to features—boolean, integers, floating and string values. Yet,

it is not necessary to specify the data type in the attribute declaration—dynamic typing is supported as in Python.

VM expresses features trees in the so-called "Relationship" block, where the hierarchical and group relations between features are defined. Cardinalities are explicitly supported for feature groups and individual features as well. Cardinalities can be specified in single features to support multi-features (also called clone enabled features), specifying a lower and upper bound of the number of clones a such feature allows. Attributes are defined in the "Attributes" block and associated with individual features defined in the Relationships block. These attributes might be of type boolean, integer, float, string and enum.

HLVL enables basic forms of variability through cardinalities. Two different kind of attached information are supported: *attributes* and *comments*. Attributes might be boolean, integer and symbolic; symbolic attributes are represented by a set of strings, what is commonly know as enumerations in other languages. Integer attributes might be bounded to a range of values or set to a fixed value.

2.5.2 Type system support

The specific data types supported by each VML are listed in Table 2.2. Additionally, it is identified where these typed variables are used on each case.

The first observation is that the majority of the evaluated VMLs provide some predefined data types, excepting FDL, Forfamel, Tree grammars and SXFM. Among the languages with predefined data types, boolean, strings, integers and floating point[2] types are commonly supported. μTVL, Clafer and HLVL do not support floats, and Clafer does not support booleans either. The languages which do not support predefined data types use feature and feature group constructs (sometimes referred to as features *types*) only. Attributes are not supported by any of the VMLs without predefined data types support.

IVML provides two kind of container types: sequences and sets. Container elements can be of any supported data type. Both sequences and sets can contain an arbitrary number of elements; sequences can contain duplicates while sets cannot [36]. **FAMILIAR** provides sets that store *references* to variables of any type [31]. An important difference between IVML and FAMILIAR regarding containers is that IVML stores data while FAMILIAR only store references—all the variables contained in FAMILIAR's sets need to be previously declared. Additionally, IVML supports derived data types by aliasing predefined types using *typedef*—akin to C/C++. Additionally, a constraint can be used to restrict the values of the derived types. For instance, an arc degree type can defined as `typedef degree Integer with (degree >= 0 and Degree <= 360)`

According to [6] and [7], **Clafer** support sets (i.e. containers). However, sets are internally used during the instance resolution process but not available as a language construct for the end-user [49]. Thus, Clafer sets are considered non-supported in our assessment. However,

[2] *Real* or *Float* is used depending on the language's keyword. Both represent floating-point data types.

Language	predefined	derived	user-defined	Applies to
FDL	-	-	-	-
Forfamel	-	-	Enum	Attribute, user feature
Tree Grammars	-	-	-	-
VSL	Boolean, Integer, Float, String	-	Enum	Parameter, user attribute
SXFM	-	-	-	-
FAMILIAR	Boolean, Integer, Real, String, Enum, model, configuration	container	-	Feature, model, configuration
TVL	Boolean, Integer, Real	-	Enum, Struct, constant	Attribute
μTVL	Boolean, Integer	-	-	Attribute
Clafer	Integer, String, clafer	constrained basic type	enum, clafer as a compound	clafer
VELVET	Boolean, Integer, Float, String	-	-	Attribute
IVML	Boolean, Integer, Real, String	container, typedef	Enum, compound	Decision variable, meta-attribute
PyFML	Boolean, Integer, Float, String	-	-	Attribute
VM	Boolean, Integer, Real, String	-	Enum	Attributes
HLVL	Boolean, Integer, comment (string)	-	Symbolic (Enum)	Items, Attribute

Table 2.2: *Type system support based on Table 3 in [6]. Updated and added values are in green*

similar to IVML, Clafer can define derived data types by relating constraints to integer variables to limit their domain. Eichelberger et al. indicates in [6] that data types are applied to *features* in Clafer which is slightly inaccurate. The concept of feature does not exist in this language, but *clafers* can be used to model features; hence, the data types are applied to clafer. Both the lack of sets support and the application of data types was updated in Table 2.2.

User-defined data types are mainly supported through enumerations as observed in Table 2.2. Besides, compound data types are also supported by TVL, Clafer and IVML. IVML's *compound* data type groups multiple type variables into a single element. Clafer and TVL use *clafers* and *structs* for the same purpose. These compounds allow combining semantically related variable elements during the modelling process. Clafer and IVML support compound nesting (compound as a member of another compound), which allows to naturally represent a system's decomposition during the modelling process. Additionally, TVL provides the *const* qualifier which sets a variable to a fixed value that is not allowed to change (semantics are equivalent to *const* in the C language).

Typed variables are primarily used to define attributes. Some languages like FAMILIAR, IVML, Clafer and HLVL also use typed variables to model variable elements. FAMILIAR use the so-called complex data types to represent feature models (e.g. *Feature model*, *Feature*,

Constraint are complex types). IVML follows a decision modelling paradigm. Variability is modelled through decision variables and can be optionally associated to an arbitrary number of attributes (know as meta-attributes). All the supported type variables in IVML can be used in both decision variables and meta-attributes. Clafer uses clafers (compound types) as basic modelling unit which can contain basic-type variables (integer and string) to model feature attributes.

It is noticeable that FDL, Tree grammars and SXFM do not support any type variables. This is because typed variables are used to represent attributes in feature modelling, and none of these languages support attributes. These VMLs support textual representation of feature diagrams in their original form (i.e. as proposed in FODA [15]), providing only the necessary constructs to represent features, feature groups (*AND-*, *OR-* and *XOR-groups*) and simple constraints (*requires* and *excludes*).

2.5.3 Constraint support

Almost all the languages considered support propositional logic for constraints specification; except FDL which only supports simple dependencies. Language designers often decide to integrate redundant constraint constructs as syntactic sugar for propositional logic, resulting in more concise constraint expressions. For instance, *requires* and *excludes* constraints (simple dependencies) are a special case of propositional logic and it is not necessary to support both, however, μTVL, PyFML, VM and HLVL do it.

Forfamel uses the *Kumbang Constraint Language* [50], which supports first-order logic, relational, arithmetic and cardinality expressions. This constraint language provides functions that allows to obtain the number of instances or the presence of a particular feature.

VSL provides an extended set of simple dependencies (needs, excludes, alternatives, suggests, impedes). Additionally, simplified propositional formulae, and arithmetic expressions are supported in constraints.

Tree Grammars, **FAMILIAR** and **SFXM** provide full propositional logic. Beek et al. evaluates SFXM propositional logic support as *unclear* in [7]. Whether or not is supported cannot be concluded from the SFXM reference consulted [30]; however, after evaluating the associated toolset, it was clear than propositional logic is supported by the language.

TVL, **μTVL** and **VELVET** specify constraint expressions in terms of propositional formulae with relational expressions for integer attributes. Besides, **μTVL** provides the constructs "require" and "exclude" for simple dependencies.

Clafer constraints are syntactically close to those of Alloy (one of its backend solvers); it supports first-order logic including propositional clauses, relational and arithmetic expressions. It is indicated by Beek et al. in [7] that simple constraints are incorporated into Clafer. However, no indication of simple constraint constructs were found neither in the consulted

references nor in the language grammar[3]. Eichelberger et al. also consider simple constraints unsupported by Clafer in [6].

IVML's constraint language is built upon the Object Constraint Language (OCL) [51] with additions for default values and value assignments. Thus, IVML also supports first-order logic on containers in combination with propositional, relational, and arithmetic expressions. Both Clafer and IVML use constraints for the restriction of existing types.

PyFML, **VM** and **HLVL** support propositional logic with relational and arithmetic expressions but no first-order logic. VM provides a unique construct called *delta* values. A delta reduces the domain of a numeric variable. For instance, consider the expression `int vehicle.speed [0..130]` indicated that the variable `vehicle.speed` can take values between 0 and 130 (it is a constrained integer variable). However, the following expression `int vehicle.speed [0..130] delta 5` is internally interpreted by the tool as `int vehicle.speed [0, 5, 10, ..,130]`. VM's delta is construct used to define a set of linear constraints in a concise way, which is not found in any other of the considered languages.

2.5.4 Configuration support

Besides configurable elements, their attributes and dependencies, a VML should provide constructs for defining configurations. The main mechanisms for configuration definition is the value assignment. Moreover, a variability language may also provide support for defining default values, partial or complete configurations. Default values are taken by variables unless a different value is set during the configuration. Partial and complete configurations allow the user to assign a value to a set of the model's variables at once.

ter Beek et al. marked all the configuration-related capabilities for **Tree-grammars** and **SXFM** as non-supported in [7] without providing any rationale for this conclusion. This lack of configuration support effectively means that these languages are not able to resolve variability from the model, and therefore product configurations cannot be derived. This was a clear indication of an inaccuracy because deriving product configurations is an mandatory requirement for any SPL. After diving into the references it was found that value assignment is *indirectly* supported by both languages.

Tree-grammars language does not support value assignment explicitly. However, variables' domains can be restricted through propositional logic expressions, providing an implicit mechanism for variable assignment.

SXFM language does not support any configuration construct as part of its definition. The S.P.L.O.T website[4], which uses SXFM as a backend, provides a visual configuration tool. The user can select values for all the variabilities defined in the model using the configurator. Then, the user can generate an output file in XML or CVS containing pair-wise values.

[3]https://github.com/gsdlab/clafer/blob/master/src/clafer.cf
[4]http://www.splot-research.org/

FDL supports value assignment (using the *include* constraint) and the possibility of defining default values within a feature group. However, based on the specification published in [25], it is unclear how the conditional value assignment, i.e. the default value, is implemented. The satisfaction rules listed in Fig. 10 in [25] shows that the default construct is translated as *true* and added to the constraint expression using a conjunction, which suggests that the conditional assignment of default values have no impact in the satisfiability problem.

Forfamel allows the user to specify of values through constraints. Even though it does not support default values, it is possible to specify complete configurations using the Kumbang Configurator [27]. The Kumbang configurator is a configuration tool for SPLs that uses the Kumbang language. Even though Forfamel and Kumbang are different languages, both are part of the Kumbang modelling framework and thus, complete configuration capability is considered covered.

FAMILIAR enables the explicit creation of configurations for a given feature model through configuration variables. A specific configuration allows the selection and deselection of features. Although not explicitly stated in literature, these configuration variables could be use to define partial configurations in FAMILIAR.

TVL, **μTVL** and **VELVET**, **PyFML**, **HLVL**, **VM** and **Clafer** support assignments based on constraints, which implicitly fix a configurable element (e.g. attributes or variation points) to a value. Value assignments are done, in all cases, as part of a boolean expression using the equallity operator (==). Besides basic assignments, VELVET and Clafer enable partial and complete configurations through inheritance and constraints. μTVL uses the *Product Selection Language* (PSL)—a separate language specification—to select features and set attributes of those features, which provides the capability to specify partial and complete configurations as well.

It is reported by ter Beek et al. in [7] that VELVET supports default values, however, we did not found any evidence of that in [34].

2.5.5 Composition mechanisms

In large variability models containing thousands of variabilities composition mechanisms support is crucial. Modelling a complex system on a VML that does not support composition is similar to develop their software in an object-oriented language using a single class definition. Six out of the fourteen VMLs considered in this work support a composition mechanism, namely, VSL, FAMILIAR, VELVET, Clafer, IVML and VM.

As in most programming languages, composition is achieved through a modular design. Modules are built independently, then, these can be referenced and used as part of a new module's definition. TVL and IVML support this form of composition. TVL provides the keyword *include*, akin to C/C++, which includes the contents of the referenced file into the current specification. Under the hood, the *include* statement is replaced with the file's contents, a simple yet efficient mechanism that allows the users to work on a modular design.

Similarly, IVML provides the *import* statement, which allows the user to import an existing project to the current one. IVML's *import* provides more advanced functionality than TVL's *include* statement. IVML toolset verifies a project's consistency before importing it; if it contains errors, the import operation fails. Furthermore, IVML provides the keyword *conflicts* to explicitly indicate incompatibility among projects. The *conflicts* annotation specifies that two projects cannot be composed (i.e. it is not possible to import one project into the other), it is even possible to mark specific project's versions as incompatible.

Greatly inspired by object-oriented languages, VSL, Clafer and VELVET support *inheritance*, allowing hierarchical composition of variability models. Inheritance is the mechanism of basing an object (child object) upon another object (parent object). The child object, in an object-oriented program, acquires all the properties and behaviours of the parent object. Analogously, the derived object in a variability specification inherits all the variable items and constraints from the base object.

VSL is part of the Compositional Variability Management (CVM) framework which supports the combination of several feature models into a global variability model. The framework provides advanced constructs to achieve modularity like *feature links* (for setting cross-tree dependencies among feature models), *configuration links* to derive model configurations from a source to a target model (i.e. sub-models in the composition) and the possibility of setting public and private variability elements (to restrict data access between models). The language borrows and adapts several concepts from OO languages, like "interface for variability" and the possibility of specifying private and public elements in a feature model to restrict data access [48].

Clafer supports composition on type level (clafers are used as a containment structure) and on model level via inheritance. A clafer is used to contain variabilities (optional elements or groups of elements with different cardinalities) and define clafer context constraints(i.e. valid inside the clafer defintion). The variability model can be split into different clafers; each clafer can be referenced from another one, which enables design modularity. Furthermore, a clafer definition can extend or specialize an existing definition (called super clafer), giving the possibility of defining a hierarchical structure and a layered design. Clafer defines *abstract* and *concrete* clafers, which can be intuitively and approximately understood as classes and objects, respectively. A concrete clafer is an instance of an abstract clafer as an object is a class instance. Concrete clafers can be inherited from abstract and concrete definitions, while abstract clafers only allows abstract clafers as the base definition.

Similarly, VELVET enables single inheritance, allowing to extend an existing definition by adding new features and constraints. Besides, VELVET supports multiple inheritance, which occurs when an object is derived from multiple base definitions. Multiple inheritance needs to merge the base feature models first, which requires (1) the union of all features and (2) the union of all constraints [34].

2.5.6 Formal semantics

Clafer, TVL and μTVL provide a formal language description while FDL and Forfamel provide a partial formal description. Clafer, TVL and μTVL provide a set of mathematical definitions as a formal specification in [37], [32] and [33], respectively. FDL provides a set of normalization, expansion and satisfaction rules. Normalization and expansion rules are applied to the model to resolve variability, ending with an expression in a disjunctive normal form representing the set of all possible configurations. Model constraints can be interpreted using the satisfaction rules, yielding true or false for every disjunct defined in the previous step. These rules provide, to some extent, a formal description of FDL's semantics. and thus, indirect support was granted to this language. Similarly, Forfamel provides a translation to the Weight Constraint Rule Language (WCRL) [26]. A mapping from a feature model with constraints to a constraint satisfaction problem is also considered an indirect description of Forfamel formal semantics.

2.5.7 Tool support

The tool support offered by each VML considered in this evaluation is listed below:

- *Full support*: **SXFM**, **μTVL**, **Clafer** and **IVML**
- *Partial support*: **FAMILIAR**, **TVL** and **PyFML**
- *Non-verifiable support*: **VSL**, **VM** and **HLVL**
- *No support*: **FDL**, **Forfamel**, **Tree-grammars** and **VELVET**

Full-support

SFXM is used as the backend language for the S.P.L.O.T (Software Product Line Online Tools) website (see Figure. 2.1) , which provides full support for the variability language. The online toolset allows the user to specify feature models with constraints and provide some feature model statistics, e.g. total number of features and cross-tree constraints. Feature model analysis is also provided and includes a model's consistency, its number of core and dead features, and the number of valid configurations. The model is consistent if it has one or more model instances, and it is inconsistent otherwise. Core features are those which appear in all model instances while dead features do not appear in any (i.e. cannot be selected due to constraints). The number of valid instances is only provided for small models (less than 100 features). For medium size models (between 100 and 300 features) only an indication is shown (e.g."more than a million instances"). When big models (more than 300 features) are analyzed, the analysis keeps running indefinitely or it shows the message "the model is too big".

[5]Link: https://drive.google.com/file/d/1TmxB4fR4oTeBgk4Z6vbfr0R5xEkMUrJ7/view?usp=sharing

VML	Toolset related links
SXFM	Online toolset: www.splot-research.org
μTVL	ABS project page: http://abs-models.org Toolset installation (Docker): https://abs-models.org/getting_started/docker/
IVML	Project page: https://sse.uni-hildesheim.de/en/research/projects/easy-producer/ Eclipse update site: https://projects.sse.uni-hildesheim.de/easy/ Source code repository: https://github.com/SSEHUB/EASyProducer
Clafer	Project page: https://www.clafer.org/ Software installation: https://www.clafer.org/p/software.html Documentation (including wikipage): https://www.clafer.org/p/documentation.html Source code repository: https://github.com/gsdlab/clafer
VSL	Project page: http://www.cvm-framework.org/ Eclipse update site (broken): http://swt.cs.tu-berlin.de/ moreiser/eclipse/
FAMILIAR	Project page: https://familiar-project.github.io/ Binaries installation: https://github.com/FAMILIAR-project/familiar-documentation/tree/master/installation
TVL	Project page and tool download: https://projects.info.unamur.be/tvl/
PyFML	Python script source: Google Drive link [5]
HLVL	Source code repository: https://github.com/angievig/Coffee/tree/master/HLVL
VM	Project page: https://mao2013.github.io/VM/ Eclipse update site: http://mao2013.github.io/VM/vmUpdateSite/

Table 2.3: *Relevant toolset links for each VML. Last consulted: 27/06/2020*

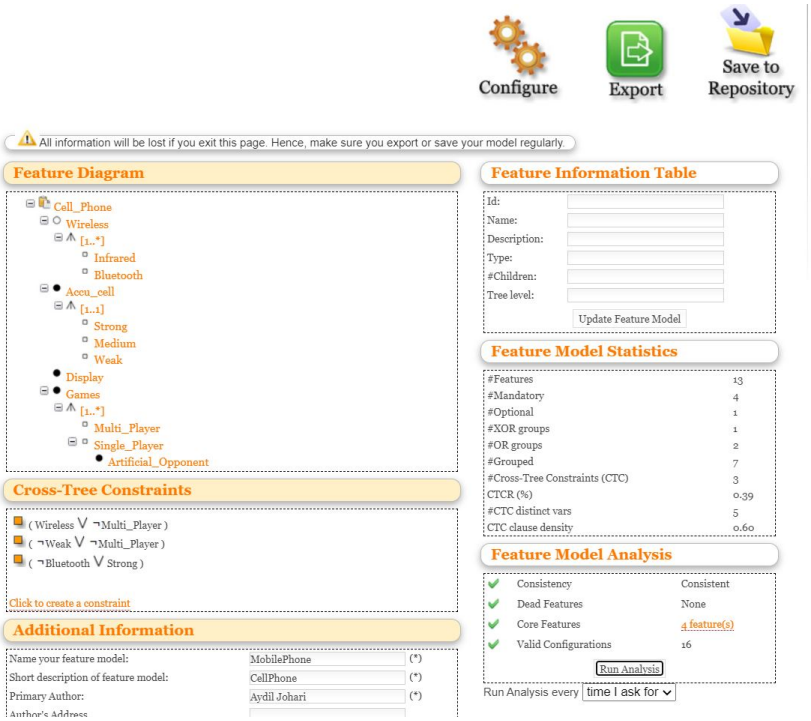

Figure 2.1: *S.P.L.O.T. feature model editor. Website: www.splot-research.org*

Furthermore, the S.P.L.O.T website keeps a repository of user's feature models (1,774 as of the writing this work), and it was used for the tool's evaluation. Besides the feature model

analysis, the website includes a section named *Automated Analysis* with more advanced analysis capabilities, however, that functionality is currently broken. This site has been in development since 2009, but it has not been updated since 2015.

μTVL is part of the Abstract Behavioral Specification (ABS) language, which provides a toolset that fully supports μTVL. For our testing, the browser-based IDE locally running on Docker[6] was used. This installation provided a set of examples that were used for this evaluation. The main goal of the ABS language is to specify behavioural models and μTVL is used only when variability is needed, thus, only a subset of the examples include feature models.

IVML is part of *EASy-Producer*, a framework for managing SPLs developed by the Software Systems Engineering group[7] at the University of Hildesheim in Germany. IVML and its toolset have been in development since 2011; the last published paper and tool's update date from 2018.

EASy-Producer is deployed as an Eclipse plugin. An IVML editor is included as part of the plugin, which provides code highlighting and automatic syntax checking (See Figure 2.2). A detailed manual[8] shows how to setup an example project using the framework. Besides the example used in the manual, the Eclipse installation provides several reference projects from which the `PL_SimElevator` was also used during our tool's assessment.

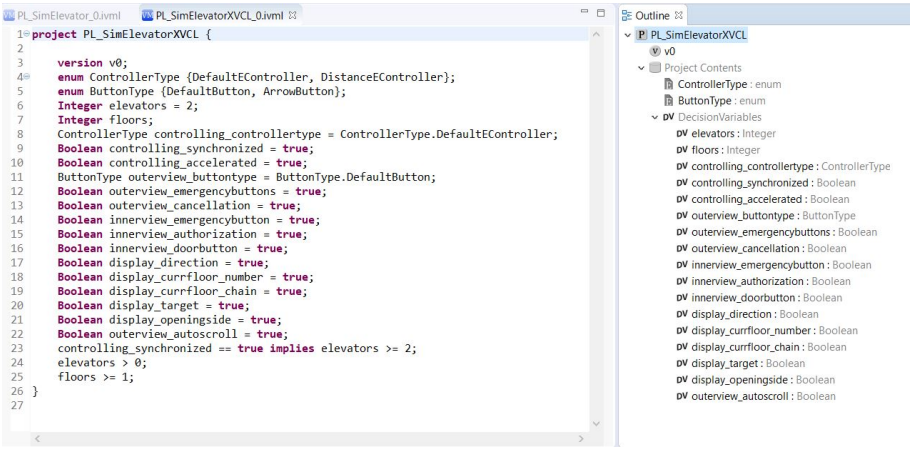

Figure 2.2: *EASy-Producer, IVML editor*

The framework not only allows to specify variability through IVML files but also supports the instantiation process of the software product line. Instantiation is supported by providing specific approaches to use the variability information to build a product from a set of artifacts. For instance, the Velocity Instantiator—which is part of EASy-Producer—adds pre-processor functionality (similar to C\C++) to Java code. In the Figure 2.3, lines 5–6 shows a Java snipped containing reference to variable elements. The references use the currency symbol ($).

[6]https://abs-models.org/getting_started/docker/
[7]https://sse.uni-hildesheim.de/en/
[8]Available here: https://projects.sse.uni-hildesheim.de/easy/docs-git/docRelease/user_guide.pdf

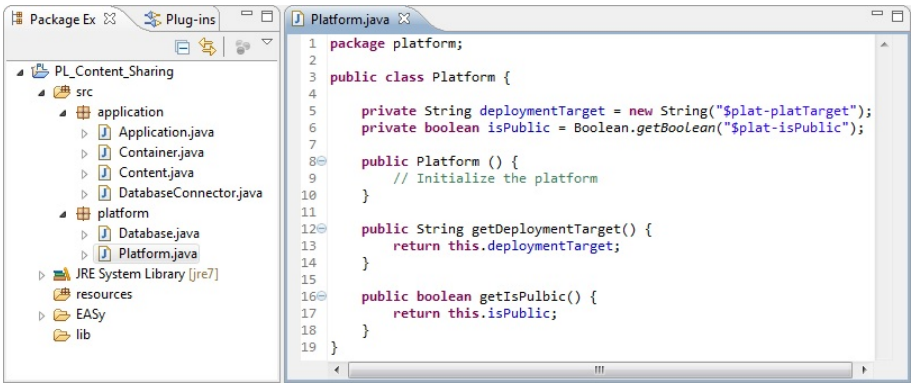

Figure 2.3: *EASy-Producer, VIL build script example*

The instantiation steps in the software product line are specified using the Variability Implementation Language (VIL). The VIL is a textual language for the specification of build information and tasks executed during the instantiation process. Fig. 2.3 shows an example VIL build script, this script specifies source and destination folders (lines 5–6), clean up tasks (lines 8–12) and the instantiator selection (line 15). VIL specification is independent of IVML.

Even though EASy-Producer is a prototypical tool, it offers an Eclipse-based integrated framework for managing SPLs. Beyond variability specification and resolution using IVML, it offers some product building mechanisms—like the Velocity Instantiator or the VIL build scripts—which none of the other analyzed tools included. The IVML specification as well as its associated tools have evolved throughout the years. The first attempt to install the Eclipse plugin failed due to a server issue, however, after reaching out to the developers at the University of Hildesheim, the problem was fixed in a couple of days.

Clafer development—for both the language and its toolset—was a joint effort by the GSD group[9] (now WISE lab[10]) at University of Waterloo and the former MODELS group [11] at IT University of Copenhagen. Clafer was actively developed from 2012 until 2018; researchers and students (both Masters and PhD) ascribed to these educational institutions were involved. Now the project is considered "finished". The following assessment applies to the Clafer 0.4.5 release.

The Clafer toolset has two core elements: a compiler and an instance generator. The compiler transforms a Clafer model to different output formats. The output formats can be used by the instance generator (.als file for Alloy and .js file for Chocosolver), as an intermediate representation (JSON) or by visualization tools (html or dot files). Clafer model instances are obtained using the instance generator. Optionally, optimal instances can be derived using this generator. These optimal instances are derived based on specification goals defined in the model. These goals are defined using functions to maximize or minimize a numeric attribute

[9]https://gsd.uwaterloo.ca/
[10]https://uwaterloo.ca/waterloo-intelligent-systems-engineering-lab/
[11]Broken link:https://www.itu.dk/research/models/wiki/index.php/Process_and_System_Models_Group

defined in the model.

Clafer's tools are provided as web-based or binary distributions. The web-based tools can be run locally or online—hosted by the University of Waterloo. Pre-built binaries are available for Windows, Linux and OSx (with exception of release 0.5.0 which only provides Linux binaries); build from source code is also an option. Additionally, an implementation of Clafer in JetBrains Meta-Programming System (MPS) is also available. The web-based toolset includes:

- **Clafer IDE**: An IDE for Clafer including an editor, compiler, and instance generator.
- **Clafer Configurator**: Allows to work with multiple configurations at the same time. It provides a better visualization for configurations (See Fig.2.4).
- **Clafer MOO Visualizer**: It allows to visualize and analyze tradeoffs among several model instances using the multi-objective optimizer.

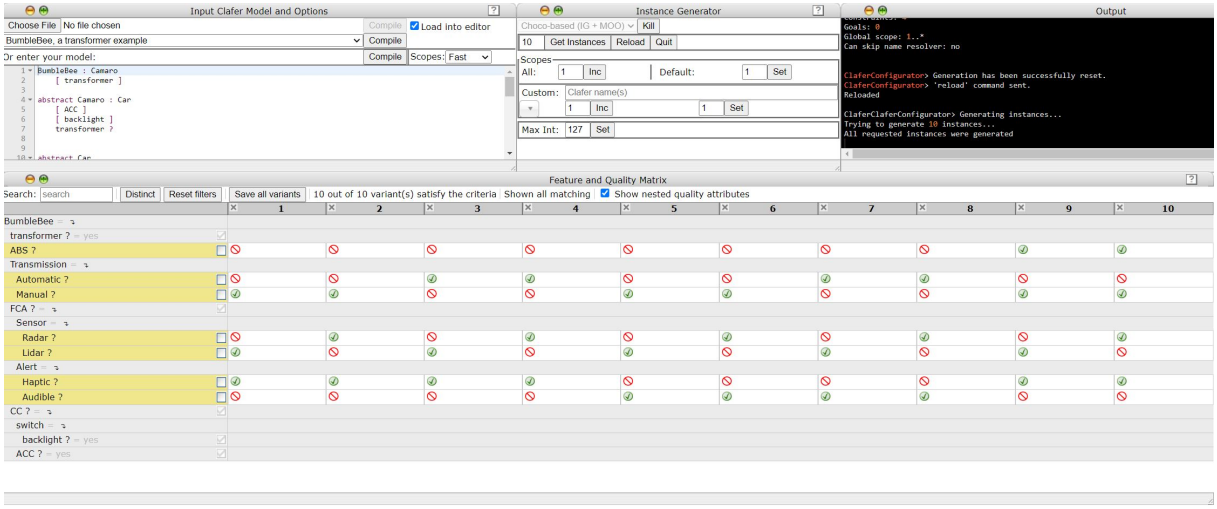

Figure 2.4: *Clafer configurator (online tool)*

Similar to IVML and the EASy-Producer framework, Clafer visual tools improve user experience compared to its counterparts which only provide command-line tools. Additionally, Clafer's compiler and instance generator can also be run from the command-line and store the output to a text file, which is particularly useful when working with large models.

An extensive documentation accompanies the Clafer toolset. The documentation comprises 21 academic publications (including a PhD dissertation), tutorials and a Wiki page with a repository of examples (see link to documentation in Table 2.3). These documents detail the language design, tools usage, and illustrate useful design patterns by modelling different use cases. The tool's installation documentation details all the steps for installation of the tools on different platforms. As part of the tool's assessment, the browser-based version—which can be run locally using a Redis server[12]—was successful installed in a Windows and a

[12]https://redis.io/

Linux machine by following the instructions provided in their website. Among all the VMLs considered in this evaluation, Clafer is by far the most exhaustively documented.

Behavioural modelling support constructs were integrated in release 0.5.0 of the Clafer compiler. It is mentioned by the Clafer designers in [37] that this release can output desugared clafer models and generate HTML. Furthermore, it is stated that "an experimental generator of Alloy input is also included in the release" [37]. We found the description of "experimental" ambiguous, so, we tested this version of the Clafer compiler. To do that, we used the pre-build binaries provided in the website[13] using a Linux computer. The compiler generates an Alloy file as output, however, this file contains errors. We used the behavioural models available in the Wiki page and examples in [37]; alloy files with errors were found in all cases. Clafer developers confirmed by email that the Alloy translation is indeed not working. Therefore, the translation from Clafer models to Alloy, described as *experimental*, is better described as *not working*. It would be interesting to investigate what is missing exactly to get it to work, but that was not assessed as part of this evaluation.

IVML and Clafer toolsets have been developed iteratively, delivering new features and bugs fixes in several software releases through the years. Source code is available as an open-source project (Table 2.3 includes links to their code repositories). EASy-Producer uses Apache License 2.0 while Clafer is licensed under the MIT License. Both licenses are almost equally permissive. Besides allowing modification and redistribution of the source code, private and commercial use is permitted with no obligation of sharing the modified source code. These licenses would facilitate industry adoption, since many companies do not want to be forced to release their modified source code.

Partial-support

FAMILIAR, **TVL** and **PyFML** languages provide tools with partial support. According to the project website, FAMILIAR is integrated into Feature IDE—an Eclipse-based framework used to create feature models using visual notation—however, the FAMILIAR plugin has been broken since 2013 (as stated in the FAMILIAR README file[14]). Thus, only the command line tools were tested. A reference TVL parser in Java (deployed as a JAR file) is available in the project page. Finally, PyFML tool is implemented in a single Python script shared through Google Drive (see Table 2.3 for the link). PyFML implementation is by far the most minimal implementation among the evaluated tools.

The FAMILIAR command line tool starts an interactive session on the terminal using the input model, which allows the user to manipulate the input model further (e.g. merge or split different FMs, assigning variables, or query configuration information). This tool was tested using the examples presented in [31]. FAMILIAR prototypical implementation uses Xtext—a framework for developing DSLs—which automatically generates a parser from a grammar description. Syntax error messages on the command line tool are those directly provided by the Xtext parser, which are verbose and difficult to understand when the user is

[13] https://gsd.uwaterloo.ca/clafer-tools-binary-distributions.html
[14] https://github.com/FAMILIAR-project/familiar-documentation/tree/master/installation

not familiar with the Xtext output. Model configurations can be validated explicitly using the `isValid` command.

A TVL parser is also provided as a command line tool. It supports most of the language constructs but unlike FAMILIAR, no interactive terminal session is supported. The syntax error messages lack useful information. For instance, if an unknown data type is used, the error message describes the problem accurately but it does not show the line where the issue was found. If a semicolon is missing, it causes a Java exception pointing to the problematic line but without any description of the problem. The tool offers the possibility of checking model constraints' satisfiability; however, it does not support numerical (int or float) values. If the TVL model contains numerical attributes, its satisfiability cannot be checked.

PyFML script is provided with no documentation. It has two Python dependencies that must be installed manually (xtext and python-constraint modules). The script does not read the input model from the command line, it should be added directly into the Python script as an object definition. The minimal example included in the script yields six different product configurations. However, the tool throws three exceptions while parsing the example provided in Listing 8 found at [39]. Since the test example used for evaluation was obtained from the language's paper and the errors found were related to the parser implementation, this indicates that the tool has some critical bugs; thus, it was not investigated further.

FAMILIAR and TVL provide a prototypical command line tool that can be used to model small examples and provide insights about the language usage. However, at this early stage of development, the lack of integration with existing tools and precarious debugging information make them challenging to use in an industrial setting. Even more challenging, PyFML's parsing issues make it incapable of dealing even with toy examples, so its industrial application is out of the question.

Non-verifiable support

VSL, VM and HLVL claim tool support, but it could not be verified. VSL is part of the CVM framework, which is deployed as an Eclipse plugin. However, the Eclipse update site provided in the project page (see Table 2.3) is broken. VM tool is also an Eclipse plugin and its update site is functional. However, dependency errors were found during the installation—apparently some plugin files are missing in the server—which prevented its completion. Finally, HLVL tool is implemented in Java and its source code is available in a code repository. The repository does not provide pre-built binaries or build instructions.

In all these cases, the contact people (taken from the project page or the paper with the language proposal) were contacted asking for support but we did not get a reply. Additionally, older Eclipse versions were used as an attempt to get VM and VSL installed—sometimes plugins have dependencies to a particular Eclipse version. None of these strategies helped to fix the installation issues.

No support

FDL, **Forfamel** and **Tree-grammars** explicitly state in their presenting papers that tools are not provided. Supposedly, VELVET was integrated into FeatureIDE. VELVET examples were opened using FeatureIDE to find that there is no support for them. VELVET files are opened as generic text files with neither code highlighting nor syntax check; the model's consistency verification is also missing. After contacting VELVET authors, they pointed out that it was possible to convert VELVET to the current FeatureIDE internal representation. Following the import procedure described in [52], it was found that VELVET is no longer supported as a valid import format. Therefore, VELVET does not provide tool support (anymore).

2.6 Variability management aspects relevant for CPSs

Besides the capabilities already explored, ter Beek et al. mention in [7] the concepts of *binding time*, *dynamic software product lines (DSPL)* and *multi-objective optimization* as other important concerns regarding language design that are not considered in the evaluation presented in Table 2.1. Binding time and DSPL introduce dynamic aspects of variability, while multi-objective optimization is a valuable configuration support mechanism when dealing with large and complex configurations. For these reasons, these three aspects are relevant in the context of cyber-physical systems.

2.6.1 Binding time

Obtaining a valid product configuration may not be the only concern in the variability management process; *when* it happens is also relevant in certain domains. Two different points in time are clearly identified: the *definition time*, when the system configuration is *determined*, and the *binding time*, when it is *applied* [7]. The configuration or binding of variants might occur at three different moments: design-, build-, and runtime. Build time generally refers to the compilation process where variability is resolved using compiler parameters, and it is effectuated through conditional compilation—the term build time or *compile time* is often used interchangeably. Design time requires that the variability (in source code or models) is resolved before the executables are generated. When the product configuration is known before compilation, variability can be resolved and variants selected at design time. Runtime is defined as any moment after the system's power-up, when the software is running. The *configuration definition* normally takes place during the design or build-time. This configuration might be applied to the system very late, e.g., through programmatic binding when the variant is needed.

In complex systems—such as those at ASML—some configuration information is only available at runtime, hence, modelling variants' binding time as part of the variability information is relevant.

2.6.2 Dynamic Software Product Lines

In the context of CPS, software is becoming increasingly complex, with extensive variation in both requirements and resource constraints. Since it has become almost impossible to foresee all the software functionality or variability that an SPL requires, there is a need for software that is capable of adapting to the system's context. *Dynamic* SPLs (DSPLs) aim to face these new challenges using runtime variability binding. Runtime binding may occur first during software initialisation, and then during normal operation to adapt to changes in environment [53].

A DSPL can be perceived as a single system—rather than a set of systems—adapting its behaviour using runtime variability binding. To accomplish this, all the valid adaptations should be accessible during the system's operation. A DSPL uses reference architectures—a template for a set of concrete architectures in a particular domain—as the system architecture, and provides explicit and deterministic support for the entire range of adaptation [54]. The set of possible system adaptations is called *adaptability scope* [55]. In practice, hybrid approaches between DSPL and SPL—such as enabling limited runtime reconfigurations during the system's initialization—can be used.

In the context of ASML, where it is desirable to have contiguously running systems, the possibility of performing runtime system's reconfigurations would prove very valuable.

2.6.3 Configuration optimization goals

During product configuration, it is often desired to consider non-functional (i.e. qualitative) parameters associated with the variable items. This need has been addressed by *attributed feature models* [56], where "price tags" are attached to features, denoting the feature's impact on some quality attribute of the resulting variant. When *attributed feature models* are used, a multi-objective optimisation problem may arise. This occurs when the user is interested in maximising or minimising functions over multiple product quality attributes, e.g. energy consumption, cost or response time, to name a few.

The Variability Modeling (VM) [40] and Clafer [37] languages are examples of variability languages that support single- and multi-objective optimisation goals. The VM is a textual variability language designed for the video domain—its name also stands for *Video Modeling*. It addressed unique requirements like video sequence variants synthetization. For example, the video sampling configurations may use certain numerical values as feature attributes; then objective functions (minimise or maximise functions) can be specified, forcing the selection of some values and restricting the configuration space. Similarly, Clafer supports minimise and maximise functions over numerical feature attributes, which assists the user during product derivation by reducing the configuration options when one or more optimisation goals are specified.

When dealing with large variability models from which gazillion configurations can be derived, modelling qualitative aspect associated to variants can help to reduce the configuration space.

Moreover, in the context of ASML, where numerical attributes such as *time* are not qualitative but functional, these optimization capabilities could be used to derive configurations that reduce the system's response time.

2.7 Conclusions

This chapter expanded the systematic literature analysis reported in [6] and updated in [7] by adding the HLVL and incorporating tool support as an evaluation criterion. A review of the references for each language specification rendered some inaccuracies in the evaluation of FDL, Tree-grammars, SXFM, Clafer, VELVET and PyFML. The capabilities of the considered languages, as well as corrections to the above–mentioned proposals, are presented in Table 2.1.

All the evaluated languages support the basic forms of variation already presented in FODA [15], namely, *optional*, *alternative*, and *multiple* selection. Most languages incorporate *cardinalities* and *attributes*, which were not part of feature modelling in its original proposal. The majority of specifications provide *predefined* data types (e.g. integer, string), mainly used as attributes attached to variable items. This possibility of specifying numeric attributes to variable items opens interesting possibilities: Clafer and VM incorporate multi-objective optimization capabilities to their toolsets whilst IVML and VSL use this capability to indicate binding time of the variable elements—as explored in Section 2.6. Constraint expressions are enabled by propositional logic in almost all languages; most of them support relational and arithmetic logic as well. Configuration is mainly supported through value assignment; more than a half of the proposals support *partial* and *complete* configurations while only five support *default* values. Composition mechanisms—a crucial element to build scalable models—is present in under half of the VMLs only. A few proposals include formal language specification and full tool support. The lack of formal language specification causes ambiguities in interpretation, which is evidenced by the amount of "unclear" (?) evaluation results in Table 2.1.

The most relevant inaccuracy found in the evaluation presented by ter Beek et al. in [7] was that Tree-grammars and SXFM do not provide any configuration capability. ter Beek's et. al. assessment turned out to be false, since both of the proposals provide configuration mechanisms indirectly; otherwise resolving variability in a model using these languages would not be possible.

The toolset support evaluation criterion added to previous works in this chapter is important for selection of the variability language to be used at ASML. This is an important requirement because evaluating the language in a real scenario uncovers limitations and strengths that would otherwise remain hidden. In our evaluation, we found that even though ten out of fourteen languages claimed to provide tools, only four actually proved full support. Moreover, it was observed that toolsets of different languages stopped being supported sometime after the publishing of their work. Our evaluation shows that regardless of what is claimed in the publications, testing the toolsets before planning their use is always necessary.

Evaluating the candidate VMLs at the onset showed that their development did not go beyond the first publication; with tool support being either limited or nonexistent. Two notable exceptions are IVML and Clafer. In both cases, the language and its associated toolset have been developed over a span of several years, with more capabilities added gradually. IVML is focused on providing advanced composition mechanisms for the easy integration and derivation of multi-product lines. Besides IVML, VSL and VELVET provide advanced composition mechanisms; however, only IVML includes a framework that implements such capabilities. Clafer is a lightweight language capable of integrating structural and behavioural models along with variability. The possibility of modelling all these different aspects in an integrated way enables its application in multiple domains. For instance, Clafer has been used to model variability of cryptographic components [57] and complex software architectures with variability in the automotive domain [58].

IVML and Clafer support most of the capabilities evaluated in the presented survey and are thus, considered two promising languages in the context of ASML. IVML's composition mechanisms allow the splitting of large variability models—such as ASML's—into different models, which permits a more modular solution where components can be easily aggregated and analyzed as a whole. Besides variability modelling, Clafer incorporates several modelling approaches into a single language; this opens the possibility of specifying hybrid models—which model different aspects of a system like physical processes, system's composition and behaviour—which is relevant for cyber-physical systems such as lithography machines at ASML.

Literature review enhanced with tool support assessment as one of the proposed aspects of VML evaluation yielded Clafer and IVML as the two most salient options based on capabilities considered relevant for ASML. The comparison of capabilities presented here is based on the systematic literature analysis carried out by Eichelberger et al. in [6] and a practical tool support assessment. The value of comparing VMLs capabilities, as is the purpose of this chapter, lays in enabling a more comprehensive and unbiased evaluation of the vast universe of options available. The approach thus far presented in this chapter proves valuable in the evaluation of variability modelling languages because we could reduce the options from fourteen to two (RQ1). To ensure completeness in our evaluation, we must continue applying the theoretical and practical language expressiveness evaluation. In the next chapter, we will focus on a theoretical assessment of VML expressiveness using ontological theories.

Chapter 3

Variability language expressiveness evaluation

This chapter presents an expressiveness evaluation of VMLs using a theoretical framework. The expressiveness of a language can be informally understood as "the breadth of ideas that can be represented and communicated in that language" [59]. An expressiveness evaluation measures the range of ideas expressible in a language. It follows from the definition that to evaluate the expressiveness of a variability language, we must first define the different forms of variability that exist or are perceived to exist in the real-world systems. Then, evaluate the capability of such language to express those forms of variability. Most of the works that presented variability languages were found to do so using toy examples that highlighted their advantages; however, a deeper dive rendered hidden shortcomings that the authors failed to address. Asadi's et. al theoretical framework for variability was used to provide a more objective evaluation and assess expressiveness in the broadest sense possible.

This framework, strongly rooted in Wand and Weber's ontological expressiveness framework, nonetheless postulate a novel idea: to consider the dynamic aspect of variability when evaluating variability language expressiveness. The lack of clarity in Asadi's et al. definitions and examples make their presented ideas hard to grasp. The ontological concepts used do not detail important implications explained in other more foundational works, and as a consequence of this, several misinterpretations were found in Asadi's et al. work and other papers referencing it [60].

This work clarifies important ontological implications in Asadi's et al. framework by incorporating concepts from foundational ontologies [61–63] to improve understanding of the dynamic aspect of variability as proposed by Asadi et al in [64]. Moreover, a different and more detailed mapping (between feature modelling and the ontological concepts) is proposed so that more insights are gained from the evaluation. Finally, since the importance of dynamic aspects in variability modelling is not evident at first sight, rationales for their relevance in the context of ASML are provided.

This chapter is organized as follows:

- Section 3.1 provides a brief introduction to ontologies and their applications in Computer Science. A justification for the selection of Asadi's et al. framework is also provided.
- Section 3.2 introduces the ontological expressiveness evaluation framework in a broad sense, as it is used to evaluate expressiveness of different conceptual languages.
- Section 3.3 presents Asadi's et al. framework in detail. The ontological concepts used in the framework are expanded using foundational ontological evaluation frameworks for conceptual modelling.
- Section 3.4 provides a detailed mapping of feature modelling using the Asadi's framework, and the results compared with those presented in [64]
- Section 3.6 presents the chapter conclusions.

3.1 Ontological considerations in variability modelling

Ontology is "the branch of philosophy that seeks to articulate models of the real world in the broadest sense possible" [65]. Ontological studies have been done over a long period of time and developed in western philosophy at least since Aristotle [65]. Ontology's aim for generality distinguishes it from several specific scientific disciplines (e.g., physics, biology, chemistry), which focus on conceptualizing phenomena in their respective domain. However, many ontological principles—like the selection of concepts and hypothesis, or the axiomatic reconstruction of scientific theories—are applicable in scientific research [66]. An ontological theory provides a set of ontological concepts from which a systematic approach to understand the structure and behaviour of real-world phenomena can be developed [62]. Hence, an ontology can be used as a reference model to investigate and conceptualize different aspects of the real-world systems.

Since the late sixties, ontological theories have been used in some fields such as artificial intelligence and data modelling. In computer science applications, ontologies are usually limited to a specific domain, where the term *domain ontologies* is adopted [66]. Furthermore, these domain ontologies are usually understood as dictionaries or taxonomies with no philosophical stand about the reality they intend to represent [63].

On the other hand, *foundational ontologies* are domain-independent and formally defined models of categories that are philosophically sound [66]. These ontologies can be used to build conceptualizations about specific science and engineering domains [67]. Two outstanding foundational ontologies can be found in computer science: the Bunge-Weber-Wand (BWW) ontology [61] and the Unified Foundational Ontology (UFO) [66]; these works reflect the need to go back to develop ontological theories in their original sense [63]. There is one outstanding

difference between these two ontologies: the BWW ontology is an adaptation and extension of the previously developed Bunge's ontology while UFO was developed from scratch [66].

According to Guizzardi et. al, the formal language type (i.e. domain-independent or a domain-specific) is the most relevant aspect to be considered in the selection of an ontology [68]. Domain-independent languages should be evaluated using foundational ontologies whilst domain-specific languages are compared using domain ontologies. Most VMLs are domain-independent languages; however, they incorporate specific constructs for variability modelling. For that reason, the ontology selected should be domain-independent but provide concepts that can be used to represent variability.

It can be argued that some VMLs—most of them created as part of industrial projects—are domain-specific languages. However, these domain-specific languages are, by design, expressive enough for their particular needs. Their expressiveness is implicitly evaluated by their ability to represent the forms of variability in their domain. Our ontological evaluation aims to assess language expressiveness in a more general sense. Hence, a domain-independent ontology is used as the base for our analysis.

A thorough search of the relevant literature yielded two evaluation frameworks based on a domain-independent ontology (in both cases, the BWW is used) that also include variability concepts: Reinhartz-Berger et al. [69] and Asadi et al. [64]. Reinhartz-Berger et al. focus on system's behaviour variability patterns whilst the framework presented by Asadi et al. include both dynamic (behavioural) and static (structural) variability aspects. Since we aim to evaluate expressiveness in the broadest sense possible, Asadi's et al. framework was selected for this work.

3.2 Ontological expressiveness evaluation framework

An ontological expressiveness theory is a framework to analyze the expressiveness of conceptual modeling languages. The expressiveness framework built by Wand and Weber is based on their *representation model* and the concepts in their ontology [62]. Such an expressiveness evaluation framework has been used as a formal reference model to evaluate conceptual models and their associated languages [60, 63, 70, 71].

The formal models proposed by Wand and Weber are based on the fundamental premise stating that: "*A formal system has the necessary and sufficient properties to represent real-world meaning*"[1] [61]. The formal *representation model* focuses on the relationships between the set of ontological constructs and the modelling language constructs used to describe real-world phenomena [61]. This formal model is based upon the premise that: "*An information system is an artefactual representation of a real-world system as perceived by someone, built to perform information processing functions*" [61]. In general, any artefactual description of real-world phenomena is termed as a conceptual model: "*Conceptual modelling is the activity of formally describing some aspects of the physical and social world around us for*

[1] We change the term "physical-symbol system" to "formal system" in the presented definition

purposes of understanding and communication" [72]. This representation model's premise deemed Information Systems as conceptual models that captures specific aspects of real-world systems. Therefore, Wand and Weber's formal model can then be used to evaluate conceptual modelling languages.

The ontological expressiveness of conceptual modeling languages can be evaluated using a bi-directional mapping between the language's constructs and a set of ontological constructs: *representation mapping* specifies if and how an ontological construct is represented by language constructs, and *interpretation mapping* specifies if and how a language construct represents a real-world (i.e. ontological) construct [62]. After the mapping is determined, assessing the expressiveness of the examined language consists of assessing the presence or absence of any of the four observable defects in conceptual modeling languages [62]: *construct deficit*, *construct excess*, *construct redundancy* and *construct overload* (see Fig.3.1). If the language exhibits a *construction deficit*, then the language is *ontologically incomplete*. The *ontological clarity* of the language is weakened when *construction excess*, *construction redundancy*, or *construction overload* defects are observed [62].

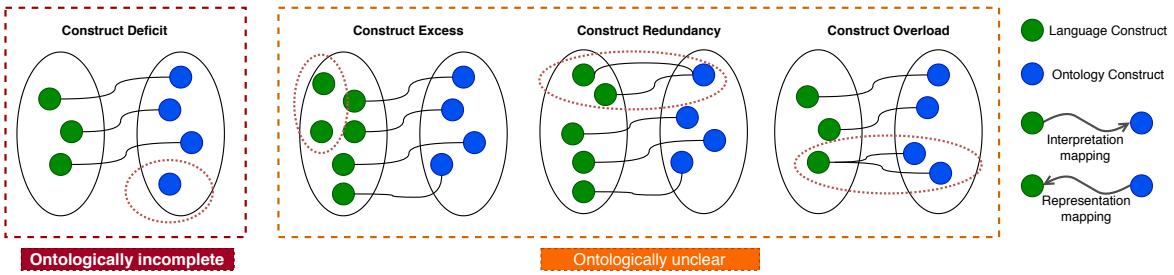

Figure 3.1: *Defects in conceptual modelling proposed in [62]*

The mapping should be *internally consistent*. This means that it must preserve and retain the relationships among ontological concepts and those among conceptual modelling language concepts [63]. For instance, in the BWW ontology things *possess* properties, this "possession relation" must be maintained in the conceptual conceptual model representing both things and properties.

3.3 Asadi's et al. theoretical framework for variability

Asadi's et al. theoretical framework for variability (ATFV) provides a reference model to evaluate variability modelling languages based on ontological considerations. This framework is based on two variability categories: *variability sources* and *variability patterns*. A *variability source* is an element in which variability occurs. A subset of the ontological concepts in the BWW ontology are used to identify variability sources. A *variability pattern* represents a recurring type of variability between two sets of phenomena [64]; here a phenomenon represents a variability source identified in the framework. Based on the previously described ontological expressiveness evaluation framework, the ATFV uses two criteria: *variability completeness* and *variability clarity*. Completeness criterion is used to investigate whether

a particular language has constructs to represent all the variability concepts—i.e. sources and patterns. Clarity means that there is a *one-to-one mapping* among modelling language constructs and variability patterns considered in the framework [64].

First the variability patterns in ATFV are introduced. Then, the ontological concepts used to represent the variability sources are presented in detail. For the ontological analysis, we provide further details from the BWW ontology [61], Wand and Weber's ontological expressiveness framework [62] and their formal models [61]—information not explicitly provided in Asadi's et al. paper [64]. Expand the information about the BWW ontology was deemed necessary to have better understanding of the ontological implications when language constructs are mapped to ontological concepts.

3.3.1 Variability patterns in ATVF

In the context of this evaluation framework, the term *phenomena* refers to "any possible observation that can be made about the domain or part of it" [64]. Here, a domain is considered with respect to a particular software product, i.e. a different domain is assigned to each product in a software product line. Thus, "phenomena in a domain" refers to "observable aspects in a product". These domains must exhibit similarity to some extent, otherwise modelling them as a software product family would not make sense. The following definitions are taken from [64].

Considering two sets of phenomena $S = \{s_1, s_2, ..., s_n\}$ belonging to domain D_1 and $T = \{t_1, t_2, ..., t_m\}$ belonging to a domain D_2, the following similarity scenarios between these two sets are identified.

Definition 1 (*Equivalent* Sets of Phenomena): S is *equivalent* to T ($S \equiv T$), if an only if there is a mapping between elements in S and elements in T.

Definition 2 (*Similar* Sets of Phenomena): S is *similar* to T with respect to p ($S \cong_p T$), if an only if there is a subset of S (i.e. $S' \subset S$) and of T (i.e. $T' \subset T$) which are equivalent, i.e.t $S' \equiv T'$. p is the equivalent subset (i.e. $p \equiv S' \equiv T'$).

Definition 3 (Completely *Dissimilar* Sets of Phenomena): S is *completely dissimilar* to T ($S \neq T$), if an only if there are no subsets of S and of T which are equivalent.

Based on Definitions 1-3, four similarity patterns are defined:

1. *Full similarity double side* - when a set of phenomena S and set of phenomena T are *equivalent* (i.e., $S \equiv T$).
2. *Full similarity one side* - when a set of phenomena S and set of phenomena T are *similar* (i.e., $S \cong_p T$) and either $S' \subset S$ and $S' \equiv T$ or $T' \subset T$ and $S \equiv T'$.
3. *Partial similarity* - when a set of phenomena S and set of phenomena T are *similar* (i.e., $S \cong_p T$) and there is no subset in one domain that is equivalent to the entire set

in the other domain.

4. *Complete dissimilarity* - when two sets of phenomena are *completely dissimilar*.

With the exception of *full similarity double side*, the other three similarity patterns represent the *variability patterns* considered in ATFV.

3.3.2 Bunge-Wand-Weber ontological concepts

The Bunge-Wand-Weber ontology is based on a subset of Bunge's ontology, adapted and extended by Yair Wand and Ron Weber to address Information Systems (IS) representation issues [61]. Bunge's ontology has been widely used in science because it is "well formalized as an axiomatic system, using a set theory representation" [63]. A set representation is necessary for our analysis since the variability patterns described previously are defined over two sets. Additionally, "Bunge models the world as a world of systems" [62], a fundamental view that provides relevant concepts to different computer science domains. It has been shown that Bunge's ontology, as adapted for IS, provides a valuable reference for the evaluation of modelling languages and methods [73–75].

We now introduce the ontological constructs as presented in [62]. A subset of those concepts, used in Asadi's et al. paper [64], is shown in Table 3.1. These concepts are sufficient to explain the *structural* variability sources considered in the ATFV but does not cover all the fundamentals aspects required to fully understand the process variability sources, including the *ordering variability pattern*. The remaining of the concepts dealing *stable* and *unstable* states as well as *internal* and *external* events will be introduced later. Additionally, the state-tracking formal model [62], as proposed by Wand and Weber, will be also introduced to support the analysis of the dynamic aspects of variability.

A more detailed explanation of the ontological concepts (Table 3.1) and their relationships, as presented by Wand and Weber in [63], is provided in the remainder of this section.

In Bunge's ontology [65], the world is made up of *things* that exist physically in it [63]. A *thing* is the basic building block in the ontology; all the other concepts are built upon it. *Things* possesses *properties* and are know through them, i.e. to distinguish *one thing* from *other thing* we need to look at their properties. *Properties in general* are those owned by things, e.g. "colour", "shape", "country of origin". An *individual property* is a particular value of the property in general, such as "color:orange", "shape:circular" or "country of origin:the Netherlands". An *intrinsic* property is that of a single thing, such as "colour", whereas a *mutual* property exists between two or more things, e.g. "manufactured by". When we say "*this pen is manufactured by machine A*", the "manufactured by" property only exist between the *pen (thing)* and *machine A (thing)*.

A *thing* can be uniquely identified in the world through its individual properties because no two things can share *all* of them. When representing existing properties of a *thing* in a conceptual model, it is not possible to represent *all* properties; instead, *some* relevant prop-

Ontological Construct	Explanation
Thing	A thing is the elementary unit in this ontological model. The real world is made up of things. A composite thing may be made up of other things (composite or primitive).
Properties	Things possess properties, A property in *general* is modelled via an *attribute* function that maps the thing into some value, i.e. an *individual property*. A property of a composite thing that belongs to a component thing is called a *hereditary property*. Otherwise, it is called an *emergent property*. A intrinsic property is inherent to an individual thing. A property that is meaningful only in the context of two or more things is called a mutual or relational property.
State	The vector of values for all attribute functions of a thing is the state of the thing.
Conceivable state space	The set of all states that the thing might ever assume is the conceivable state space of the thing
State law	A state law restricts the values of the properties of a thing to a subset that is deemed lawful because of natural laws or human laws.
Lawful state space	The lawful state space is the set of states of a thing that comply with the state laws of the thing. The lawful state space is usually a proper subset of the conceivable state space.
Event	An event is a change of state of a thing it is effected via a transformation (see below)
Event space	The event space of a thing is the set of all possible events that can occur in the thing
Transformation	A transformation is a mapping from a domain comprising states to a co-domain comprising states.
Lawful transformation	A lawful transformation defines which events in a thing are lawful.
Lawful event space	The lawful event space is the set of all events in a thing that are lawful
History	The chronologically ordered states that a thing traverses are the history of the thing.

Table 3.1: *BWW ontology concepts part I, as presented in [62]*

erties are modelled using *attributes*. Attributes are a human-created abstraction mechanism to represent a set of identifiable properties existing in the real-world [63]; e.g. *colour* is not a property of an object that exist in the real world, there are several chemical properties of the material that make it, let us say *orange*, the tag '*orange*' is a human-created abstraction that simplifies the task of identify an object. Additionally, attributes like "price" have no meaning in the natural world, only exist within our human societies, and for that reason we represent them in our conceptual models.

Things can be combined to form a *composite thing*. Composite things can be decomposed in their parts, that are in turn either composite things or *simple things*—if they cannot be decomposed any further. As matter in the universe, things cannot be created or destroyed, but only change their properties, combined to form composites, or separated into their components [63].

A *property in general* is represented by a *state function*; state function values are *individual*

properties. A set of state functions form a *model* or *functional schema*. A model can represent similar things, while a thing may be described using different models. A *state* of a thing is the set of values for all the state functions representing such thing. The set of all possible states obtained from a model is the *conceivable state space*.

An *event* is a *change* in the state of a thing and it is effected via a *transformation*. *Change* can be either *quantitative* or *qualitative*. When change occurs in one or more individual properties is *quantitative*, if general properties are acquired or lost the change is *qualitative*. Qualitative change usually correlates to change on the *behaviour* of a thing [63]. The set of all the possible events that can occur in a thing is called (conceivable) *event space*.

A *state law* is any restriction on the values of the state functions of a thing. The *lawful state-space* is the set of states that comply with the state laws of a thing and it is a proper subset of the conceivable state-space. Similarly, a *lawful transformation* define which events in a thing are lawful. The set of all the lawful events comprises the *lawful event-space*. The set of laws that a thing adheres to determines its *behaviour*.

The set of states or events traversed by a thing in a chronological order represents its *history*. Either states or events can be used to specify the history of a thing because an ordered set of states implicitly define a set of events and vice versa.

Importantly, the BWW ontology claims to provide *necessary* concepts to describe real-world systems; however, whether or not these concepts are *sufficient* is an open research question [61].

3.3.3 Structure and process of a domain in ATFV

Asadi's et al. framework postulates that variability among products domains might occur in both static (i.e. its structure) and dynamics (i.e. its processes). The *structure of a domain* is composed by *things*, *properties* and *lawful state space* whereas its *lawful event space* and the *ordering* between these events describes the *process of a domain*. The previously described variability patterns (*full-similarity one side*, *partial similarity* and *complete dissimilarity*) are common for both the structure and the process of domains. Additionally, the *ordering variability pattern* is found in processes. This pattern captures the fact that two systems might allow the same state transitions but the sequence of those transitions for a particular run (i.e. its history) might differ. The structural variability is represented using the three common variability patterns over two sets of things, properties or (lawful) states. Similarly, process variability is obtained comparing two sets of lawful events using the common variability patterns and the variability ordering pattern. The ordering pattern only indicates if two sets of events happen in the same order or not.

The *ordering variability pattern* is not formally described by Asadi et al. in [64]. First, they present an example where three scientific conferences (**A**, **B** and **C**) are described. Each conference's description includes a set of *things* (e.g. papers, demos), *properties* (e.g. Paper's title), *lawful state space* (e.g. Paper-status) and *lawful event space*. The *lawful*

event space is specified as transitions between paper's states. For instance, the following set of events are defined for Conference **A**: $Paper\{(Submitted \rightarrow Accepted), (Submitted \rightarrow Rejected), (Submitted \rightarrow Short\_Paper\_Accepted)\}$, each event has an explicit initial state (e.g. *Submitted*) and a final state (e.g. *Accepted*). The ordering pattern between events is then explained as follows: "*assume that determining the program of conference A involves an order* $Workshop \rightarrow Tutorial \rightarrow Main\_Conference$ *and for conference* **B** $Main\_Conference \rightarrow Workshop \rightarrow Tutoral$. *Although both the conferences contain the same set of events, the ordering of events is different*".

We consider the *ordering pattern* explanation in Asadi's et al. paper rather confusing. In their example, they defined the lawful event space as transitions between paper's states, e.g. $(Submitted \rightarrow Accepted)$, where each event has an initial state (e.g. *Submitted*) and a final state (e.g. *Accepted*). Then, three different conference's "*events*" are used as an example to explain the ordering concept: *Workshop*, *Tutorial* and *Main_Conference*. Apparently, these elements are now considered events, not defined as a transition between an initial and a final state but rather as a label representing a set of activities (e.g. Workshop). It is not clear the relation of this example with what is usually represented in variability models.

Instead, we use state machines to explain the notion of ordering. Using this representation states retain its ontological meaning; each state represent a *unique* set of properties obtained from the set of state functions (model). Similarly, the transitions between states are related to *events* in the ontological sense; they represent change in properties since initial and final states are always different. Consider the two state machines depicted in Fig. 3.2 representing System A (left) and System B (right). Both systems have the same lawful state and event spaces—only lawful states and events are modelled. A finite sequence of transitions in a state machine is called a *trace*. Similar to the concept of *history* of a thing, traces represent chronologically ordered transitions between states, therefore, we related the ontological concept of *history* to a *trace* in the state machine representation.

Even though the set of states and events are the same for both systems (listed in the right side of Fig. 3.2) , their initial state is different and therefore, the sequence of state transitions differ as well.

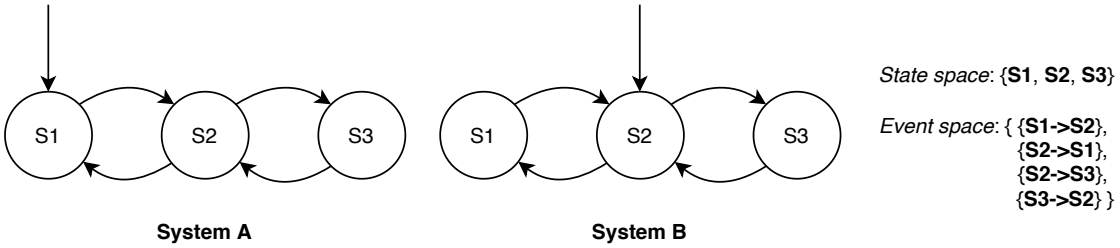

Figure 3.2: *State machines for two different systems*

Consider the trace for System A: $t_A = \{\{S1 \rightarrow S2\}, \{S2 \rightarrow S1\}\}$, and the trace for System B: $t_B = \{\{S2 \rightarrow S1\}, \{S1 \rightarrow S2\}\}$. We define the ordering pattern as the *order* function, such that $b = order(t_x, t_y)$, where t_x and t_y are two traces and b is a boolean value. The *order* function yields *true* if and only if t_x and t_y contain the *same set of events* in the *same*

order, otherwise *false*. Using the provided example, $order(t_A, t_B)$ yields *false* because the two traces contain the same events but in a different order.

We have now introduced the ontological concepts representing variability sources and the different variability patterns used in ATFV. The presentation of ontological concepts presented by Asadi's et al. in [64] was expanded using the BWW ontology. Additionally, a more formal definition of the *ordering pattern* than the one presented in [64] was provided. In the next section we apply Asadi's et. al framework to provide an alternative representation mapping of feature modelling to that presented by them.

3.4 Representation mapping of feature modelling using ATFV

Asadi's et al. representation mapping between variability sources and feature model constructs proved to be oversimplified and in need of clarification. An understanding of all the implications of their proposal calls for detailed explanations which are missing in their original work, and are necessary in order to avoid ontological inconsistencies in the mapping of concepts. For this reason, we will first build a detailed mapping expanding the explanation of key concepts when necessary, providing the rationale behind all decisions and using simple feature models as examples to support our arguments. Though lengthy, this level of detail is necessary in order to ensure accuracy and improve clarity.

The remaining of the section is organized as follows:

- Subsection 3.4.1: Maps the structural variability sources to feature modelling constructs.
- Subsection 3.4.2: Details the concepts comprising process variability by introducing the *state-tracking model* and determines whether or not feature modelling is able to represent those concepts.
- Subsection 3.4.3: Presents Asadi's et al. mapping of variability sources (for both structural and process variability).
- Subsection 3.4.4: Maps the general variability patterns to feature modelling constructs.
- Subsection 3.4.5: Summarizes the results obtained in this mapping.

3.4.1 Mapping ontological constructs representing structural variability sources

As discussed in previous chapters, feature modelling (FM) is the most widely used approach and it is supported by almost any variability management tool. A representation mapping of feature modelling concepts, as presented in its original proposal [15], is addressed in this section.

The running example

The basic representation unit in FM is the *feature*. A feature is defined as "user-visible aspects or characteristics of the domain" [15]. *Feature groups* relate a set of features to form a tree-like structure called *feature model*. A single feature may be *mandatory* or *optional*; features can be grouped in *OR* groups or *alternative* groups. Cross-tree constrains are defined between any pair of features in the feature model (e.g. *requires* and *excludes* relations). A simple feature model of a IoT computer using a visual notation is shown in Fig. 3.3; it incorporates all the elements previously described.

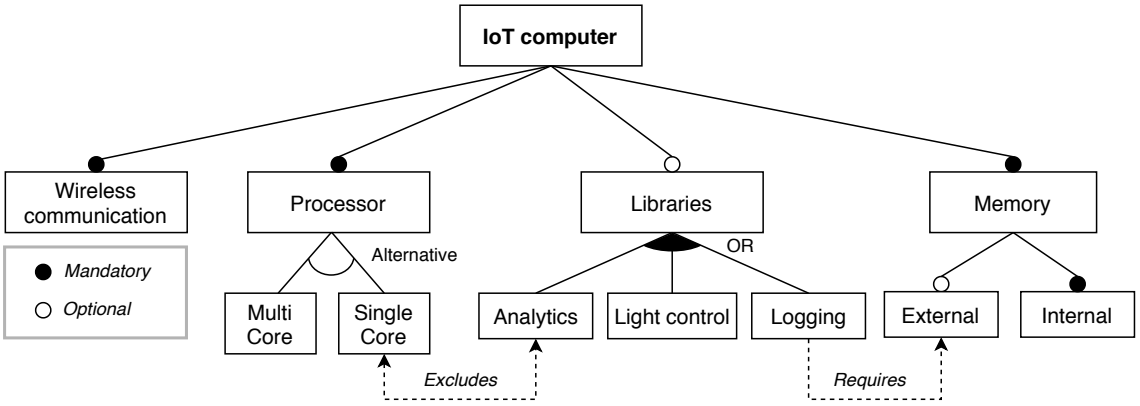

Figure 3.3: *Example Feature Model from an IoT computer*

Starting by considering an IoT computer (root node in the feature model), it represents a physical item in the real-world, therefore, it is related to the ontological concept of a *thing*. The mandatory feature "*Wireless communication*" refers to the capability of the IoT computer to communicate wirelessly with other devices. *Wireless communication* can be deemed a property of the IoT computer (thing). The *Wireless communication* feature can be mapped to the concept of *thing* as well, if it is interpreted as a wireless communication (hardware) module.

The *Processor* is a *property in general* while *Single-core* and *Multi-core* are *individual properties* related to it. As in the case of the *Wireless communication* feature, if we interpret the *Processor* feature as the actual integrated circuit, it can be seen as a physical *thing*.

The duality described above constitutes an issue with the ontological interpretation of feature models; namely, the concept of feature is *overloaded*. A feature can be seen as a *thing* or a *property*. Whenever we face this ambiguity, we interpret features as properties, more precisely, *attributes*—since it is a human-created abstraction. In the original proposal, features are defined as "user-visible aspects or characteristics of the domain" [15]. Sometimes it is convenient to represent a system's composition to describe two different domains. To continue with the running example: the IoT computer has a processor, both the IoT computer and the processor are physical objects (i.e. things). However, in the context of feature modelling, when we say that an "IoT computer has a processor" we are not interested in the composition relation between these two physical objects. We use this description as an

attribute function or a *property in general* describing an user-visible aspect of the system being modelled. In the example, two domains can be distinguished by the processor type: *single-code* or *multi-core*, which represent attribute values or individual properties.

The IoT computer includes three optional *Libraries*: *Analytics*, *Light control* and *Logging*. In this case, *Libraries* is a property in general, while each specific instance is an individual property. Similarly, *memory* represents a memory type and it might be internal memory (mandatory) or external memory (optional). Even though memory represents a physical object, it is interpreted as a property that describes the domain of IoT computers for the same reasons discussed above.

This mapping is *internally consistent* with the BWW ontology. In the ontology, things *have* properties while in the feature model *feature-things* also relate to *feature-properties*. The possession relationship is expressed through the hierarchical structure in the feature tree; an IoT computer (root node) *possesses* all the properties expressed in the lower levels on the tree. The same hierarchical structure is used to relate a *property in general* to an *individual property*. A property in general (feature group) comprises a set of individual properties (features included in the feature group); the hierarchical structure in feature trees describes this relationship consistently.

Now let us consider the information derived from a feature model. After all the required decisions in a feature model have been made, we are left with a model instance—also called *product configuration* in the context of SPL. For example: we must select a feature from an *alternative* group—this is a required decision. We relate the ontological concept of a *state* with a *configuration* derived from a feature model. A *state* is ontologically defined as an set of individual properties of a thing. A configuration is also a set of individual properties of a thing—represented by the root node in the feature model.

The feature model's constraints reduce the number of possible configurations (i.e. states). For instance in Fig. 3.3, the constraint "**Libraries:Logging** *requires* **Memory:External**" enforces the inclusion of both features in a configuration in order to be considered valid. Likewise, the constraint "**Processor:Single Core** *excludes* **Library:Analytics**" invalidates all the configurations that include both features. These constraints reduce the set of all possible configurations to a subset of *valid* configurations. Thus, in feature models, constraints relate to the *state law* ontological concept. The *conceivable state space* corresponds to all the possible configurations while the *lawful state space* is represented by the set of *valid configurations*—i.e. configurations satisfying all the constraints in the model.

The domain of the feature model, represented by the root node name, was related to the ontological concept of *thing*. A feature group represents a *property in general* while its elements represent *individual properties*. A product configuration is mapped to the ontological concept of *state* while a constraint represents a *state law*. All the possible configurations derived from a feature model conform the *conceivable state space* while all the valid configurations comprise the *lawful state space*. It is thus confirmed that feature modelling is expressive enough to represent *structural variability sources* as proposed in Asadi's et al. framework.

3.4.2 Mapping ontological constructs representing process variability sources

Before analyzing *process variability sources*, we need to explain the formal *state-tracking model* proposed by Wand and Weber [61]. The analysis presented thus far is based on a *representation* model that address the problem of statically representing the essential aspects of real-world systems as conceptual models, however, when the concept of *change* comes into play—the essence in process variability–this model results insufficient. The *state-tracking* model is based on the premise that "*an information system is a state-tracking mechanism for the real-world systems it is intended to model*"[61]. The *state-tracking* model must meet the following four requirements in order to faithfully represent a real-world system: *mapping requirement*, *tracking requirement*, *reporting requirement* and *sequencing requirement*.

The *mapping requirement* is defined as follows [61]: "*a one-to-many mapping must exist from the set of real-world system states into the set of information system states*". In the context of SPLE, the real-world system refers to a computer running a software product while the information system is the variability model used to build it. Since we already mapped the ontological concepts of *state* to *product configuration*, it follows that a real-world system state is a software product—one built from a set of software assets and a configuration—running on a computer system. There is a one-to-one mapping between a software product (real-world) and a variability model instance (information system) because every software product in a SPL is built from one and only one configuration. Thus, there is a one-to-one mapping between real-world and model states in our current analysis.

Both one-to-many and one-to-one mappings ensure that at least one information system state exists for every state in the real-world, which is relevant for the remaining requirements. As Wand and Weber explain in [61], the *mapping* requirement was loosened in order to consider those real-world systems in which a single state is represented by multiple states in the information system. The case of a payment system was used as an example: a real-world transaction may be represented by multiple states in the information system due to its implementation details. The system may batch the transactions to improve software efficiency; in that case, one transaction is represented by two states—waiting-transaction and processed-transaction. Thus, a one-to-one mapping (as the one we provided) is a more faithful representation of the real-world compared to one-to-many mapping (as in this example), therefore it is more accurate and still compliant with *mapping* requirement.

The tracking requirement states that: "*When the real-world system changes states, the information system must be able to change from a state that corresponds to the initial real-world system state to a state that corresponds to the subsequent real-world system state*" [61]. In other words, every change of state in the real-world system correlates with a change of state in the information system, and initial and final states must be clearly identified. These two requirements are important to map the ontological concept of *event*.

An *event* is a change in the state of a thing. We have concluded that the state of a thing is represented by a *product configuration* in an SPL, therefore, we can relate the ontological concept of *event* to a *change in a product configuration*. A configuration change in an SPL may be *offline* or *online*. Offline configuration changes happen in the traditional SPL

approach where the application engineer derives a configuration from the variability model; then that configuration information is used to build a software product from a common set of artifacts. The derivation of a product configuration involves a series of transitions from one configuration to another until the final configuration is obtained. However, the software structure does not change after the software product is built from that final configuration. On the other hand, online configuration changes happen at runtime; the software undergoes structural changes while running in the hardware platform.

The variable modules in a software product can be bound at three different moments: *design time*, *build time* and *runtime*. *Design time* means that the variable artefacts (e.g. source code modules) are selected before building the product (e.g. compiling the source code into a binary file). *Build time* binding means that the variable items are select when the product is built, for example, compiler flags can be used to either include or remove code fragments. Finally, modules can be bound at runtime under certain scenarios; a common one is when the information required to resolve variability is available at runtime only. Design and build time binding are offline configuration changes, while runtime binding represents online configuration changes.

The concepts of *binding time* and Dynamic Software Product Lines (DSPLs), mainly explored in academia, address some of the challenges of online configuration changes (see Chapter 2, Section 2.6). Runtime variability is the key element that sets DSPLs apart from the original SPLE approach [53]. In the current analysis, we exclusively consider runtime (i.e. online) configuration changes to represent the ontological concept of *event*. This decision is based on two reasons. First, online configuration complies with the *tracking requirement* in Wand and Weber's *state-tracking* model because changes occur in the real-world system, while offline configuration changes do not represent real-world system events. Second, the *ordering variability pattern* proposed in Asadi's et al. framework is only relevant at runtime; the order in which the offline configuration is changed is not relevant because only the final configuration is used to build the software product.

Given a set of valid configurations (lawful state space) derived from a feature model, would the transition between any pair of valid configurations correctly model a *conceivable event space*? Yes, however, this consideration is of limited application because there is no construct that allows us to reduce such event space to a subset considered lawful (i.e. define a lawful event space). This is important because in the real-world system, not all the transitions between states are possible.

In feature modelling we do not find any construct that represents a runtime transition between an initial configuration (initial *state*) and a final configuration (final *state*)—i.e. the *event* representation we have built. Thus, we conclude that the ontological concept of *event* is not represented in feature modelling. Since all the remaining concepts in process variability (event law, lawful event space and the ordering pattern) are built upon the concept of *event*, those are therefore not represented in feature modelling either.

In order to make this analysis useful for the evaluation of other VMLs (other than feature modelling), we continue our analysis to build a consistent interpretation of these ontological

constructs based on the mapping presented thus far. In order to do so, we now present the two remaining requirements for the *state-tracking* model before diving into the ordering variability pattern.

The *reporting requirement* states that [61]: "*If an external (input) event occurs in the real-world system, an external (input) event that is a faithful representation of the real-world external event must occur in the information system*". This means that runtime configuration changes caused by a change in the system's environment need to be represented in the variability model. Still, there is a missing requirement to ensure that our model faithfully represents the transitions occurring in the real-world system.

The *sequencing requirement* states that [61]: "*The order in which external events occur in the information system must be the same as the order in which external events represented by these information-system external events occur in the real-world system*". This requirement enforces that the variability model keeps track not only of the events in the real-world system, but also the order in which these events occur. Two systems might have the same valid configuration space and the same set of valid transitions within that valid configuration space, but the order in which these transitions occur might be different, giving rise to the *variability ordering pattern* proposed by Asadi et al.

We have concluded that the ontological concept of *event* can be relate to an *online configuration change*. Now we refine this mapping further, and argue that the concept of *internal event* can be related to runtime variability binding.

Runtime variability binding and internal events

The ontological concept of *internal event* is defined in the BWW ontology as: "an event that arises in a thing, subsystem, or system by virtue of transition laws in the thing, subsystem, or system" [61]. As an example, consider the feature model with four generic features (A, B, C, D) in Figure 3.4. This small feature tree has only four valid configurations (C1-C4); each configuration is represented as a set of features, each feature is mapped to a software module with the same name—each module implements the associated feature. A software product can be generated from each configuration using a set of software modules implementing all the functionality defined in the feature model. Due to information being unknown during design or build time, it is the case that some variable items are bound at runtime. During runtime binding, dependencies may affect the order in which these modules are bound.

Fig. 3.5 below shows two different binding sequences. Each module in the sequences has two timing values: ti and tf, representing the time when the binding started and finished, respectively. The binding operation time is a non-zero value calculated as $tf - ti$. The systems that allow runtime binding oftentimes have some dependencies between modules. A simple dependency such as **B** *after* **A**, means that module **B** can start the binding operation only after module **A** is bound, i.e. $B_{ti} > A_{tf}$. We call these kind of dependencies between dynamically bounded modules *ordering constraints* because they affect the order in which the variants are bound to the system.

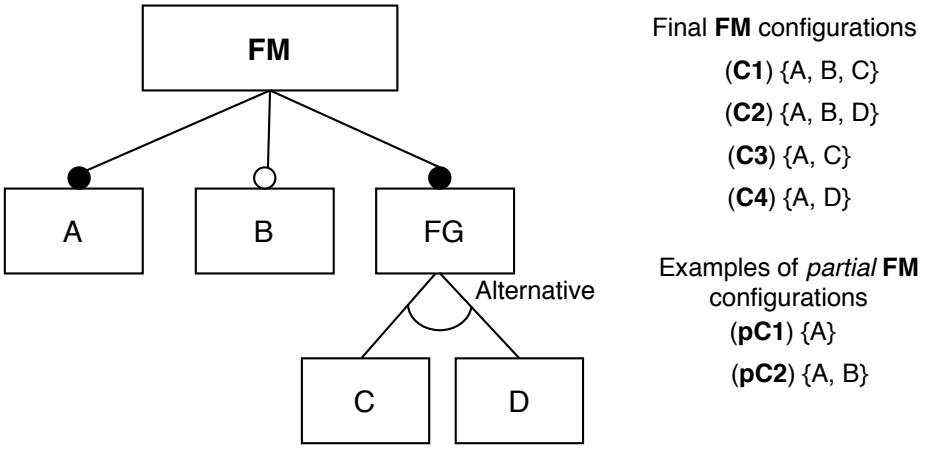

Figure 3.4: *Minimal FM to exemplify variability ordering and binding time*

Fig. 3.5 (left) shows the runtime binding of a software product, built from the FM in Fig. 3.4 and the configuration {A, B, C}. First, module *A* is bound, followed by *B* and then *C*. Each time a module's binding process is done, there is a transition between one partial configuration to another, until the final configuration (in this case {A, B, C}) is reached; this particular order complies with the ordering constraints shown at the top of the diagram. These transitions between configurations are represented at the bottom of Fig. 3.5 (left) using gray arrows; a partial configuration (state) is marked in orange (e.g. pC1), and the final configurations in green (e.g. C1).

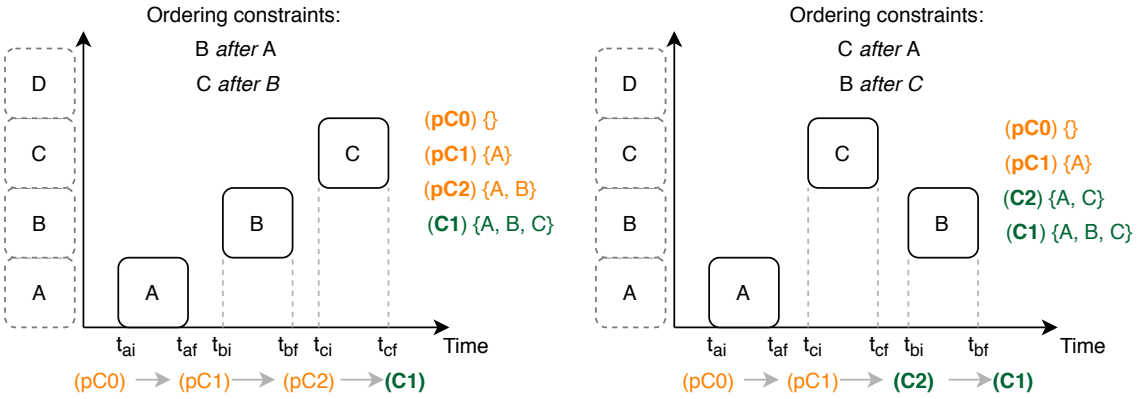

Figure 3.5: *Runtime binding of configuration {A,B,C} with different ordering constraints (dependencies)*

Fig. 3.5 (right) shows the same software product with different ordering constraints, causing a different set of partial configurations and therefore, a different set of transitions. Modules *A*, *C* and *B* are bound in that particular order.

This small example uncovers and highlights differences between two systems that seem identical if we only look at the feature model configurations. This difference between two systems is identified by the *order variability pattern*, but it is not possible to represent it using feature

modelling.

In order to define events more accurately, we need to introduce the concepts of *unstable* and *stable* states. The BWW ontology provides the following definitions: "an *unstable* state is a state that as a consequence of the system's transition laws, will be transformed into another *unstable* state or a *stable* state. A *stable* state is a state that will remain the same, unless an external action coming from the system's environment forces it to change" [61]. We relate *partial* and *final* configurations in SPLs to the ontological concepts of *unstable* and *stable* states, respectively. Let us clarify this proposed mapping using the following example: in the modules' binding sequences shown in Fig. 3.5, both systems start with the same unstable state (pC0, i.e. no modules present) and end in the same stable state (C1, {A,B,C}). A system that uses dynamic binding of (variable) software modules always traverses a set of unstable states until the final state is reached. Some VMLs incorporate a language construct to represent binding time information, opening the possibility to differentiate the two binding sequences by attaching a binding time to each module. The second scenario, when there is a transition between two stable states (i.e. two final configurations) due to a change in the system's environment leaves the realm of SPLs to enter that of Dynamic SPLs.

Reconfiguration in DSPL and external events

The key idea behind Dynamic Software Product Lines is to allow the system to transition between final configurations at runtime. A transition between two (final) configurations is called dynamic *reconfiguration* [54] and it is triggered by a context change, i.e. "a change that takes place in the external environment of a system" [76]. Fig. 3.6 shows a dynamic system reconfiguration from **C1** to **C2**. As in the definition of *stable state* discussed earlier, here a change in a final configuration is triggered by a change in the system context or environment.

An *external event* is defined in the BWW ontology as "an event that arises in a thing, subsystem, or system by virtue of the action of some thing in the environment of the thing, subsystem, or system" [61]. We can then relate the concept of dynamic reconfiguration, in the context of DSPL, to an external event in the BWW ontology.

Finally, the sequence of runtime transitions that the system undergoes is related to the ontological concept of *history*, which was previously defined (in Table 3.1) as: "the chronologically ordered states that a thing traverses". We will use this concept to define the ordering variability pattern next.

In summary, thus far we have mapped the following ontological concepts: :

- **Event**: A runtime change in software configuration. These changes are triggered by internal or external events.
- **Unstable state**: Partial configurations (variants still need to be bound to the system).
- **Stable state**: Final configuration (all the mandatory variants are bound to the system).

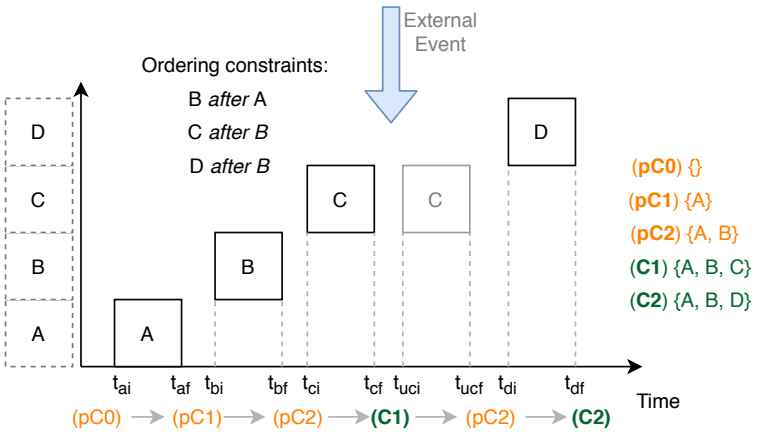

Figure 3.6: *Transition between two final configurations in a Dynamic Software Product Line*

- **Internal event**: Binding of a variable element at runtime. An internal event is a transition from one unstable state to another unstable state or to a stable state.
- **External event**: Change between two final configurations due to a change in the environment of the system.
- **History**: An ordered sequence of runtime changes in a software configuration.

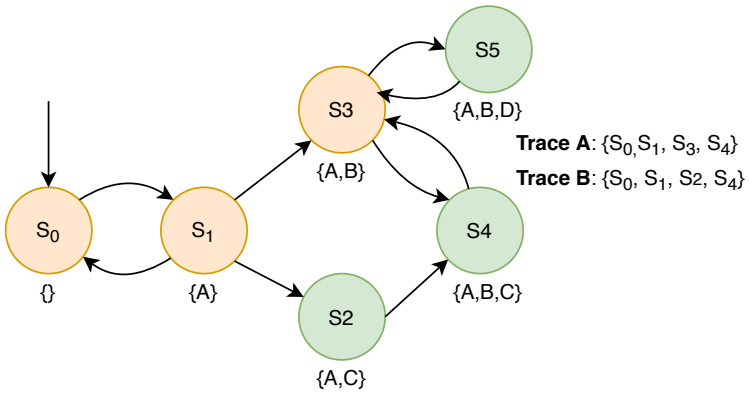

Figure 3.7: *Transition between two final configurations in a Dynamic Software Product Line*

As we proposed earlier, we represent states and events using a state machine. A state machine includes *unstable* and *stable* states; transitions between states are *internal* and *external* events. Fig. 3.7 shows part of a state machine defined for the feature model in Fig.3.4. Each state has configuration information attached to it; unstable states are represented in orange and stable states in green. This diagram represents part of the *lawful event space* of the system.

The *ordering variability pattern* indicates whether or not two different systems with the same

configuration reach that state undergoing the same sequence of transitions (i.e. whether their *history* is the same or not). As in the previous example, two systems can reach the final configuration {A,B,C} by following two different traces, trace A and trace B in Fig. 3.7. In that case, the ordering pattern would yield *false*.

As mentioned earlier, since *events* are not representable in feature modelling, the *ordering variability pattern* cannot be represented either, because order is evaluated over a set of events. However, the completed representation mapping for process variability presented in this section will be useful to evaluate other VMLs that support the same or new constructs beyond those supported by feature modelling.

Now that we used ATFV for to build a representation mapping for feature modelling, let us introduce Asadi's et al. proposal so that we can show the need we identified for a more detail mapping.

3.4.3 Variability sources representation mapping in Asadi's et al. work

Asadi et al. use their proposed framework to evaluate feature modelling in [64]. In their representation mapping of variability sources, they use the ontological concept of "natural kinds" that we have not mentioned thus far. Let us then first introduce the concept of *natural kinds* so that we can discuss the mapping provided by Asadi et al.

Asadi et al. use the concept of *natural kinds*, taken from Bunge's ontology and explained by Wand and Weber as follows: "A set of things adhering to the same *laws* is called a *natural kind*. Since laws relate properties, a natural kind implies a set of properties as well. As laws determine possible states, a natural kind is a set of things that exhibits like behaviour" [63].

The term *natural kind* is borrowed from philosophy. In its original context, the discussion about what is and what is not a *natural kind* may be subject of intense debate [77]. A simplified approach to understanding it is the following: Natural kinds are classification schemes that exist in nature as opposed to other classification schemes that are merely arbitrary. In the natural world, examples of natural kinds would be *insect* and *bird*, but not *butterfly* or *eagle*, or *insects-living-in-the-Amazon* and *eagles-born-in-America*, which are non-natural or "artificial" kinds [77].

This concept was incorporated by Mario Bunge in his ontology to formulate the *principle of nominal invariance*: "Rather than assigning things a new name on every change of a property, Bunge advocates keeping the name of a thing until it changes its natural kind (principle of nominal invariance): A thing, if named, shall keep its name throughout its history as long as the latter does not include changes in natural kind—changes which call for changes in name" [63]. As a closer example applied to our domain, the concept of "computer" can considered a natural kind. Every computer adheres to a set of laws that allows it to *read data, process it, and output a result*. Changing its processor model or increasing its memory capacity do not change its intrinsic laws, therefore, it can still be called a computer.

The proposed mapping of feature modelling to Asadi's framework presented in [64] is as follows: "According to definitions for features, we can conclude a feature is a particular set of properties or processes of one or more products in a product family. To interpret features based on Bunge's ontology, we can relate features to natural kinds because natural kinds are used to define things with a set of common properties that adhere to the same laws including both transition and state laws. Hence, the natural kinds similar to features can be used to represent both the processes (lawful event space and time) and structure (things, properties, and lawful state space) of the domain."

Then, their conclusion is: "considering the ways features and natural kinds can be related, we propose that a 'good' set of features is required to represent all sources of conceptual variability including things, properties, lawful state space and lawful event space" [64].

We found the proposed mapping too general because although mapping features to the concept of *natural kinds* is correct in some scenarios, there are others were a feature represents an individual property. For instance, a feature can represent an optional *backup computer* (*natural kind*) but it can also be memory size (*individual property*). In such instances, Asadi's proposed mapping proves inaccurate.

A set of features can represent things, properties and (when considering constraints among them) a lawful state space. No *explicit* behavioural representation was found in feature models, however. If by relating a feature (or a set of features) to the concept of *natural kinds* it is claimed that behaviour is represented in feature models, this representation is at best *implicit*. With an implicit representation it is not possible to check dynamic aspects the systems, such as the proposed *ordering variability pattern*.

3.4.4 Variability patterns in feature modelling

Now we present the mapping between variability patterns and variability constructs in feature modelling.

Fig. 3.8 is a visual representation of the four similarity patterns presented in 3.3.1. If the *full similarity double side* is omitted, we are left with the three variability patterns considered in Asadi's et al. framework.

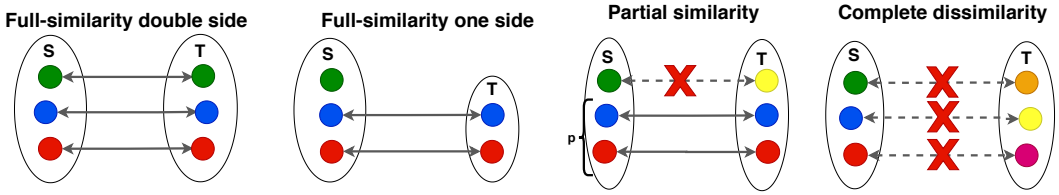

Figure 3.8: *Visual representation of similarity patterns*

Asadi's et al. postulates in [64] that *optional* features, *XOR* (alternative), and *OR* feature groups are sufficient to represent all the variability patterns in the framework. In order to delve deeper into Asadi's et al. proposed mapping, let us clarify their postulates graphically.

Fig. 3.9 represents the feature model constructs to be mapped. Three hypothetical features (**A**, **B** and **C**) are modelled. Each diagram includes example configurations for two different systems (X and Y).

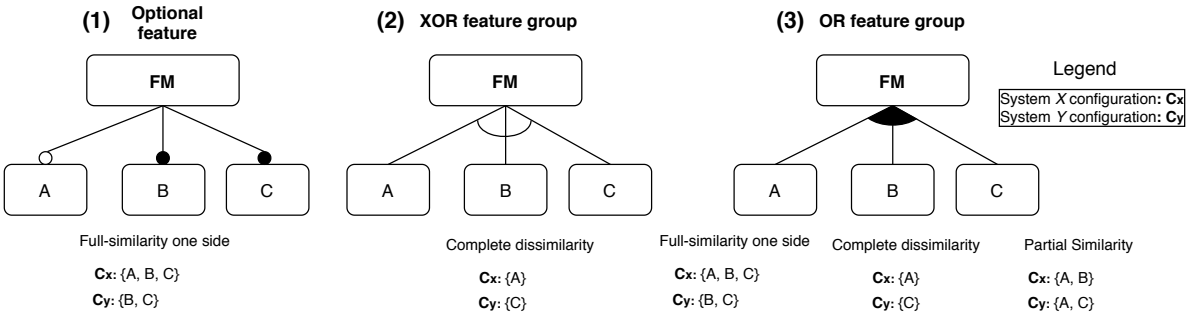

Figure 3.9: *Variability patterns mapped to FM constraints. 1) optional, 2) XOR group, 3) OR group.*

Using the *optional* construct (Fig. 3.9 (1)), configuration C_x includes optional feature **A** while configuration C_y does not. Note that for an optional feature to make sense, we need at least one mandatory feature. If only an optional feature were modelled, one of the systems would be represented with that feature while the other would have an empty set of features. Having at least one mandatory feature, this construct represents the *full-similarity one side* variability pattern.

When a set of features are modelled as an *XOR* group, one and only one feature must be selected from that group. When two different systems are modelled using this construct, it yields the *complete dissimilarity* pattern.

Finally, the *OR* construct (see Fig. 3.9 (3)) allows selection of one or more items from the group, which enables expression of the three different variability patterns. In the case of C_x and Cy, it is possible to select a disjoint subset of features from the group, yielding the *complete dissimilarity* pattern. C_y might be a subset of C_x, expressing the *full-similarity one side* pattern. Finally, if a subset of C_x and a subset of C_y are equivalent sets, we obtain the *partial similarity* pattern.

When the previously described constructs are combined in a bigger (and more realistic) feature tree, not only variability but also similarity is modelled. Combining these constructs enables the description of two different systems by choosing the same set of features in some cases (commonality) while selecting different options in others (variability).

Fig. 3.10 shows the combination of an *optional* feature and an *XOR* group. Two different variability patterns results from using the *XOR* group for modelling variability and the optional feature for commonality. When feature A is included in both systems, it yields partial similarity; when omitted, complete dissimilarity is obtained. Conversely, *full-similarity one side* is obtained when the XOR group models commonality (feature B is part of both configurations in the example shown in Fig. 3.10). This small example shows that by combining different variability constructs it is possible to express variability patterns that are

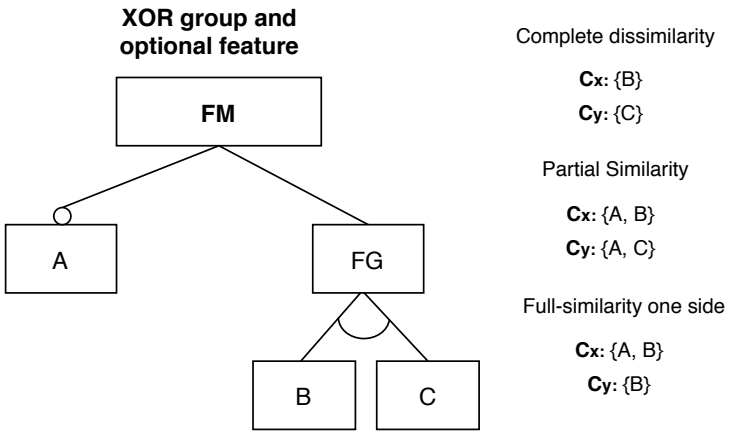

Figure 3.10: *Combining different FM constraints derive in more variability patterns*

not expressable by its constituent parts, such as *partial similarity* in this case.

Optional and *XOR* groups have a *one-to-one* mapping to variability patterns while *OR* groups have a *one-to-many* mapping. Based on Asadi's et. al evaluation framework, it is concluded that these constructs are *complete* since all the variability patterns can be represented. *Clarity* on the other hand, is not accomplished because *OR* groups can represent more than one variability pattern. Taking Asadi's et al mapping a step further, we showed that by combining variability constructs (e.g. *optional* and *XOR* groups) new variability patterns—not individually expressible—can be represented. This means that the postulated mapping is only valid when the constructs are analyzed in isolation but may change when they are combined.

3.4.5 Feature modelling evaluation using ATFV

Table 3.2 shows the results of our representation mapping for feature modelling. Contrary to the mapping proposed by Asadi et al., the result of our evaluation showed that feature modelling cannot represent the concepts of *events* and *lawful event space*. Moreover, instead of claiming that all the concepts that comprise structural variability are mapped to "features", we provide a more detailed mapping of each element to a particular construct in feature modelling (distinguishing, for instance, single features and feature groups). Even though process variability is not supported in feature modelling, we provided an interpretation of these ontological concepts—consistent with the mapping developed for feature modelling—in subsection 3.4.2. This interpretation provides a conceptual reference that can be used to evaluate other VMLs.

The result of the mapping for the variability patterns is the same as that obtained by Asadi et al. [64]. We explained this mapping using a visual representation and argued that this mapping is valid only when feature modelling constructs are considered individually, but change when combined (e.g. optional and XOR group).

		Framework concepts	Feature modelling construct
Variability sources	Structure	Thing	Root node in the feature model.
		Properties in General	Feature group
		Individual properties	Single Features (leaf nodes in the feature tree).
		State	Set of features (configuration) derived from the FM.
		Conceivable state space	Set of all product configurations
		State Law	Constraints among features
		Lawful state space	Set of all valid product configurations
	Process	Events	Non-supported
		Lawful Event Space	Non-supported
Variability Patterns		Full-similarity one side	Optional, OR group
		Partial Similarity	OR group
		Complete dissimilarity	XOR, OR group
		Ordering	Non-supported

Table 3.2: *Feature modelling mapping using ATFV*

The result of assessing feature modelling using ATFV is that the language is ontologically *incomplete* because there are no language constructs that represent process variability sources (*events* and *lawful event space*) and the *ordering variability pattern*. Furthermore, the language is also *unclear* because the *OR* group (multiple selection) can be used to represent more than one variability pattern—a *construct overload* defect. It is important to note that if we only consider structural variability, the language is ontologically complete. We can conclude from that that this language is sufficient if we are only interested in modelling static aspects of variability.

3.5 VMLs expressiveness evaluation using ATFV

In Chapter 2 we compared the capabilities of fourteen languages, the results of which were summarized in Table 2.1. Based on this comparison, let us explore how these languages stand on the Asadi's expressiveness framework by looking into the representation of variability patterns and variability sources.

All the evaluated languages support (directly or indirectly) the three basic forms of variability constructs in feature modelling: *optional*, *alternative* (XOR) and *multiple selection* (OR), therefore, all the evaluated languages are as expressive as feature modelling with respect to the variability patterns *full-similarity one side*, *partial similarity* and *complete dissimilarity*.

Twelve out of fourteen evaluated languages are based on the feature modelling approach; the concepts of features and features groups are directly supported by them. IVML follows the decision modelling approach instead of feature modelling, however, the equivalence between variability represented as variable features and decision variables has been shown by Czarnecki et. al in [13]. Clafer supports feature modelling by using modelling patterns as shown in [37]. Thus, all the evaluated languages can express the ontological concepts of *things*, *properties*

in general, and *individual properties* represented by feature and feature groups.

The ontological concept of *state* was mapped to a set of properties derived from a feature model after resolving all variability (i.e. a product configuration). Since all fourteen languages are able to resolve variability by supporting at least one mechanism to derive configurations, all VMLs can express the concept of state as well. Similarly, all the languages support specification of constraints, thus, all languages express *state laws* and as a consequence, define a *lawful state space*.

All VMLs considered in Chapter 2 are capable of representing the same constructs of feature modelling considered in our representation mapping. Therefore, it can be concluded that all of them are expressive enough to represent *structural variability* using ATFV.

Process variability, on the other hand, includes the ontological concepts of *lawful event state* and the *ordering variability pattern*. The concepts of *event*, *conceivable event space* and *event law* (also called *transition law*) are implicit in *lawful event space*. As concluded in section 3.4.2, none of these are supported by feature modelling.

Among the consulted language specifications, IVML and VSL include the concept of binding time into the language. IVML uses *annotations*—attributes attached to decision variables or projects—to specify binding times [36]. VSL also offers the possibility of specifying upper and lower bounds for binding times, and attaching them to variable items [78]. In both cases, the binding time is specified as an attribute to a variable element in the model. We can assume that other languages (ten out of fourteen in our survey) that support the specification of attributes to arbitrary variable items can also specify binding time.

The last version of Clafer introduces the operator `-->`, which represent a temporal constraint between two predicates. "The meaning of `X --> Y` is that if `X` holds in a state then `Y` must hold in the next state" [37]. Clafer also incorporates patterns to express more complicated temporal constraints; for instance, the expression *always* **A** *between* **B** *and* **C** enforces that action **A** always occurs between **B** and **C**. A complete modelling example of a state machine is presented in [37].

Even though Clafer integrates temporal constructs into the language specification, it is important to note that full tool support is missing. It is stated that as part of release 0.5.0 of Clafer compiler "an experimental generator of Alloy input is also included in the release" [37], however, based on our tool support assessment presented in Chapter 2, here *experimental* means *not working*. Although the tools are not available, a deep understanding of the modelling capabilities offered by *Clafer with behaviour* is certainly a topic worth of further exploration; it is, however, lengthy enough for a full master project onto itself.

3.6 Conclusions

In our literature study, we encountered either a lack of formality in the evaluation of expressiveness of variability languages, or no assessment whatsoever. Asadi et al. adapted Wand

and Weber's ontological expressiveness evaluation framework [62] to variability modelling languages in [64]. Asadi's et al. main contributions are that they use ontological concepts to model variability sources, and add a set of variability patterns to create a consistent framework to assess expressiveness of variability languages. From their work, they derived process variability to point out the existence of variability in the dynamics of systems. In this chapter, we expand upon Asadi's et al. work in [64] by clarifying confusing ontological concepts, providing an alternative and more detailed mapping of feature modelling, and extrapolating this analysis to the fourteen languages compared in Chapter 2. This analysis will be applied towards the selection of a VML to be used at ASML.

The ontological concept of *event* and the *ordering variability* pattern were further clarified by using the *state-tracking* formal model proposed by Wand and Weber in [61]. We postulated that in the ontological sense, an *event* represents a runtime transition between two valid configurations defined in the variability model—i.e. a runtime reconfiguration. This idea has been previously explored in Dynamic Software Product Lines but it was not present in Asadi's et al. work. Furthermore, we also proposed that the *ordering variability* pattern evaluates whether or not two systems traverse the same sequence of runtime reconfigurations in the *same order*.

Our mapping of feature modelling using ATFV rendered it expressive enough to represent structural variability (*things*, *properties* and *lawful state space*) but not process variability (*lawful event space* and *ordering variability pattern*). Hence, our work shows feature modelling (as presented in its original approach [15]) cannot express dynamic aspects of the domain, as concluded by Asadi et al.

Based on the survey presented in Chapter 3, the analysis of feature modelling was extrapolated to the fourteen languages. We concluded that they can represent structural variability based on ATFV. The capability of adding attributes to variability items—not present in the original feature modelling proposal—can be used to specify its binding time—which is dynamic information. Only Clafer incorporates temporal aspects to the language, introducing the possibility of modelling dynamic aspects of variability.

A key interest of ASML's customers is for lithography machines to be continuously running due to the high cost associated with interrupting production; hence, the ability of changing a machine's configuration without restarting the system is a highly desirable one. The current variability modelling approaches—even the most recent proposals—focus only on static aspects of variability, thus, the goal of dynamic configuration of such complex systems is still far on the horizon. Only by integrating dynamic information to variability models is it possible to achieve highly configurable and continuously running systems. Our work sheds light on the importance of developing languages that support dynamic variability further.

The value of ATFV lies in its formal approach to addressing the challenge of expressiveness evaluation by building on previous work as opposed to other authors that build a proposal from scratch (e.g. [79]). Using a pre-existent, well-founded, and systematized ontological framework gives ATFV a solid foundation upon which to specialize in the direction of VMLs analysis. In the same spirit, the work presented in this chapter builds upon Asadi's et al.

work because it is our belief that advancement in any field is mostly the result of collective, iterative effort rather than fresh strikes of individual genius. Hence, we see more value in refining and clarifying confusing ideas rather than discarding them completely before they have been fully explored.

ATFV as used to evaluate the fourteen languages considered in this work, shows an overlap in expressiveness. Most of the languages provide almost the same expressiveness that feature modelling does. Thus, rather than developing new variability languages that are yet another variation of feature modelling, an interesting research direction for ASML could be to focus on integrating dynamic information into existing languages—be they internally or externally developed. The value of this evaluation approach is that it highlights the absense of dynamic aspects of variability—a gap in most of the language proposals—which the previous evaluation in Chapter 2 did not identify. Since system dynamics is a fundamental aspect in the analysis of CPSs such as ASML's lithography machines, this element also proves valuable as an evaluation criteria (RQ1).

In Chapter 2 we concluded that Clafer supported most of the capabilities considered; its support for multiple modelling approaches opens the possibility of specifying hybrid models—relevant for cyber-physical systems—and includes robust tool support. IVML provides similar capabilities, but instead of integrating multiple modelling approaches, its value centers around providing advanced composition mechanisms. As discussed in this chapter, based on ATFV, Clafer stands out among the other specifications by integrating a temporal dimension into the language—something that introduces new possibilities for expressing process variability. Since it is not possible to unequivocally determine suitability of a modelling language without carrying out an in-depth practical assessment, we argue that the above rationale constitutes enough evidence to support the selection of Clafer as the language to be thoroughly evaluated in chapter 4.

Chapter 4

Clafer evaluation using ASML's variability model

In this chapter, Clafer is used to model one of the ASML's Software Product Lines and the System Manager Driver Specification (SMDC) files. Variability models of four different machine families (EXE, NXE, NXT_DRY and NXT_WET) are transformed into Clafer models. For SMDC files, only a mapping from SMDC constructs to Clafer is provided. The purpose of this study is to evaluate the expressiveness of the Clafer language and the scalability of its toolset in ASML's setting.

By developing a model transformation from ASML's variability representation to Clafer, missing constructs in the target language are identified. ASML's variability models are large, and provide an adequate setting to evaluate the Clafer toolset scalability using a real industrial environment—an evaluation that, to the best of our knowledge, has not been done in the past.

Mapping the concepts from SMDC to Clafer constructs allow us to evaluate Clafer both as a structural and variability modelling language. SMDC specifications define system drivers and dependencies among them; variability is specified by a set of inclusion conditions based on VP-values. Then, the information derived from such models and its possible applications are briefly discussed.

This chapter is organized as follows:

- Section 4.1: Description of the VP-based variability model.
- Section 4.2: An overview of the model transformation—including the framework used to implement it—is presented.
- Section 4.3: Explains how VP definitions are modelled in Clafer. Then, the transformation of each kind of VP is detailed.

- Section 4.4: Presents an overview of Clafer constraints expressions and its instance generator. Afterwards, it is detailed how VPO's hierarchical and interface constraints are represented as Clafer's constraints.
- Section 4.5: The results of the model transformation are presented.
- Section 4.6: Models all the SMDC specification constructs in Clafer.
- Section 4.7: The missing Clafer's constructs to model ASML's variability model and SMDC files are summarized and some alternatives outlined.
- Section 4.8: Clafer toolset suitability for modelling variability of ASML's large systems is briefly discussed.
- Section 4.9: Final remarks and conclusions about this modelling experiment are presented.

4.1 ASML's Software Product Line

ASML's TWINSCAN software is an SPL that supports a large variety of platforms, machine models and optional features. The basic variability unit in ASML's variability model is the Variability Parameter (VP). Variability parameters (VPs) are variables that can be evaluated to a set of predefined VP-values; i.e. VPs represent variation points while VP-values specify their variants.

Currently, there are three different VP types: *System Variability Parameters* (SVPs), *Product Variability Parameters* (PVPs) and *Element Variability Parameters* (EVPs). PVPs are further divided into *protected* and *non-protected* (See Fig. 4.1). SVPs are the highest hierarchy among the three categories and represent a commercial value—hardware and software options that the customer can optionally buy.

ASML's variability model defines VP interfaces, which contain a set of—semantically related—PVPs or EVPs and, optionally, dependencies among them. Unlike SVPs, each PVP or EVP definition in the model is always contained in a VP interface. Optional dependencies between VPs of the same category (either PVP or EVP), defined as part of the VP interface specification are called *interface* or *horizontal* constraints throughout this document.

SVPs are mapped to protected PVPs through SVP2PVP mappings while EVP values are derived from PVPs using PVP2EVP mappings; We call these *hierarchical mappings*.

There are two main scenarios for paid software functionality (SVPs): *optional* and *configurable* features. Optional features support specialised hardware or provide application-specific software functionality requested by the end customer; enabled or disabled are the only alternatives. Configurable items refer to software functionality with several operational modes; this might be due to hardware variations or not. The available software modes depend on the software license bought by the user. Protected PVPs are derived from SVPs values and thus,

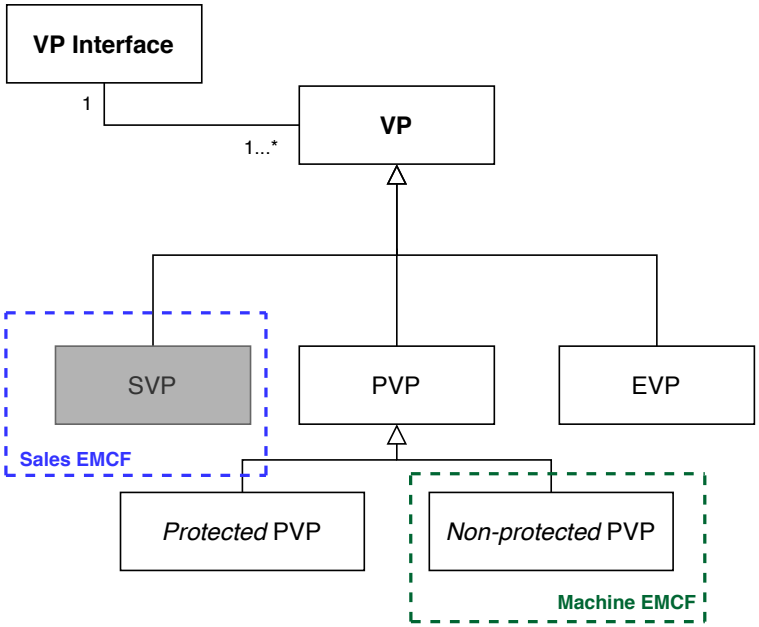

Figure 4.1: *UML diagram - ASML's variability parameters types*

represent the same paid options, but at the product level. Usually a single SVP is mapped to more than one protected PVPs.

Non-protected PVPs deal with software variability with no commercial value. The difference in HW versions (not detectable by software) or user preferences are the most common variability sources. Non-protected PVPs can be modified by the customer.

Finally, EVPs were introduced to decouple external and internal configuration elements. EVPs represent the internal machine configuration directly tied to software modules. EVPs are primarily used to manage variability at the implementation level, improving software maintainability.

Machine configurations—i.e. a mapping between VPs and VP values—are supplied in an EMCF (Encoded Machine Configuration File). There are two different EMCF types: *Sales* and *Machine* EMCFs. Sales EMCFs contain SVP values only and are further divided into certified and non-certified. Certified sales EMCFs are "read-only" configurations, generated by the sales department. Non-certified sales EMCFs are generated using CM tools and can be modified. The certified versions are used by end-customers while non-certified versions are used within ASML for testing and development purposes. On the other hand, Machine EMCFs only contain non-protected PVP values—not included in Sales EMCFs—and can be modified by both ASML engineers and the customer. Protected PVPs assignments are derived from SVP values and SVP2PVP mappings. EVP assignments are determined from protected and non-protected PVPs. Protected PVPs' and EVPs' values are not contained in EMCFs. If for any reason, a protected PVP or a EVP value is not derived from an SVP2EVP or a PVP2EVP mapping respectively, its default value is used.

Default values—also called *safe* values—are included as part of PVP and EVP definitions. When a VP value is neither specified in the EMCF nor derived from a mapping or constraint, the default value is used. A valued VP can be either *pinned* or *overruled*. When a VP is pinned, a new value is assigned to it, ignoring mappings or constraints affecting this variable; only Machine EMCFs and non-certified Sales EMCFs can be pinned. When a VP is re-assigned to its default value, it is overruled. Overrule operation is only available for certified sales ECMFs. Pinning and overruling VP operations must be manually performed using the `EMCF_edit` tool, which is provided by the ASML's configuration management team. These operations are mostly used for debugging configuration issues.

4.2 Overview of the VPO to Clafer transformation

The Configuration Management (CM) team at ASML is responsible for all the VP information—variability models, tools and documentation. Each lithography machine family has its own VP-based model, available through the VP overviews—an internal website where the list of VP definitions and their dependencies can be easily navigated. VP Overviews (VPO) data can be downloaded as XML files—which contain the same information available in the website. Furthermore, the structure of VPO files is specified in an XML Schema Definition (XSD) file; a standard specification that is often used to auto-generate parsers for any file compliant with the schema.

There is a VPO file for each machine type, which encodes the whole variability model—i.e. VP definitions, hierarchical mappings and horizontal constraints. Every VPO file contains two non-empty sets of SVP definitions and *Functional Cluster* definitions (FC). Each FC definition has a non-empty set of *Building Blocks* (BBs), which in turn contain at least one VP interface definition (see Fig. 4.3).

The Eclipse Modeling Framework (EMF) was selected to develop the model transformation from VPO to Clafer. EMF is a robust framework that supports a tools ecosystem for model-based software development. Moreover, EMF is used in different projects within the ASML R&D team; this guarantees the software is pre-approved for installation on the company's computers—which might be a relevant roadblock otherwise. Furthermore, EMF is capable of automatically generating an ECORE model from an XSD definition. In turn, this ECORE model was used to generate Java code for accessing VPO files programmatically.

Two alternatives were considered for the VPO to Clafer transformation: a Model to Model (M2M) and a Model to Text (M2T) transformation. The M2M transformation requires the additional effort of defining an ECORE metamodel for the Clafer language, but this option ensures that the generated Clafer models are always syntactically valid—assuming the ECORE model is correct. The M2T transformation does not require specifying a model for the target language, and the output model is directly written to a text file during the transformation. Considering the project's time constraints and given that Clafer is a minimalist language with very few constructs and simple syntax, the M2T approach was chosen.

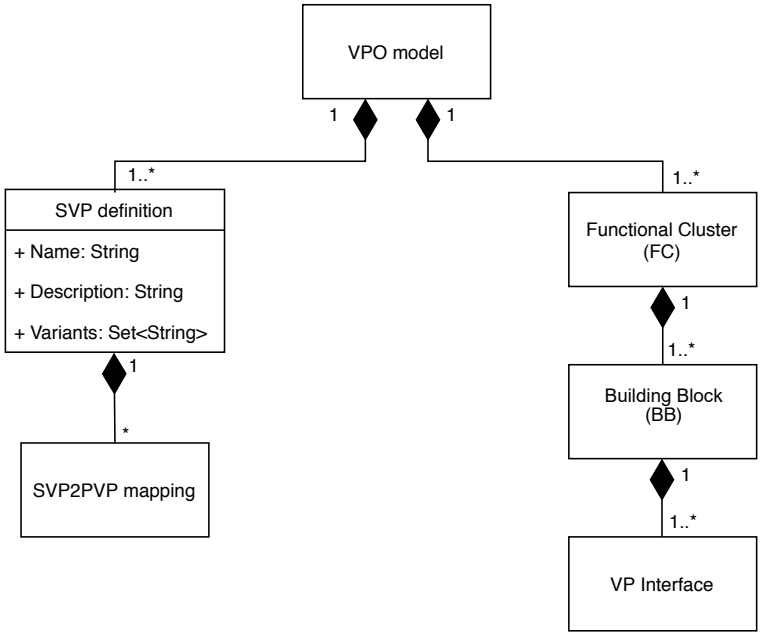

Figure 4.2: *VPO files - Basic structure*

Xtend was the programming language selected to perform the VPO to Clafer model transformation. Xtend is a flexible and expressive dialect of Java, which compiles into readable Java 8 compatible source code [80]. Xtend provides a high-level of abstraction and fewer constructs compared to Java, making it a powerful language and relatively easy to learn. One particularly useful feature for M2T transformation are the template expressions. Xtend's template expressions provide powerful string manipulation constructs and an automatic string output concatenation mechanism resulting in concise and expressive code for out model transformation.

Moreover, due to its Java interoperability, it provides full access to Java libraries. This is important since Xtend's constructs are limited and Java provides useful libraries for our transformation (e.g. regular expressions libraries).

There are two types of elements involved in the VPO to Clafer transformation: VP definitions and constraints. In a VPO file, all the VP information and constraints are contained in either SVP definitions or VP interfaces.

From an SVP definition, its *name*, *description*, the list of *variants* and, if defined, the SVP2PVP mappings are decoded during the *VPO2Clafer* transformation.

VP interfaces contain a non-empty set of PVP or EVP definitions. Optionally, these interfaces contain horizontal constraints (also called interface constraints) and hierarchical mappings. Horizontal constraints specify dependencies among different subsystems variabilities (mapped to different VPs) while hierarchical mappings related VPs at different abstraction levels.

A VP interface might hold a list of PVP definitions (Fig. 4.3 left) or EVP definitions (Fig.

4.3 right), hierarchical mappings are then SVP2PVP and PVP2EVP, respectively. Similar to SVPs, *name*, *description* and *variants* are decoded from PVP or EVP definitions. The boolean attribute *protected* is also decoded from PVPs because it is relevant information to build the variability model in Clafer.

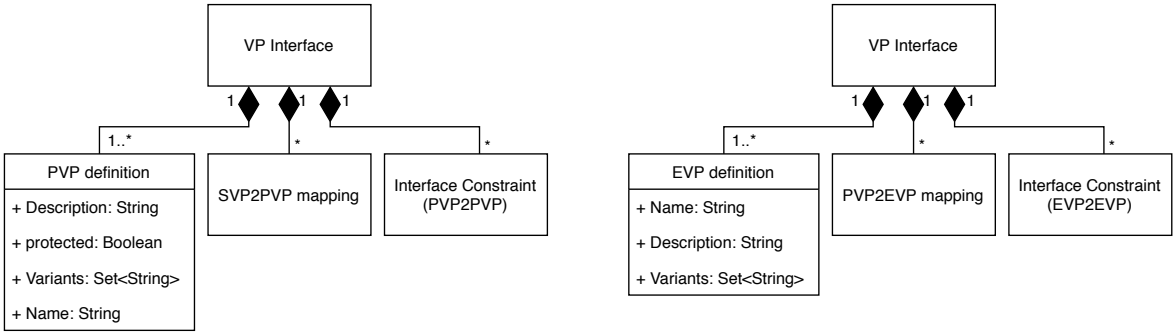

Figure 4.3: *VP interface containing PVP definitions (left) or EVP definitions (right).*

VP definitions in VPO files contain more attributes that are not decoded during the *VPO2Clafer* transformation. These attributes are informative and do not provide variability information. The developed model transformation is limited to translate relevant variability and dependencies information from the model—non-essential information such as the description attribute is also included as context for each variable.

4.3 Variability Parameters in Clafer

This section is divided in two parts. First, Clafer's basic modelling constructs are described and applied to model the VP's definitions. Second, using the VP objects modelled in Clafer, it is detailed how the VP definition information is obtained from the VPO files and translated to Clafer.

4.3.1 Modelling VPs in Clafer

The basic modelling unit in Clafer (the language) is the *clafer*—which stands for ***cl****ass*, ***fe****ature*, ***r****eference*. A clafer is a generic data type that can be used to represent feature models; however, it is flexible enough to build class diagrams and state machines as well. A Clafer model is built from a tree of clafer declarations that represent domain concepts and relations among them. Clafer allows arranging models into multiple specialization and extension layers via constraints and inheritance.

There are two basic types of clafers: *abstract* and *concrete*. An abstract clafer cannot have direct instances on its own but only via concrete clafers which extend it via a generalization relation (similar to abstract classes in OO languages). A clafer (abstract or concrete) is used as

a containment structure; it might contain basic type variables, other clafers, groups of clafers or references. Groups contain a set of elements commonly used to model atomic features—features that cannot be divided further—which together specify a variable's domain.

The Clafer constructs are exemplified by modelling ASML's VPs. Consider the follow definition:

```
1  abstract VP_DEFINE
2      VP_name -> string
3      Description -> string
4
5      xor kind
6          SVP
7          PVP
8          EVP
```

Listing 4.1: Basic VP definition using an abstract clafer

ASML's software variability is modelled by a set of VPs, each one defining a range of possible VP values. There are three different kinds of VPs: SVP, PVP and EVP. Each VP has a unique name and includes a brief description as part of its definition. We start modelling these elements as an abstract clafer shown in Listing 4.1. `VP_DEFINE` contains the VP name, its description and the VP kind. In Clafer, containment is syntactically represented via indentation. `VP_DEFINE` shows two levels of containment, the first level defines the elements included in the top clafer (`VP_name`, `Description` and `xor kind`, a *xor* group with name *kind*), the second indentation for `SVP`, `PVP` and `EVP` elements indicates that those elements are contained within the xor group.

Clafer provides five group cardinality constructs to restrict the number of children elements that can be selected : `xor = 1..1`, `or = 1..*`, `mux = 0..1`, `opt = 0..*` and `range n..m`, where `n`, `m` are integer literals defining the lower and upper bounds, respectively. The first four elements are syntactic sugar for the most used cardinality settings, which can be also expressed using range. Since every VP has one and only one type associated with it, xor is used in all cases.

Now inheritance is used to refine the basic VP definition.

```
1  abstract SVP_DEFINE: VP_DEFINE
2      [ SVP ]
3
4  abstract NON_SVP_DEFINE: VP_DEFINE
5      VPi_name -> string
```

Listing 4.2: SVP and NON-SVP definitions

Clafer inheritance is done using the type annotation: $\langle new\_clafer \rangle : \langle super\_clafer \rangle$. The new clafer specializes the (existing) super clafer—similar to class generalization in object oriented modelling. In Listing 4.2, `SVP_DEFINE` (line 1) and `NON_SVP_DEFINE` (line 4)

are derived from VP_DEFINE, inheriting all its elements. NON_SVP_DEFINE includes a new string variable (VPi_name on line 5), used to store the VP interface name.

Constraint expressions in Clafer are surrounded by square brackets. The constraint [SVP] in Listing 4.2 line 2 is satisfied only if the element SVP is selected from the *kind* group. Notice that the SVP container's name on the constraint expression (i.e. the group's name, kind) is omitted; the Clafer compiler looks automatically in all the nested elements when clafer context constraints are used. Alternatively, the modeller could prepend the container's name to the constraint expression ([kind.SVP]), which is syntactically correct and semantically equivalent.

```
1  abstract EVP_DEFINE: NON_SVP_DEFINE
2      [ EVP ]
3
4  abstract PVP_DEFINE: NON_SVP_DEFINE
5      [ PVP ]
6      xor protected
7          Y
8          N
9  abstract PROTECTED_PVP_DEFINE: PVP_DEFINE
10     [Y]
11
12 abstract UNPROTECTED_PVP_DEFINE: PVP_DEFINE
13     [N]
```

Listing 4.3: EVP and PVP definitions

Listing 4.3 shows further refinements to define EVPs (line 1) and PVPs (line 4). Both EVP_DEFINE and PVP_DEFINE specialize NON_SVP_DEFINE and contain a constraint for setting the VP *kind* (similar to SVP_DEFINE). The PVP definition is further refined in two derived versions, protected (line 9) and non-protected (line 12).

4.3.2 Decoding VP definitions from a VP overviews file

Thus far, the Clafer model contains only abstract clafers that represent the different VP types. Bear in mind that abstract definitions do not represent VP *instances* but VP *classes*. Each VP definition found in the VPO file is then translated to a *concrete* clafer in the output model, representing VP instances. First, the SVPs decoding process is described, then PVPs and finally EVPs.

Listing 4.4 shows an SVP definition for the NXT-WET machine type. The resulting element in the output model is shown in Listing 4.5. The clafer definition is derived from SVP_DEFINE (line 1) and extended with three variants (lines 6–8). The first two variants ('Y' and 'N') are obtained from the input model while the third variant is introduced during the transformation.

```
1  <SVP Name="2D_BARCODE_READER" Description="2D Barcode Reader" Microhelp="TBD">
2      <VARIANT Name="N" Description="No" />
3      <VARIANT Name="Y" Description="Yes" />
4      <CONTROLS Kind="PVP">
5          <VP Name="BC2D_READER" VPInterface="BCCMxVPxTYP" />
6      </CONTROLS>
7  </SVP>
```

Listing 4.4: Example SVP definition in VPO file

```
1  SVP_2D_BARCODE_READER: SVP_DEFINE
2      [ VP_name = "2D_BARCODE_READER" ]
3      [ Description = "2D Barcode Reader" ]
4
5      xor variant
6          N
7          Y
8          SVP_UNDEFINED
9
10     [ SVP_UNDEFINED ]
```

Listing 4.5: Example SVP definition translated to Clafer

ASML's machine configuration files allow undefined SVPs. It was decided to model each SVP's undefined state explicitly by adding an extra variant, `SVP_UNDEFINED` (see Listing 4.5 line 8). Then, all the SVPs were set as undefined by default (constraint shown on line 10). The main reason to do so is to simplify the constraint evaluation, which was found to be the main challenge for Clafer's tools. `VP_name` and `Description` are set using constraints on line 2 and line 3 respectively.

Unlike previous definitions, this is a concrete clafer (`abstract` keyword is absent). The substring "`SVP_`" is appended to the SVP's name during the model transformation because Clafer's ID cannot start with numbers (`2D_BARCODE_READER` is an invalid identifier); the same operation is applied to all SVPs (whether or not its name start with a digit) to keep a consistent name convention.

There are two elements showed in Listing 4.4 that are not decoded because they do not contain essential variability information: the `Microhelp` attribute (line 1) and the `CONTROLS` structure (lines 4–6). The microhelp provides complementary description information, although sometimes it is not defined (as in the running example). The `CONTROLS` structure specifies the kind `PVP` (line 4), meaning that this SVP belongs to at least one SVP2PVP mapping defined in the model—but it does not provide the actual mapping. Inside the `CONTROLS` structure there is a list with all the PVPs controlled by this SVP (line 5), in this example, only one element is defined, but it might be—and usually are—many more. It is clear that none of these elements provide essential variability or constraint information, thus, they are not translated.

```
1  <VPINTERFACE Name="WHPUxVPxCOSY">
2      <VP Name="WFR_TEMP_COND_SYSTEM"
3          Description="WH temperature conditioning type"
4          Number="11080"
```

```
5           Kind="PVP"
6           Protected="true"
7           Default="CONVENTIONAL"
8           Type="Feature"
9           Microhelp="Indicates the wafer conditioning procedure
10                    that will be used in the wafer handler."
11          DocId="D000350315"
12          Version="00">
13
14          <VARIANT Name="CONVENTIONAL"
15                   Description="Slow performance type used for XTIII and below."
16                   Number="11081" />
17          <VARIANT Name="TYPE_1"
18                   Description="Fast performance type used for XTIV and higher."
19                   Number="11082" />
20          <VARIANT Name="TYPE_2"
21                   Description="Mk5 Store Unit conditioning active."
22                   Number="11083" />
23          <VARIANT Name="TYPE_4"
24                   Description="Mk5 Store Unit and pre-aligner conditioning
                         active"
25                   Number="11085" />
26
27          <MMDEF>
28              <NXT1950Ai MmDefault="TYPE_1"
29                         Source="CMIM_WHPUxVPxCOSY_mm_def.xml"
30              />
31              <NXT1960Bi MmDefault="TYPE_1"
32                         Source="CMIM_WHPUxVPxCOSY_mm_def.xml"
33              .
34              .
35              <NXT2100i MmDefault="TYPE_4"
36                        Source="CMIM_WHPUxVPxCOSY_mm_def.xml"
37              />
38          </MMDEF>
39
40          <CONTROLLED_BY Kind="SVP">
41              <VP Name="MES_MACHINE_TYPE" />
42              <VP Name="OFP_1" />
43              .
44              .
45              <VP Name="SNEP_C2D" />
46              <VP Name="TOP_PRODUCTS_NXT1965CI" />
47          </CONTROLLED_BY>
48
49          <CONTROLS Kind="EVP">
50              <VP Name="WFR_LOAD_PATH_THERMAL_COND" VPInterface="LOPWxEVPxWH" />
51              <VP Name="WH_HMAN_USAGE" VPInterface="WBTCxEVPxHTxCONF" />
52              <VP Name="WH_SU_TABLE_USAGE" VPInterface="WHSUxEVPxTABLE" />
53          </CONTROLS>
54      </VP>
```

Listing 4.6: Example of a protected PVP definition, including Machine default values.

Now, consider the VP interface containing a protected PVP definition shown in Listing 4.6. At the top of the structure we find the VP interface, containing its name as an attribute

(line 1). Inside, there is a PVP definition with several attributes (lines 2–12), from which only name (line 2), description (line 3), kind (line 5) and protected (line 6) are decoded. The default value (line 7) is a deprecated attribute—and hence ignored—because it was replaced by machine model default (MMDEF) values (lines 27–38). The remaining parameters are not translated either, because they do not carry relevant information for our analysis. Four PVP's variants are defined on lines 14–25; only their name attribute is decoded.

Right below the variants there is a MMDEF block (lines 27–38) defining the machine defaults (some elements were removed to save space). When a PVP or an EVP does not have a value or its value is overruled, their default value (also called safe value) is used. Instead of defining a unique default value per PVP (a deprecated feature), the MMDEF list defines a default value for each machine type. For instance, when the machine type is `NXT1950Ai` the default value is `TYPE_1` (line 28). These machine defaults values are originally stored in separate XML files and encoded into the VPO file, the attribute `Source` indicates the name of such an external file (line 29); again, an ignored attribute with no relevant information for our analysis.

Unfortunately, Clafer does not support default values directly and therefore these are not translated in the target model. Nonetheless, alternatives for indirect support are briefly discussed in Section 4.7.

Finally, `CONTROLLED_BY` (lines 40–47) and `CONTROLS` (lines 49–53) structures are shown under the PVP definition. `CONTROLLED_BY` defines a list of SVPs (line 40) used in a SVP2PVP mapping, which define the value for the current PVP. Since it is a protected PVP, its value is automatically derived from one or more SVPs. `CONTROLS` use was previously explained, the only difference in this case is that now the mapped (or controlled) variables are EVPs (as indicated on line 49), indicating that this PVP is part of a PVP2EVP mapping.

The resulting PVP definition in Clafer is presented in Listing 4.7. The protected attribute (`true` in this case) determines which super clafer is used (i.e. protected or non-protected). In the VPO constraint expressions (that will be explained later on), PVPs and EVPs are referenced using an identifier of the form: `VPi_name:VP_name`. For example:

```
WHPUxVPxCOSY:WFR_TEMP_COND_SYSTEM
```

The VP reference name cannot be used as it appears in VPO files because the symbol ":" is interpreted as a type annotation in Clafer, hence the colon is replaced by an underscore as shown on line 1 in Listing 4.7. Notice that this might lead to ambiguity because the `VP_name` also contains underscore characters. However, the name of both the VP and its associated interface are stored inside the VP definition; which can be used to resolve ambiguity if needed.

```
1  WHPUxVPxCOSY_WFR_TEMP_COND_SYSTEM: PROTECTED_PVP_DEFINE
2      [ VPi_name = "WHPUxVPxCOSY" ]
3      [ VP_name = "WFR_TEMP_COND_SYSTEM" ]
4      [ Description = "WH temperature conditioning type" ]
5
6      xor variant
7          CONVENTIONAL
8          TYPE_1
```

```
9           TYPE_2
10          TYPE_4
```

Listing 4.7: Protected PVP definition translated to Clafer

Lastly, an example VP interface containing an EVP definition is shown in Listing 4.8. The main differences with respect to the previous PVP definition are the following:

- *Protected* attribute (line 6): Even though the attribute is included, it is deprecated for EVPs. Still, some EVP definitions have different values in this field.
- *Default* value (line 7): The default attribute is deprecated for PVPs but not for EVPs. On a well-formed VPO model, all the EVPs values are derived from PVPs or set directly using a mapping (i.e. not depending on any PVP). If a particular EVP does not get a value from a mapping, then its default value is used.
- *MMDEF* block: Machine defaults do not apply to EVPs, default attribute is used instead (see previous bullet).
- *CONTROLS* block: EVPs are at the bottom of the VP hierarchy, thus, CONTROLS block does not apply.

```
1  <VPINTERFACE Name="KDDAxEVPxRBF">
2      <VP Name="RBF_RELAXATION"
3          Description="Toggle relaxation of the RBF model."
4          Number="29160"
5          Kind="EVP"
6          Protected="false"
7          Default="DISABLED"
8          Type="Feature"
9          Microhelp="Toggle relaxation of the RBF model."
10         DocId="579758"
11         Version="00">
12
13         <VARIANT Name="DISABLED"
14                  Description="Disable RBF Relaxation" Number="29161"
15         />
16         <VARIANT Name="ENABLED"
17                  Description="Enable RBF Relaxation" Number="29162"
18         />
19
20         <CONTROLLED_BY Kind="PVP">
21             <VP Name="SWITCH_303" VPInterface="CMIMxVPxCFG" />
22         </CONTROLLED_BY>
23     </VP>
```

Listing 4.8: Example of an EVP definition

The resulting EVP definition in the output model is presented in Listing 4.9.

```
1 KDDAxEVPxRBF_RBF_RELAXATION: EVP_DEFINE
2         [ VPi_name = "KDDAxEVPxRBF" ]
3         [ VP_name = "RBF_RELAXATION" ]
4         [ Description = "Toggle relaxation of the RBF model." ]
5
6         xor variant
7                 DISABLED
8                 ENABLED
```

Listing 4.9: EVP definition translated to Clafer

4.4 VPO constraints in Clafer

Clafer constraints are boolean expressions defined in a Clafer model that are required to be true and are used for instance generation. A model instance is correct if and only if all constraints hold. Constraints in Clafer have two different contexts: top-level or clafer level. Top-level constraints are defined without indentation and must be true for each instance of the model. Clafer level constraints are nested under a clafer's definition and must be true for each instance of the containing clafer. Both cases are shown below.

```
/*Top level constraint*/
[ < boolean expression > ]

/*Clafer context constraint*/
<clafer>
    [ < boolean expression > ]
```

4.4.1 Boolean Expressions in Clafer

Clafer supports different categories of expressions that return a boolean value: boolean logic, numeric comparisons, simply quantified and quantified with local declarations. Only the first two categories are used in this project and specified below.

Boolean Logic

```
<boolean expression>:
     if <boolean expression> then <boolean expression> else <boolean
         expression>
    <boolean expression> <=> <boolean expression>
    <boolean expression> => <boolean expression>
    <boolean expression> || <boolean expression>
    <boolean expression> xor <boolean expression>
    <boolean expression> && <boolean expression>
    ! <boolean expression>
```

Clafer supports the basic boolean operators: *conjunction* (`&&`), *disjunction* (`||`) and *negation* (`!`). Besides, *implication* (`=>`), *double-sided implication* (`<=>`), *exclusive disjunction* (`xor`)

and the `if..then..else` construct are also supported—all of them expressible using basic boolean operators. The if construct does not support nesting. The VPO constraints heavily rely on *else-if* constructs and *nested* if structures; neither of those are expressible with Clafer's if construct. Thus, this Clafer construct is not used in our model transformation.

Numeric comparisons

```
<boolean expression>:
     <numeric expression> < <numeric expression>
    <numeric expression> > <numeric expression>
    <numeric expression> <= <numeric expression>
    <numeric expression> >= <numeric expression>
```

Constraints involving numeric comparisons are useful to limit integer values. For instance, if an integer variable represents the angle of rotation in degrees, its domain might be constrained as follows:

```
[(degrees >= 0) && (degrees <= 360)]
```

Neither boolean variables nor boolean literals are part of the Clafer language. The lack of boolean variables makes the model transformation more complex and results in very large constraint expressions. Boolean literals—required in the model transformation—are modelled with the following clafer:

```
Boolean
    xor is
        true
        false
    [is.true]
```

Constraints `[Boolean.is.true]` and `[Boolean.is.false]` evaluate to *true* and $false$ respectively, and are used as a replacement for boolean literals in our Clafer model.

4.4.2 Clafer constraints and variability model resolution

The simplified Clafer's instance generation process in shown in Fig. 4.4. The Clafer model is compiled into a Constraint Satisfaction Problem (CSP). A CPS is formally defined as a triple $\langle X, D, C \rangle$, where [81]:

$X = X_1, ..., X_n$ is a set of variables
$D = D_1, ..., D_n$ is a set of all possible values that can be assigned to each variable
$C = C_1, ..., C_m$ is a set of constraints.

A value assignment is a variable-value pair $\langle X_i, v \rangle$, where $X_i \in X$ and $v \in D_i$. A constraint $C_j \in C$ defines a relation R_j between the set of variables t_j, where $t_j \subset X$. R_j is a k-ary

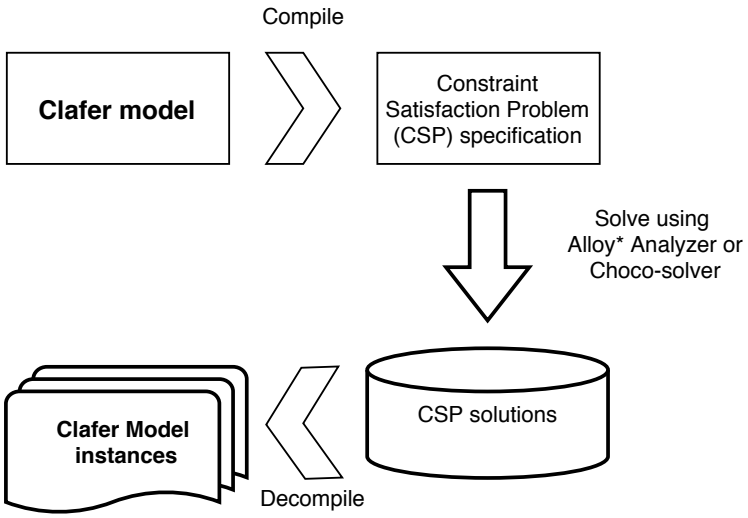

Figure 4.4: *Simplified workflow for the Clafer instance generator.*

relation between the subset of domains D_j, where the subset $D_j \subset D$ corresponds to the variables in t_j.

A mapping from a subset of variables to a set of values in their respective subset of domains is an *evaluation*. When the values assigned to the variables t_j satisfy the relation R_j, we say that the evaluation w_j satisfies C_j.

A *consistent* evaluation does not violate any of the constraints in C. A *complete* evaluation includes all variables in X. An evaluation is a *solution* of the CSP if it is consistent and complete.

In our Clafer model, each X_i corresponds to a VP definition while its associated variants define the domain D_i. All the model constraints, translated from VPO hierarchical mappings and interface constraints, form the set of constraints C.

The Clafer instance generator first transforms the Clafer model to a CSP specification (.als format for Alloy* Analyzer and .js for Choco-solver). Then, one of the backend solvers takes the CSP specifications and searches for all the solutions. Each solution found is then translated back to Clafer and presented as a model instance. Each model instance specifies a single value for each VP defined in the model.

VPO hierarchical and interface constraints reduce the set of all possible model instances to a subset of valid instances, i.e. valid machine configurations. VP assignments are contained in EMCFs, when those assignments are introduced to the Clafer model as additional constraints, a single machine configuration is obtained. In the following subsection the transformation of VPO constraints is detailed.

4.4.3 VPO constraints as propositional formulae in Clafer

VPO model's constraints can be expressed using only propositional formulae in Clafer. In general, a proposition is a declarative sentence that is either *true* or *false*; thus, we model propositions as boolean expressions that evaluate to true or false.

Consider an arbitrary variable VP with a domain $V = \{v_1, ..., v_n\}$. A proposition p_i is defined as $(VP == v_i)$ where $v_i \in V$ (the symbol == is used to represent the comparison operator). p_i yields *true* when the variant v_i is selected for the variability parameter VP, *false* otherwise.

All the VPs in the model define a non-empty set of variants, from which one and only one value must be selected (*xor* group has cardinality 1..1). Thus, all the CSP solutions require that one variant is assigned to each VP in the variability model.

All the constraints in the VPO model are expressed using only propositions and the boolean operators *negation* ($\neg$), *conjunction* ($\wedge$), *disjunction* ($\vee$) and *implication* ($\Longrightarrow$). The most elementary constraint consists of a single proposition, which assigns a variant to a VP, we call it *assignment constraint*. Assignment constraints are used when a VP value is fixed using VPO constraints or when a machine configuration (i.e. an EMCF file) is integrated to the Clafer model.

If a proposition is negated ($\neg p_i$), it means that such variant cannot be assigned to the VP, i.e. $(VP\ != v_i)$. A negated proposition effectively removes a variant from the variable's domain, i.e. $V = \{v_1, ..., v_n\} \backslash \{v_i\}$.

An *unguarded* constraint expression consists of a non-empty set of propositions using negation, conjunction or disjunction operators. If the implication operator is used, a *guarded* constraint expression is obtained. Consider three different VPs and the following propositions p_i, p_j, q_k and r_l. These propositions are translated as $(VP_p == v_{p_i})$, $(VP_p == v_{p_j})$, $(VP_q == v_{q_k})$ and $(VP_r == v_{r_l})$, respectively.

$$p_i \wedge q_k \wedge \neg r_l \tag{4.1}$$

$$p_i \wedge p_j \tag{4.2}$$

Eq. 4.1 sets VP_p and VP_q to a specific value and removes a variant from the domain values of VP_r. Equation 4.2 is unsatisfiable (i.e. it always yields false) because VP_p—or any other VP—cannot hold two values at the same time.

Now, consider the following guarded constraint expressions.

$$p_i \implies r_l \tag{4.3}$$

$$p_i \implies \neg r_l \tag{4.4}$$

$$((p_i \vee p_j) \wedge q_k) \implies r_l \tag{4.5}$$

Eq. 4.3 is equivalent to v_{p_i} ***requires*** v_{r_l} while Eq. 4.4 represents v_{p_i} ***excludes*** v_{r_l}. Most of the constraints resulting from the transformation are guarded constraints. Among these constraints, the left-hand side (lhs) expression usually contains multiple propositions and the right-hand side (rhs) is an assignment constraint (e.g. Eq. 4.5).

Strictly speaking, a propositional logic formula does not support the implication operator. However, the following logical equivalence is used during the transformation of the variability model to a CSP problem:

$$(p \implies q) \iff (\neg p \vee q) \tag{4.6}$$

The previously described guarded and unguarded constraints suffice to express all the VPO model's constraints in Clafer.

An assignment constraint, containing a single proposition, is expressed in Clafer as follows:

```
[ <VP_name>.variant.<variant_name> ]
```

The proposition from above means the variant <variant_name> is assigned to the VP <VP_name>. A guarded constraint has the following form:

```
[ <constraint_expression_lhs> => <constraint_expression_rhs> ]
```

A minimal Clafer example that illustrates these guarded and unguarded constraints is presented in Listing 4.10. The *dynamic dose controller* VP (defined on lines 1–8) is set to VERSION_1 using an unguarded constraint (line 11). The *power dose limiting* variable (defined in lines 13–20) is set using guarded constraints that depend on the *dynamic dose controller's* value (two constraints defined on lines 24–27 and lines 29–33); these constraints are part of a PVP2EVP mapping. Many mappings follow the same structure, but the guard condition usually contains multiple variables.

```
1  CMEUxVPxDOSE_DYNAMIC_DOSE_CONTROLLER: PROTECTED_PVP_DEFINE
2      [ VPi_name = "CMEUxVPxDOSE" ]
3      [ VP_name = "DYNAMIC_DOSE_CONTROLLER" ]
4      [ Description = "Dynamic dose controller" ]
5
6      xor variant
7          NONE
8          VERSION_1
9
10 /*Assignment constraint*/
11 [ CMEUxVPxDOSE_DYNAMIC_DOSE_CONTROLLER.variant.VERSION_1 ]
12
13 WDxEVPxPWRxLIM_WD_POWER_DOSE_LIMITING: EVP_DEFINE
14     [ VPi_name = "WDxEVPxPWRxLIM" ]
15     [ VP_name = "WD_POWER_DOSE_LIMITING" ]
16     [ Description = "CPD Power controlling based on requested dose" ]
17
18     xor variant
19         DISABLED
20         ENABLED
21
22 /*PVP2EVP mapping -Guarded constraints*/
23
24 [ (!CMEUxVPxDOSE_DYNAMIC_DOSE_CONTROLLER.variant.NONE)
25    =>
26    WDxEVPxPWRxLIM_WD_POWER_DOSE_LIMITING.variant.ENABLED
27 ]
```

```
28
29 [
30   CMEUxVPxDOSE_DYNAMIC_DOSE_CONTROLLER.variant.NONE
31   =>
32   WDxEVPxPWRxLIM_WD_POWER_DOSE_LIMITING.variant.DISABLED
33 ]
```

Listing 4.10: Simple guarded and unguarded constraints

Every VP in the Clafer model is either *under-constrained*, *constrained*, or *over-constrained*. When a VP is under-constrained, it is possible to select more than one variant for the VP. Under-constrained VPs either have no constraints or have some constraints that shrink their domain to more than one element (using negated propositions, for instance). A VP which only can take one variant is constrained. If there is an unsatisfiable constraint causing that no variant can be selected, we say it is an over-constrained VP. A variability model with over-constrained variables has no valid instances, i.e. the CSP has no solutions. Thus, it is considered an *invalid* model.

Listing 4.11 shows an SVP definition (lines 1–8) with three constraints (A-C defined on lines 10–12 respectively) to illustrate each constraint scenario. When only constraint A is specified, the SVP is under-constrained—either `Y` or `PS` can be selected. The same scenario occurs when no constraints are specified. The constraint B ensures that only `Y` can be selected, which makes the SVP constrained. `SVP_SPOTLESS_NXE` is still constrained if only constraint C or both constraint A and B are specified.

```
1  SVP_SPOTLESS_NXE: SVP_DEFINE
2       [ VP_name = "SPOTLESS_NXE" ]
3       [ Description = "Spotless NXE" ]
4
5       xor variant
6          N
7          PS
8          Y
9
10 [!SVP_SPOTLESS_NXE.variant.N]  /*Constraint A*/
11 [SVP_SPOTLESS_NXE.variant.Y]   /*Constraint B*/
12 [SVP_SPOTLESS_NXE.variant.N]   /*Constraint C*/
```

Listing 4.11: Example that illustrates the three different constraint scenarios

The remaining combination of constraints makes the variable over-constrained. It is impossible, for instance, that both `Y` and `N` are selected (constraints B and C). With the large set of constraints contained in the VPO models, a minimal error in the implementation or even certain dependency scenarios—correctly translated—might lead to over-constrained—and thus invalid—models. It is crucial to ensure that the constraints introduced to the target model do not make the CSP problem unsatisfiable.

4.4.4 VP Interface Constraints

Interface constraints are translated to both guarded and unguarded constraints. An example of the latter is shown in Listing 4.13. Each constraint that the model needs to satisfy is specified in the `Condition` attribute of each `MUSTHOLD` item (lines 2–10). Conditions have two comparison scenarios: equal (==) and not equal (!=), which translate a proposition or its negation in Clafer. The condition is given as a single string, which is parsed during the model transformation.

```
1  <CONSTRAINTS Source="CMIM_GVxVPxMACHxTYPE_constr.xml">
2      <MUSTHOLD Condition="GVxVPxMACHxTYPE:MACHINE_ARCHITECTURE == NXT" />
3      <MUSTHOLD Condition="GVxVPxMACHxTYPE:MAX_WAFER_SIZE == 300MM" />
4      <MUSTHOLD Condition="GVxVPxMACHxTYPE:WAVELENGTH_RANGE == DUV" />
5      <MUSTHOLD Condition="GVxVPxMACHxTYPE:XT_ARCH_REVISION == NONE" />
6      <MUSTHOLD Condition="GVxVPxMACHxTYPE:NXT_ARCH_REVISION != NONE" />
7      <MUSTHOLD Condition="GVxVPxMACHxTYPE:NXE_ARCH_REVISION == NONE" />
8      <MUSTHOLD Condition="GVxVPxMACHxTYPE:QXT_ARCH_REVISION == NONE" />
9      <MUSTHOLD Condition="GVxVPxMACHxTYPE:QXE_ARCH_REVISION == NONE" />
10     <MUSTHOLD Condition="GVxVPxMACHxTYPE:EXE_ARCH_REVISION == NONE" />
11 </CONSTRAINTS>
```

Listing 4.12: Unguarded interface constraints example

The conditions shown in Listing 4.12 are translated as a set of unguarded constraints in Clafer (Listing 4.13). Some variants names in the VPO specification (e.g., `300MM` in Listing 4.12 line 3) start with a digit, which is an invalid identifier in Clafer. These invalid names are detected and fixed by appending an 'n' character (see line 2 in the Listing below) during the transformation process.

```
1  [ GVxVPxMACHxTYPE_MACHINE_ARCHITECTURE.variant.NXT ]
2  [ GVxVPxMACHxTYPE_MAX_WAFER_SIZE.variant.n300MM ]
3  [ GVxVPxMACHxTYPE_WAVELENGTH_RANGE.variant.DUV ]
4  [ GVxVPxMACHxTYPE_XT_ARCH_REVISION.variant.NONE ]
5  [ !GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.NONE ]
6  [ GVxVPxMACHxTYPE_NXE_ARCH_REVISION.variant.NONE ]
7  [ GVxVPxMACHxTYPE_QXT_ARCH_REVISION.variant.NONE ]
8  [ GVxVPxMACHxTYPE_QXE_ARCH_REVISION.variant.NONE ]
9  [ GVxVPxMACHxTYPE_EXE_ARCH_REVISION.variant.NONE ]
```

Listing 4.13: Unguarded interface constraints translated to Clafer

VPO interface constraints might specify a block of `MUSTHOLD` elements guarded by an `IF` condition (see Listing 4.14 lines 3 and 11). Horizontal constraints with if blocks are translated as guarded constraints to Clafer; the output is shown in Listing 4.15. The `MUSTHOLD` conditions in the given example contain ill-formed VP references. `WP_SETPOINT_A_J_RATIO` (lines 7 and 19 in Listing 4.14) does not include its container VP interface name–more cases were found in other VPO models. The implemented model transformation was enhanced to keep track of the VP interfaces and its associated VPs. When an incorrect reference was found, it was fixed by appending the VP interface name (see lines 7 and 20 in Listing 4.15).

```
<CONSTRAINTS Source="CMIM_WPPACMxVPxCFG_constr.xml">

    <IF Condition="(GVxVPxMACHxTYPE:NXT_ARCH_REVISION == REV1) OR
                   (GVxVPxMACHxTYPE:NXT_ARCH_REVISION == REV1_1) OR
                   (GVxVPxMACHxTYPE:NXT_ARCH_REVISION == REV2)">

        <MUSTHOLD Condition="WP_SETPOINT_A_J_RATIO == VARIABLE" />

    </IF>

    <IF Condition="(GVxVPxMACHxTYPE:NXT_ARCH_REVISION != REV1) AND
                   (GVxVPxMACHxTYPE:NXT_ARCH_REVISION != REV1_1) AND
                   (GVxVPxMACHxTYPE:NXT_ARCH_REVISION != REV2) AND
                   (GVxVPxMACHxTYPE:NXT_ARCH_REVISION != REV3) AND
                   (GVxVPxMACHxTYPE:NXT_ARCH_REVISION != REV3_1) AND
                   (GVxVPxMACHxTYPE:NXT_ARCH_REVISION != REV3_2) AND
                   (GVxVPxMACHxTYPE:NXT_ARCH_REVISION != UNKNOWN)">

        <MUSTHOLD Condition="WP_SETPOINT_A_J_RATIO == CONSTANT" />

    </IF>

</CONSTRAINTS>
```

Listing 4.14: Guarded interface constraints example

```
/* VP interface CONSTRAINTS */
[
  (( GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.REV1 )   ||
   ( GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.REV1_1 ) ||
   ( GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.REV2 ))
   =>
   WPPACMxVPxCFG_WP_SETPOINT_A_J_RATIO.variant.VARIABLE
]

/* VP interface CONSTRAINTS */
[
  (( !GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.REV1 ) &&
   ( !GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.REV1_1 ) &&
   ( !GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.REV2 ) &&
   ( !GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.REV3 ) &&
   ( !GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.REV3_1 ) &&
   ( !GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.REV3_2 ) &&
   ( !GVxVPxMACHxTYPE_NXT_ARCH_REVISION.variant.UNKNOWN ))
    =>
    WPPACMxVPxCFG_WP_SETPOINT_A_J_RATIO.variant.CONSTANT
]
```

Listing 4.15: Guarded interface constraints example in Clafer

4.4.5 VPO hierarchical mappings

The VPO hierarchical mappings (i.e *SVP2PVP* and *PVP2EVP*) use XML elements to split the constraint expression—not a single string as in the interface constraints. Mapping's constructs are combined in different ways to form constraint expressions, resulting in many more cases to handle compared to the interface constraints.

The NXT_WET is used as the reference model for this section. This machine type has the largest and most complex variability model; the remaining models represent a subset of the scenarios considered for this one.

Almost all the SVP2PVP mappings are defined within VP interface elements, with the only exception of a mapping linked to the SVP `MES_MACHINE_TYPE`. This machine type mapping defines a large set of *Named Expressions* (NEs). These NEs are boolean expressions assigned to variables and then used in many of the SVP2PVP mappings throughout the model—the system's variability is largely driven by the machine type.

First, we describe how NEs are decoded since model's dependencies are built upon them.

Named Expressions and Named Expression Assignments

Named Expressions are boolean variables that depend on SVP or PVP values. NEs are defined using the so-called *Named Expression Statements* (NESes), which declare and initialize them. A NES starts with the `expression` keyword followed by a unique name, and then, a boolean expression. Listing 4.16 shows the first lines of a machine type's SVP2PVP mapping, which contain mostly NESes. For clarity purposes, a human-readable format—provided as XML comments in VPO files—is used to present mapping's snippets. As a reference, Listing 4.17 shows the actual XML fragment for just the `IS_1970_HYBRID` assignment expression (lines 17–18 in Listing 4.16), which is the actual representation being parsed.

```
1  if defined(:MES_MACHINE_TYPE)
2  {
3      expression MMT_IS_SAWS_org = ("SAWS" in :MES_MACHINE_TYPE);
4      expression MMT_IS_1950_org = ("1950" in :MES_MACHINE_TYPE);
5      expression MMT_IS_1960_org = ("1960" in :MES_MACHINE_TYPE);
6      expression MMT_IS_1965_org = ("1965" in :MES_MACHINE_TYPE);
7      expression MMT_IS_1970_org = ("1970" in :MES_MACHINE_TYPE);
8      expression MMT_IS_1980_org = ("1980" in :MES_MACHINE_TYPE);
9      expression MMT_IS_2000_org = ("2000" in :MES_MACHINE_TYPE);
10     expression MMT_IS_2050_org = ("2050" in :MES_MACHINE_TYPE);
11     expression MMT_IS_2100_org = ("2100" in :MES_MACHINE_TYPE);
12
13     expression HAS_1970_OPTIONS = defined(:PEP_1970) || defined(:OFP_1) ||
14                                   defined(:OFP_2) || defined(:SNEP_C2D) ||
15                                   defined(:SNEP2000);
16
17     expression IS_1970_HYBRID = (MMT_IS_1950_org || MMT_IS_1960_org) &&
18                                 HAS_1970_OPTIONS;
19     expression MMT_IS_SAWS = MMT_IS_SAWS_org;
```

```
20      expression MMT_IS_1950 = MMT_IS_1950_org && !IS_1970_HYBRID;
21      expression MMT_IS_1960 = MMT_IS_1960_org && !IS_1970_HYBRID;
22      expression MMT_IS_1965 = MMT_IS_1965_org;
23      expression MMT_IS_1970 = MMT_IS_1970_org || IS_1970_HYBRID;
24      expression MMT_IS_1980 = MMT_IS_1980_org;
25      expression MMT_IS_2000 = MMT_IS_2000_org;
26      expression MMT_IS_2050 = MMT_IS_2050_org;
27      expression MMT_IS_2100 = MMT_IS_2100_org;
```

Listing 4.16: Named expression statements example

```
1  <CONSTRAINTS Source="CMIM_WPPACMxVPxCFG_constr.xml">
2  <named_expr_stmt linenr="62">
3           <keyword>expression</keyword>
4           <name>IS_1970_HYBRID</name>
5           <operator>=</operator>
6           <mult_expr>
7               <par_expr>(
8                   <mult_expr>
9                           <named_expr>
10                                  <name>MMT_IS_1950_org</name>
11                          </named_expr>
12                          <operator>||</operator>
13                          <named_expr>
14                              <name>MMT_IS_1960_org</name>
15                          </named_expr>
16                  </mult_expr>
17          )</par_expr>
18          <operator>&amp;&amp;</operator>
19          <named_expr>
20              <name>HAS_1970_OPTIONS</name>
21          </named_expr>
22          </mult_expr>
23          ;
24 </named_expr_stmt>
```

Listing 4.17: IS_1970_HYBRID definition in XML

The boolean expressions in the NES use a set a language constructs specified in the VPO's XSD file. Instead of presenting XSD fragments, the information was translated to the Backus-Naur form (BNF) notation—an elegant and concise notation, widely used in programming language theory. The syntactic structure of NES is shown in Listing 4.18.

```
1
2  <name>::= type:String
3  <key>::= type:String
4  <value>::= type:String
5
6  <named_expression_statement>::= "expression" <name> "=" <expression_group>
7
8  <expression_group>::= <def_expr>   |
9                        <par_expr>   |
10                       <mult_expr>  |
11                       <named_expr> |
12                       <oit_expr>   |
```

```
13                          <neg_expr>   |
14                          <bin_expr>   |
15                          <in_expr>
16
17  <def_expr> ::= "defined" "(" <key> ")"
18
19  <par_expr> ::= "(" <expression_group> ")"
20
21  <mult_operator> ::= "&&" | "||"
22
23  <mult_expr> ::= <expression_group> <multiple_operator> <expression_group>
24
25  <named_expr> ::= <name>
26
27  <oit_expr> ::= "one_is_true" "(" 1*<expression_group> ")"
28
29  <neg_expr> ::= "!" <expression_group>
30
31  <bin_operator> ::= "==" | "!=" ; "equal" or "not equal" operators
32
33  <bin_expr> ::= <key> <bin_operator> <value>
34
35  <in_expr> ::= <in_string_expression_group> | <in_list_expression_group>
36
37  <in_string_expression_group>::= type:String "in" <key>
38
39  <in_list_expression_group>::= <key> "in" "[" <value_list> "]"
40
41  <value_list>::= 1*<value>
```

Listing 4.18: Named Expressions Statements grammar in BNF format

Three different string elements are used in NESes. `<name>` is an NE identifier, `<key>` refers to a VP name, and a VP variant's name is represented by `<value>` (lines 1–3 in Listing 4.18). Eight different constructs can be used as part of the boolean expressions (`expression group` in Listing 4.18, lines 8–15), their semantics are—informally—described as follow:

- **Define expression** (*def_expr*): Returns whether or not a value has been assigned to an SVP (e.g. Lines 1 and 13 in Listing 4.15).
- **Parenthesis expression** (*par_expr*): Add parenthesis to a sub-expression. It is used to enforce a precedence order in the expression evaluation (e.g. Lines 3–11 in Listing 4.16).
- **Multiple expression** (*mult_expr*): Defines a composed boolean expression using conjunction or disjuction. An `<expression group>` is used in the left- and right-hand side sub-expressions; a recursive definition (see Lines 13 and 17 in Listing 4.16, for an example).
- **Named expression** (*named_expr*): Reference to a previously defined boolean expression. Lines 17 - 27 in Listing 4.16 show NE as part of NEs statements.

- **One-is-true expression** (*oit_expr*): Returns true if at least one of the sub-expressions yields true.
- **Negation expression** (*neg_expr*): Logical negation of the value returned by the sub-expression.
- **Binary expression** (*bin_expr*): Checks the value assigned to a VP, using the equal and not equal comparison operators.
- **In expression** (*in_expr*): There are two types of in expressions: string and list.
  - **In-string expression** takes an arbitrary string and a VP name as inputs. It returns true when the input string is a sub-string of the value assigned to the specified VP, otherwise false.
  - **In-list expression** requires a VP reference and a list containing a sub-set of its associated VP values. It returns true if the value assigned to the referenced VP is in the given list.

Translating Named Expressions Statements as Clafer constraints

All the NESes found in the VPO model are decoded during the model transformation using a recursive implementation, which returns the equivalent Clafer constraint as a string. During the decoding process, when an expression is defined recursively (e.g. `<mult_expr>`, Listing 4.18 line 23), the recursive calls return the translated sub-expressions as a strings, then these are concatenated using the specified logic operator. There are non-recursive definitions (e.g. `in_expr` or `bin_expr`) that guarantee the recursion always ends. Once a NES is decoded, the result is stored in a hash table. This table is used later during the transformation when an NE is referenced, i.e. when it appears as a term in another boolean expression.

Next, it is provided an example for each element of the expression group. First, the original expression in VPO is shown, and then its equivalence in Clafer.

Define expression

The define-expression only applies to SVPs. It was decided to model undefined SVPs explicitly which makes the translation of this expression rather straightforward:

```
defined(:<SVP_name>)
```

translates to

```
[ !<SVP_name>.variant.SVP_UNDEFINED ]
```

Named expression

A named expression is a string key that refers to a previously defined boolean equation. All the NESes decoded during the model transformation process are stored in a hash table. When a named-expression is processed, the retrieved `<key>` string is used to perform a look-up in the hash-table, returning the associated Clafer constraint if found or throwing an error otherwise. Listing 4.16, lines 19–27 show examples of NE used as part of assignments.

Multiple expression

The multiple expression has a fixed structure with two sub-expressions related by a conjunction or a disjunction. Each sub-expression generates a recursive call; the returned strings are then concatenated using the "`&&`" or "`||`" operator.

```
MMT_IS_1950_org || MMT_IS_1960_org
```

translates to

```
[
    (SVP_MES_MACHINE_TYPE.variant.NXT1950AI) ||
    (SVP_MES_MACHINE_TYPE.variant.NXT1960BI)
]
```

One-is-true Expression

The elements of the list are decoded and concatenated using the `||` operator.

```
one_is_true(MMT_IS_SAWS,MMT_IS_1470)
```

becomes

```
[
( SVP_MES_MACHINE_TYPE.variant.SAWS ||
  SVP_MES_MACHINE_TYPE.variant.NXT1470D ||
  SVP_MES_MACHINE_TYPE.variant.NXT1470E)
]
```

Notice that the named expression that appears in the VPO expression (`MMT_IS_1470`) is translated as:

```
SVP_MES_MACHINE_TYPE.variant.NXT1470D || SVP_MES_MACHINE_TYPE.variant.NXT1470E
```

In the example for the *In-String* expression it will be explained how this is determined.

Negation Expression

The sub-expression is resolved recursively and the "!" is appended to the returned value.

```
!one_is_true(MMT_IS_SAWS,MMT_IS_1470)
```

is transformed to the constraint

```
[
!( SVP_MES_MACHINE_TYPE.variant.SAWS     ||
   SVP_MES_MACHINE_TYPE.variant.NXT1470D ||
   SVP_MES_MACHINE_TYPE.variant.NXT1470E)
]
```

Binary Expression

Binary expressions are transformed to Clafer constraint with a single proposition or its negation.

```
<key> == <value> and  <key> != <value>
```

are translated to Clafer as

```
<key>.variant.<value> and  !<key>.variant.<value>
```

Example:

```
:RETICLE_HEATING_WILDCARD == 'Y'
```

is equivalent to the Clafer constraint

```
[ SVP_RETICLE_HEATING_WILDCARD.variant.Y ]
```

In-String expression

The expression `<string> in <key>` is translated as:

```
<key>.variant.<value_1> || ... || <key>.variant.<value_n>
```

Example:

```
expression MMT_IS_1470_org = ("1470" in :MES_MACHINE_TYPE)
```

is transformed to

```
  SVP_MES_MACHINE_TYPE.variant.NXT1470D ||
  SVP_MES_MACHINE_TYPE.variant.NXT1470E
```

The input string `<string>` (`"1470"` in the example)is a substring of each variant's name in the set {`<value_1>`, ..., `<value_n>`}, which is the set of all machines types in the example above. If there is no variant name with a sub-string `<string>`, the decoding function returns *false* (i.e. `Boolean.is.false`). In the expression used as example, two machine types contained the substring "1470", NXT1470D and NXT1470E.

In-List expression

The expression <key> in [<value_##>+], where <value_##>+ represents a non-empty subset of variants of the variable <key>, is translated to a disjunction of propositions, a proposition for each variant of the input subset.

Example.

```
DOSEMAPPER in ['Y','S']
```

is equivalent to

```
SVP_DOSEMAPPER.variant.Y || SVP_DOSEMAPPER.variant.S
```

4.4.6 Assignment expressions

An *assignment expression* (AE) has the following structure:

```
<key> = <value> [guard]
```

Assignment Expressions are straightforwardly mapped to assignment constraints in Clafer. If the AE defines a guard condition, it appears as a guarded constraint expression in the output model.

An Assignment Expression without a guard.

```
<key> = <value>
```

is transformed to

```
[key.variant.value]
```

A guarded Assignment Expression.

```
<key> = <value> [<constraint_expression>]
```

is transformed to

```
[ <constraint_expression> => key.variant.value ]
```

The constraint expression used as a guard is decoded from `group_expression` in the VPO model.

If a VPO mapping contains a set of assignments affecting the same VP, implicit constraints must be considered. The ASML's CM tools evaluate assignment expressions in a mapping from top to bottom; the first condition met determines which assignment is executed.

Consider the PVP2EVP mapping in the Listing 4.19 below. It has three assignments, two guarded (lines 3–7) and an unguarded (line 9). If the guard condition on line 4 holds, the first assignment (line 3) is executed—ignoring the remaining assignments. If not, the second assignment is evaluated and so on; the last unguarded assignment (line 9) is applied if no guard condition holds. This ordered evaluation is preserved by adding extra propositions to the guarded expressions in the output model.

```
1  mapping PWASxEVPxCFG
2  {
3      PWASxEVPxCFG:PWAS_CH1_SS_COILS_THERMAL_CHAR = 'CHARACTERISTICS_SET_1'
4      [WPPAxVPxHW:WP_CH1_SS_ACTUATOR_VERSION == '1'];
5
6      PWASxEVPxCFG:PWAS_CH1_SS_COILS_THERMAL_CHAR = 'CHARACTERISTICS_SET_2'
7      [WPPAxVPxHW:WP_CH1_SS_ACTUATOR_VERSION == '2'];
8
9      PWASxEVPxCFG:PWAS_CH1_SS_COILS_THERMAL_CHAR = 'CHARACTERISTICS_SET_1';
10 }
```

Listing 4.19: Assignment expressions in a PVP2EVP mapping

The same mapping in Clafer is presented in Listing 4.20. The first assignment (lines 1–5) is translated directly from the input statement. The second assignment (lines 6–11) is executed only if the first guard condition does not hold and the second does. The third assignment (lines 12–17) occurs only if the two previous guard conditions yield false. All the constraints in the output model are evaluated (they cannot be "skipped"). The evaluation order is achieved by making the guard conditions mutually exclusive. Mutual exclusion requires to add extra negated propositions to the guard expressions—see lines 7, 13 and 14.

```
1  [
2      (WPPAxVPxHW_WP_CH1_SS_ACTUATOR_VERSION.variant.n1)
3      =>
4      PWASxEVPxCFG_PWAS_CH1_SS_COILS_THERMAL_CHAR.variant.CHARACTERISTICS_SET_1
5  ]
6  [
7      !(WPPAxVPxHW_WP_CH1_SS_ACTUATOR_VERSION.variant.n1) &&
8      (WPPAxVPxHW_WP_CH1_SS_ACTUATOR_VERSION.variant.n2)
9      =>
10     PWASxEVPxCFG_PWAS_CH1_SS_COILS_THERMAL_CHAR.variant.CHARACTERISTICS_SET_2
11 ]
12 [
13     !(WPPAxVPxHW_WP_CH1_SS_ACTUATOR_VERSION.variant.n1) &&
14     !(WPPAxVPxHW_WP_CH1_SS_ACTUATOR_VERSION.variant.n2)
15     =>
16     PWASxEVPxCFG_PWAS_CH1_SS_COILS_THERMAL_CHAR.variant.CHARACTERISTICS_SET_1
17 ]
```

Listing 4.20: Assignment expressions translated in Clafer

4.4.7 IF expressions

Hierarchical mappings use IF Expressions (IFEs)—an *if..then..else* construct. The *if* condition is an `<expression_group>` (see definition on lines 8–15 in Listing 4.18). Inside the *if* blocks, assignment expressions are found.

Consider the IFE shown in Listing 4.21. The *import* statement (line 2) makes all the named expressions defined in `MES_MACHINE_TYPE` available for this mapping; there is no need to translate this instruction since all the NEs are always available during the transformation.

Lines 6 and 7 define a local NES used on line 12. All the locally defined NESes are also stored during the model transformation because they might be referenced in another mapping.

```
1  <!--
2  import "MES_MACHINE_TYPE";
3
4  mapping KCRHEXxVPxCALIBxREUSE
5  {
6      expression HAS_RHRR = (defined(:RETICLE_HEATING_WILDCARD) &&
7                             (:RETICLE_HEATING_WILDCARD == 'Y'));
8      if (MMT_IS_SAWS)
9      {
10         KCRHEXxVPxCALIBxREUSE:RETICLE_HEATING_REC_REUSE = 'DISABLED';
11     }
12     else if (HAS_RHRR)
13     {
14         KCRHEXxVPxCALIBxREUSE:RETICLE_HEATING_REC_REUSE = 'MODE0';
15     }
16     else
17     {
18         KCRHEXxVPxCALIBxREUSE:RETICLE_HEATING_REC_REUSE = 'DISABLED';
19     }
20 }
21 -->
```

Listing 4.21: If-then-else construct used in a VPO mapping

Clafer's *if* construct cannot be used in this case, the *else-if* block (lines 12–15) is an *if* nested in the *else* block and the native Clafer's expression does not support nesting. Therefore, the *if-then-else* construct is transformed as a set of guarded assignments. The presented *if* example is an alternate representation of the list of guarded assignments presented earlier. The second assignment (line 14) is done only when the first *if* condition is false and the second is true. When no *if* condition holds, the assignment in the else block is effectuated. The resulting Clafer constraints are shown in Listing 4.22.

```
1  [
2      (SVP_MES_MACHINE_TYPE.variant.SAWS)
3      =>
4      KCRHEXxVPxCALIBxREUSE_RETICLE_HEATING_REC_REUSE.variant.DISABLED
5  ]
6
7  [
8      !(SVP_MES_MACHINE_TYPE.variant.SAWS) &&
9      (!SVP_RETICLE_HEATING_WILDCARD.variant.SVP_UNDEFINED &&
10     (SVP_RETICLE_HEATING_WILDCARD.variant.Y) )
11     =>
12     KCRHEXxVPxCALIBxREUSE_RETICLE_HEATING_REC_REUSE.variant.MODE0
13 ]
14
15 [
16     !(SVP_MES_MACHINE_TYPE.variant.SAWS) &&
17     !(
18        !SVP_RETICLE_HEATING_WILDCARD.variant.SVP_UNDEFINED &&
```

```
19          (SVP_RETICLE_HEATING_WILDCARD.variant.Y)
20       )
21      =>
22      KCRHEXxVPxCALIBxREUSE_RETICLE_HEATING_REC_REUSE.variant.DISABLED
23  ]
```

Listing 4.22: If-then-else construct translated to Clafer

IF expressions in VPO models support nesting and can be of an arbitrary size; these expressions are decoded recursively during the model transformation. Nested IF expressions prepend the condition specified in the containing *if* block. Listing 4.23 shows a template of a nested IF expression; tags are used to represent *if* conditions and variable assignments.

```
1  <!--
2  if (<cond_1.0>)
3  {
4      if( <cond_1.1> )
5      {
6          <assignment_1.1>;
7      }
8      else if ( <cond_1.2> )
9      {
10         <assignment_1.2>;
11     }
12     else
13     {
14         <assignment_1.3>;
15     }
16 }
17 else
18 {
19     if( cond_2.1 )
20         <assignment_2.1>;
21 }
22 -->
```

Listing 4.23: Nested If-then-else construct in VPO, generic example

The resulting Clafer constraints are provided in Listing 4.24—the tags are capitalized to indicate it is the same VPO expression translated to Clafer. The three guard constraint expressions in the first nested block prepend the outer *if* condition (<cond_1.0>) with a conjunction (lines 2, 8 and 14). Each assignment in a nested IF expression considers both inner and outer(s) *if* conditions. The last assignment prepends the outer else condition (i.e. the negation of the corresponding if condition) to the guard expression (line 20).

```
1  [
2      ( <COND_1.0> && <COND_1.1> )
3      =>
4      <ASSIGNMENT_1.1>
5  ]
6
7  [
8      ( <COND_1.0> && ( !<COND_1.1> && <COND_1.2> ) )
9      =>
```

```
10      <ASSIGNMENT_1.2>
11  ]
12
13  [
14      ( <COND_1.0> && ( !<COND_1.1> && !<COND_1.2> ) )
15      =>
16      <ASSIGNMENT_1.3>
17  ]
18
19  [
20      (!<COND_1.0> && <COND_2.1>)
21      =>
22      <ASSIGNMENT_2.1>
23  ]
```

Listing 4.24: Nested If-then-else construct translated to Clafer

The implemented transformation decode nested *if* statements with an arbitrary structure. Each time a (nested) IF expression is found, the decoding function is called recursively, passing on the guard constraint expression of the current block. When the guard expression is generated for an assignment in an nested *if* block, the passed constraint is added as a sub expression.

Internal variables defined within hierarchical mappings

VPO models for NXT machines also define internal variables within if expressions (See listing 4.25). Internal variables are defined by assignment expressions where the <key> is not a reference to an existing VP but a new variable. Internal variables, by convention, start with an underscore character. These internal variables are then used as part of constraints expressions throughout the model.

```
1  if (MMT_IS_2100)
2  {
3      _TPT = 'E';
4      _OVL = 'OVL8';
5      _REV = 'REV4_1';
6  }
7  else if (MMT_IS_2050)
8  {
9      _TPT = 'E';
10     _OVL = 'OVL7';
11     _REV = 'REV4';
12 }
13 else if (MMT_IS_2000 || HAS_SNEP_2000)
14 {
15     _TPT = 'STD_D';
16     _OVL = 'OVL6';
17     _REV = 'REV3_2';
18 }
19
```

```
20  .
21  .
22
23  else if (_TPT == 'STD_D')
24  {
25      if (HAS_PEP_2E)
26      {
27          _TPT = 'E';
28      }
29      else if (!HAS_TPT_C)
30      {
31          _TPT = 'D';
32      }
33      else
34      {
35          _TPT = 'C';
36      }
37  }
```

Listing 4.25: If-then-else with internal variable's assignment

Internal variables required additional decoding considerations because their variants are not explicitly defined in one place. Listing 4.25 shows the beginning and the end of an IF expression containing internal variables (_TPT, _OVL and _REV). The IF expression is decoded following the same procedure described previously, with the only difference that new internal variables and their values are stored during the transformation. When the parsing is done, a new concrete clafer with all the internal definitions is created (see Listing 4.26 for an example). Then, a set of guarded constraints derived from the if-then-else structure are written to the output model, ensuring the internal variables a set to a value.

```
1   MACHINE_TYPE_INTERNALS
2       xor TPT
3           E
4           STD_D
5           STD_C
6           B
7           PEP_A
8           A
9           D
10          C
11
12      xor REV
13          REV4_1
14          REV4
15          REV3_2
16          REV3_1
17          REV3
18          REV2
19          REV1_1
20          REV1
21
22      xor OVL
23          OVL8
```

```
24          OVL7
25          OVL6
26          OVL4
27          OVL5
28          OVL3
29          OVL2
30          OVL2A
31          OVL1
32  }
```

Listing 4.26: Internal variables modelled in Clafer

Internal variables can be re-assigned in the SVP2PVP mappings. In Listing 4.25 line 27 `_TPT` should have the value `'STD_D'` (see condition on line 23) and is reassigned to `'E'`. These reassignments are problematic in the Clafer model because they lead to unsatisfiable constraints. The `_TPT` re-assignment gets translated as the following guarded constraint (only the relevant part of the LHS expression is shown):

```
[
  ... & MACHINE_TYPE_INTERNALS.TPT.STD_D & !HAS_PEP_2E)
  =>
  MACHINE_TYPE_INTERNALS.TPT.E
]
```

The `_TPT` variable needs to be assigned to a single value, which means that the constraint above is satisfied by assigning `'E'` to `_TPT` variable and making the proposition `MACHINE_TYPE_INTERNALS.TPT.STD_D` false. However, `_TPT` was set to `STD_D` in the first place by the following constraint:

```
[
    ( (!SVP_MES_MACHINE_TYPE.variant.NXT2050EI) && SVP_MES_MACHINE_TYPE.
       variant.NXT2000DI ) || SVP_SNEP2000.variant.Y) )
    =>
    MACHINE_TYPE_INTERNALS.TPT.STD_D
]
```

If the right-hand side of the implication is *false*, the left-hand side must yield *false* as well. If either `SVP_MES_MACHINE_TYPE.variant.NXT2000DI` or `SVP_SNEP2000.variant.Y` is false, the lhs of the expression is *false*. This meaning that either the machine type is different from `NXT2000DI` or the the `SNEP2000` is not selected (i.e. it is set to `'N'`).

All the SVP values are directly read from the machine configuration file and translated as setting constraints. Consider the following setting:

```
[ SVP_MES_MACHINE_TYPE.variant.NXT2000DI ]
[ SVP_SNEP2000.variant.Y ]
```

The above constraints select the machine type NXT2000DI and include the SNEP2000 feature. The TPT variable re-assignment constraint required that either NXT2000DI or SNEP2000 are not selected, while the machine configuration is selecting both. These contradictory requirements make the constraint problem unsatisfiable; therefore, there is not a single valid instance that can be generated from this model.

When a re-assignment is found, the guard conditions of both the first and the second assignment need to be re-written to make them mutually exclusive. Alternatively, a new fresh variable can be introduced for each re-assignment. Neither of these strategies were implemented as part of this work. This far-from-trivial functionality was deferred in the first iterations of the model transformation because it does not have a large impact, only a few variable's reassignments were found in the NXT_WET machine. As it was found later, the Clafer tools could not handle all the constraints obtained from model, so, adding more constraints would not provide useful information for our analysis using the available tools. In the final version of the model transformation, internal variables reassignments were removed from the output model.

4.4.8 Default values as guarded constraints in Clafer

A naive approach to set default values in Clafer might be to use guarded constraints inside the VP definition (see Listing 4.27, lines 13–19). The left-hand side of the implication specifies a machine type variant; the right-hand side sets the default value in the current VP definition. Since only one machine variant can be selected, this mapping is consistent.

```
1
2  WHPUxVPxCOSY_WFR_TEMP_COND_SYSTEM: PROTECTED_PVP_DEFINE
3      [ VPi_name = "WHPUxVPxCOSY" ]
4      [ VP_name = "WFR_TEMP_COND_SYSTEM" ]
5      [ Description = "WH temperature conditioning type" ]
6
7      xor variant
8          CONVENTIONAL
9          TYPE_1
10         TYPE_2
11         TYPE_4
12
13     [ SVP_MES_MACHINE_TYPE.variant.NXT2000DI => TYPE_2 ]
14     [ SVP_MES_MACHINE_TYPE.variant.NXT2050EI => TYPE_4 ]
15     [ SVP_MES_MACHINE_TYPE.variant.NXT1965CI => TYPE_1 ]
16     [ SVP_MES_MACHINE_TYPE.variant.NXT1980DI => TYPE_2 ]
17     [ SVP_MES_MACHINE_TYPE.variant.NXT1950AI => TYPE_1 ]
18     [ SVP_MES_MACHINE_TYPE.variant.NXT1960BI => TYPE_1 ]
19     [ SVP_MES_MACHINE_TYPE.variant.NXT1970CI => TYPE_2 ]
```

Listing 4.27: Machine defaults expressed as constraints in Clafer

However, the PVP defined in Listing 4.27 is assigned to a value as part of a SVP2PVP mapping. One constraint of that mapping is shown in Listing 4.28. Assume a particular machine configuration in which NXT2000DI machine type is selected, the default constraints set the value TYPE_2 for the current example (see Listing 4.27 line 13). In the same configuration, the values for the internal variables (MACHINE_TYPE_INTERNALS) are such that the lhs of the guarded constraint in Listing 4.28 yields true. The mapping constraint sets the PVP to the value TYPE_4. This pair of constraints are equivalent to the internal variable reassignment, and leads to unsatisfiability.

```
1 [
2     ((MACHINE_TYPE_INTERNALS.OVL.OVL7 || MACHINE_TYPE_INTERNALS.OVL.OVL8 ) &&
3     !SVP_MES_MACHINE_TYPE.variant.SAWS)
4     =>
5     WHPUxVPxCOSY_WFR_TEMP_COND_SYSTEM.variant.TYPE_4
6 ]
```

Listing 4.28: A SVP2PVP constraint setting a variable with a default value also defined.

As a result, this approach for setting default values does not work. It would be necessary to include setting constraints for default values *only if* the variable does not have a value already; there is not construct in the language to determine that without using the backend solvers.

4.5 Model transformation results

The number of VPs (divided by category) and constraints translated in the model transformation is shown in Table 4.1. NXT machines have more years in the market and therefore, their variability model is considerably larger compared with the latest models (NXE and EXE).

	Machine type			
Decoded items	**NXT_WET**	**NXT_DRY**	**NXE**	**EXE**
SVPs	177	175	44	45
PVPs	708	704	573	569
EVPs	1707	1694	1295	1305
VP interface constraints	107	107	60	61
SVP2PVP constraints	725	721	356	326
PVP2EVP constraints	4113	4069	2613	2443

Table 4.1: *VPs and constraints decoded from the VPO2Clafer transformation*

The Clafer toolset could not handle ASML's variability models. Once all the VPO constraints were included in the model transformation, compilation time increased to unpractical levels. While investigating the issue, it was found that the more variables a constraint includes, the more time it takes to compile it; i.e. multi-variable constraint expressions increased compilation time drastically.

Michał Antkiewicz—one of the main Clafer developers—helped brainstorming the issue in an exchange of several emails from which two ideas came out. The first one is to ensure that all the elements in the model have unique names and skip the variable name resolution check—compiler option `--skip-resolver`. This should speed up the compilation process. The second is to profile the Clafer compiler to find the bottleneck and fix it; a more elaborate and time consuming endeavor.

Only the first option was implemented; the variability model for the EXE machine was compiled in 4 days, 21 hours and 31 minutes (!). This option did not fix the issue for the remaining larger models, which were left compiling for a couple of weeks until the process was interrupted.

The compiled Clafer model for the EXE machine was used to generate model instances. This time the process ran for 15 days with no results. The constraint problem is apparently too large and complex for the tools to manage it efficiently. It is possible that some unsatisfiable constraints in the model caused the instance generator to not find a single solution in all this time. However, given the time that it took to compile these models it would be unfeasible to troubleshoot a potential issue such as this.

The Clafer instance generator was tested using the `NXT_WET` model containing only the VP interface constraints but not the hierarchical mapping constraints. The model was compiled and let run for 45 hrs. 149,888 configurations were found during that time, showing that Clafer tools are able to generate instances when fewer constraints are included.

Importantly, we found out during conversations with configuration management domain experts that the current variability models have missing constraints. Hence, invalid configurations can be derived from them. When a machine has a configuration that yields to a failure, ASML engineers rely on manual procedures to fix it. This is one of the most relevant issues the company is interested in addressing.

4.6 Modelling SMDC specification using Clafer

In order to evaluate the structural and variability modelling capabilities of Clafer, the System Manager Driver Control (SMDC) component was selected. The SMDC is a TWINSCAN software component that specifies the initialization and termination of all drivers; the information of which is contained in a SMDC configuration file. SMDC files describe the drivers applicable for a specific machine configuration and the dependencies among them.

SMDC files contain variation points based on VP-values. Drivers' definitions can be conditionally included based on the VP values defined in the machine configuration. The description of the SMDC files presented in this section, including the fictitious examples used to illustrate the SMDC constructs, are taken from ASML's EPS System Manager Driver Control document [82].

We use Clafer to model both the SMDC specification and its variability. Using the SMDC specification, it is evaluated to what extent Clafer can integrate structural modelling and variability using a real scenario. Unlike for ASML's variability model in Sections 4.3 and 4.4, we only provide a conceptual mapping between SMDC and Clafer constructs; the SMDC to Clafer model transformation remains as future work.

The presented section is divided into three subsections. Subsection 4.6.1 introduces the structure of a SMDC configuration and the constructs used in it; it also provides a minimal

SMDC configuration example. Subsection 4.6.2 shows how these constructs can be modelled in Clafer. Finally, Subsection 4.6.3 provides final remarks about the information that can be obtained from SMDC configurations using the Clafer toolset.

4.6.1 SMDC configuration file

Every SMDC file contains the following five sections:

- **Driver definitions**: Define all the (non-virtual) drivers defined for this platform (we call these *individual* drivers).
- **Virtual definitions**: A virtual driver comprises one or more individual driver's definitions.
- **Operational dependencies**: Define dependencies between drivers during normal operation.
- **Initialize dependencies**: Define initialization dependencies between drivers.
- **Re-initialize dependencies**: A more restrictive dependency compared to operational and initialize dependencies. Re-initialize dependencies determine whether or not a driver needs to be re-initialized after another driver is initialized or terminated.

Each individual driver definition has two mandatory fields: a *name* and a *description*. Optionally, the driver can be split into an arbitrary number of phases, each of which is identified by a name and separated from each other by a comma. Both the driver description and its phases are string names enclosed in double quotes. The rationale behind dividing a driver initialization into phases is to specify dependencies between these phases rather than between the whole initialization, enabling fast startup of the machine.

Driver definitions might be guarded using an if construct. The driver definition is included or removed depending on the VP value specified in the guard condition, we call these *inclusion conditions*.

Each driver in the system has two states: *terminated* (default state) and *initialized*, the SMDC module is responsible for initializing and terminating all the drivers in the system. The transition between *terminated* and *initialized* states, and the transition between *initialized* and *terminated* states have predefined *timeout* values. In a SMDC configuration file, it is possible to update predefined initialization and termination timeout values for any driver.

A virtual driver comprises one or more drivers, however, it behaves as an individual driver for other systems. When a virtual driver is initialized, all their referenced drivers are initialized. Likewise, terminating a virtual driver results in the termination of all its individual drivers.

As indicated by the bullet list above, there are three different kind of dependencies between drivers: *initialize*, *operational* and *reinitialize* dependencies. Let us consider the dependency

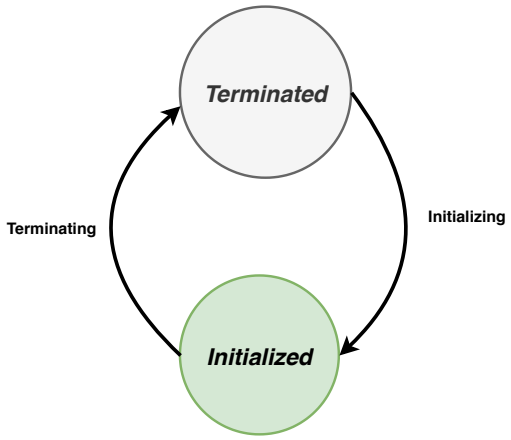

Figure 4.5: *SMDC driver states*

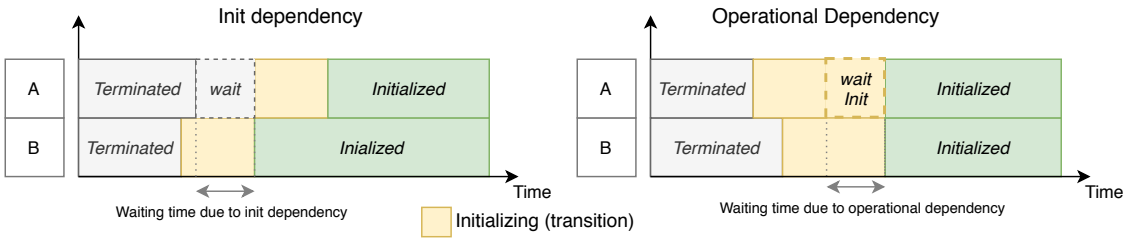

Figure 4.6: *Initialization dependencies (left) and operational dependencies (right)*

A *depends on* **B** for two arbitrary drivers **A** and **B**. An *initialize* dependency means that **A** can start initialization only after **B** has been initialized (Fig. 4.6 left). Operational dependencies are slightly different, driver **A** can reach the *initialization* state only after **B** has been *initialized* (Fig. 4.6 right). Under certain scenarios, drivers are forced to delay initialization or prolong the initialization phase due to *initialize* and *operational* constraints respectively.

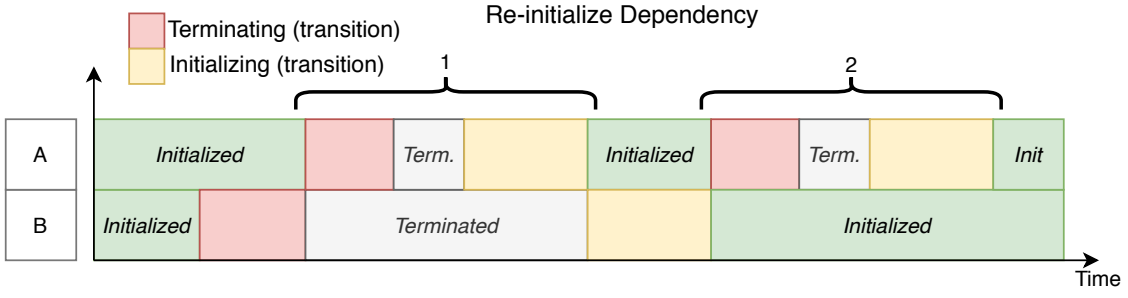

Figure 4.7: *Re-init dependencies. 1) Reinit on termination, 2) Reinit on initialization*

Re-init dependencies are more complex. Driver **A** should be re-initialized if driver **B** is initialized or terminated. The initialization sequence for both scenarios is shown in Fig. 4.7.

When a driver initialization is divided into phases, dependencies can be specified among them rather than on the whole driver initialization. This strategy helps improve the initialization speed by minimizing the waiting time due to dependencies.

```
1  DRIVER_DEFINITIONS
2      DEFINE: AM "Air Mounts"
3      DEFINE: WH "Wafer Handler"
4
5      # Driver IR is only conditionally available.
6      if INT_RETICLE_INSP_SYSTEM == PRESENT
7          DEFINE: IR "Int. Rtcl Inspect System"
8      endif
9
10     DEFINE: AC "Alignment Scan Control" 300
11     DEFINE: AF "Alignment Subsystem" 300 600
12
13     DEFINE: WS "Wafer Stage"
14     "phase_A" phase_B" 90, "phase_C" 30 60
15
16 VIRTUAL_DEFINITIONS
17     WW "Wafer subsystem" : WH, WS
18
19 OPER_DEPENDENCIES
20     AC: AF, AM
21     WS: WH
22
23 INIT_DEPENDENCIES
24     AM: AF
25     WS: WH
26     WS "phase_B": IR
27     WS "phase_C": AF
28
29 REINIT_DEPENDENCIES
30     WS: AC
```

Listing 4.29: Example SMDC specification

A minimal example of a SMDC file, including all the elements previously described, is shown in Listing 4.29. Driver definitions are included in lines (1–14), simple driver definitions (i.e. those containing only mandatory information) are shown in lines 2, 3 and 7. The definition in line 7 is included only when `INT_RETICLE_INST_SYSTEM` (a Variability Parameter) has the value `PRESENT`—an inclusion condition. Definitions in lines 10, 11 and 13–14 define new timeout values. *AC* (line 10) and *AF* (line 11) drivers define an initialization timeout of 300 seconds. Additionally, *AF* specifies a termination timeout value of 600 seconds. Driver `WS` definition in lines 13–14 is split into *phase_A*, *phase_B* and *phase_C*. An initialization timeout value is defined for *phase_B* while both initialization and termination timeouts are specified for *phase_C*.

A virtual driver definition that includes drivers `WH` and `WS` is shown in lines 16–17.

A set of operational dependencies (lines 19–21), init dependencies (lines 23–27) and reinit

dependencies (lines 29–30) are included in Listing 4.29 as well. All dependencies have the same syntactic structure, the dependent driver is followed by a colon and then a list of the drivers it depends on, separated by commas. When a driver is split into phases, dependencies can be specified on them. For instance, init dependencies are specified in all the`WS` driver phases. Line 25 specified a dependency on *phase_A* (when the phase name is omitted, the first phase is implicitly chosen). Dependencies on *phase_B* and *phase_C* are defined in lines 26 and 27.

4.6.2 Modelling the SMDC specification in Clafer

Before building a driver definition, we define the three abstract clafers shown in Listing 4.30. Lines 1–3 model the *timeout* values as an abstract clafer that contains an integer variable (line 2) and a constraint (line 3); this constraint reduces the domain of variable `timeout` to positive numbers only. It is important to note that the constraint uses the keyword `this`, which refers to the containing object. Since the constraint is nested under the variable definition, `this` refers to the variable `timeout`. Using relational logic, timeout values must be greater than zero.

The abstract driver definition is specified in the `DRIVER_DEFINE` in lines 5–6. Only two mandatory elements are related to the driver definition: its name (implicitly specified by the concrete clafer's name when this abstract definition is instantiated) and a description (line 6). Finally, a driver phase definition is shown in line 8. `DRIVER_phase` implicitly defines the phase name during instantiation as well.

```
1 abstract Timeout
2     timeout -> integer
3         [ this > 0 ]
4
5 abstract DRIVER_DEFINE
6         Desc -> string
7
8 abstract DRIVER_phase
```

Listing 4.30: Abstract clafers to model SMDC driver definition

The `WS` definition was taken from Listing 4.29 lines 13–14, and their dependencies from lines 21, 25–27 and 30 are used as an example. This driver was chosen because it incorporates different phases, new timeout values and all dependency types.

Listing 4.31 shows the `ABS_WS` driver definition modelled in Clafer. The description is set in line 2. *Phase_A* (line 4), *phase_B* (line 10) and *phase_C* (line 19) are instantiated from the `DRIVER_phase` abstract clafer. *Phase_B* defines an init timeout in lines 11–12. Init and terminate timeout values for *phase_C* are defined in lines 20–21 and 23–24, respectively. Timeout clafers are first instantiated and then a constraint is used to set their values.

All dependencies are modelled as references using the -> operator. A reference is a pointer to any existing clafer definition. The three `WS` init dependencies are: a dependency on `ABS_WH`

in line 7, on ABS_IR in line 16–17 and on ABS_AF in line 26. The driver ABS_IR is optionally defined, thus, the dependency on this driver has an inclusion condition. The implementation of an inclusion condition has two components: first, the reference declaration is made optional using the **?** operator (line 16); second, a constraint relates the condition to the existence of the reference using a bidirectional implication (line 17). When the condition yields true, the optional reference (id02) is included.

```
1  abstract ABS_WS : DRIVER_DEFINE
2          [Desc = "Wafer Stage"]
3
4          phase_A: DRIVER_phase
5              INIT_DEPS
6                  /*Init dependency WS: WH*/
7                  id01-> ABS_WH
8
9
10         phase_B: DRIVER_phase
11         phase_B_init: Timeout
12                 [phase_B_init.timeout = 90]
13
14         INIT_DEPS
15             /*Init dependency WS "phase_B": IR*/
16             id02 -> ABS_IR ?
17                 [INT_RETICLE_INSP_SYSTEM.variants.PRESENT <=> id02]
18
19         phase_C: DRIVER_phase
20         phase_C_init:Timeout
21                 [phase_C_init.timeout = 30]
22
23         phase_C_end:Timeout
24                 [phase_C_end.timeout = 60]
25
26         OPER_DEPS
27             od01-> ABS_WH
28
29         REINIT_DEPS
30             rd01-> ABS_AC
```

Listing 4.31: Driver definition in Clafer. It contains three phases and all the kind of dependencies

Similarly, operational and reinit dependencies are defined in lines 26–27 and 29–30, respectively. This approach uses hierarchical decomposition to associate dependencies to each driver definition or its individual phases; all the dependency information is encoded inside driver definitions in this way.

Listing 4.32 shows DRIVER_DEFINITIONS where all the specified abstract driver definitions are instantiated inside. Each abstract clafer defined in the minimal SMDC example is instantiated to a concrete clafer. Note that the IR definition is optional, so, an inclusion condition is used (lines 5–6).

```
1 DRIVER_DEFINITIONS
2         AM:ABS_AM
3         WH:ABS_WH
4         AF:ABS_IR ?
5         [INT_RETICLE_INSP_SYSTEM.variants.PRESENT <=> IR]
6         AC:ABS_AC
7         AF:ABS_AF
8         WS:ABS_WS
```

Listing 4.32: Instantiation of abstract driver definitions

The virtual driver definition in the SMDC example can be represented in Clafer as shown in Listing 4.33.

```
1 abstract ABS_VIRTUAL : VIRTUAL_DRIVER_DEFINE
2         [Desc = "Wafer subsystem"]
3         REF_WH -> DRIVER_DEFINITIONS.WH
4         REF_WS -> DRIVER_DEFINITIONS.WS
```

Listing 4.33: Virtual Driver abstract definition

As for the individual driver definitions, virtual driver (abstract) definitions can be instantiated in a clafer constainer as shown in Listing 4.34.

```
1 VIRTUAL_DRIVER_DEFINITIONS
2         VIRTUAL:ABS_VIRTUAL
```

Listing 4.34: Virtual Driver instantiation

An more extensive example of an SMDC specification and its translation to Clafer is shown in Appendix A.

4.6.3 Analysis of SMDC specifications in Clafer

The presented Clafer model encodes all the driver's information contained in a SMDC file with the inclusion conditions based on VP-values. If the information about the VPs (included in the SMDC file) and their dependencies are part of the Clafer model, the Clafer instance generator derives all the possible combinations of VP values that satisfy such dependencies. Based on the different VP-values and the inclusion conditions, all the possible SMDC file versions can be generated.

Driver states (*initialized* and *terminated*) and transitions between them could be also modelled in the Clafer model. Then, driver dependencies could be expressed as constraints instead of as references. For instance, an init dependency can be easily modelled using a logical implication: the initialization transition of the dependent driver on the left-hand side and the initialized state of the driver it depends upon on the right-hand side. If each driver is linked to a state, and its dependencies are constraints between them, the instance generator can find all the valid combinations of driver states for each SMDC file version.

Unfortunately, reinit dependencies cannot be expressed using Clafer constraints. Reinit dependencies require specifying a set of state transitions in the dependent driver and propositional logic cannot be used to express such dynamic behaviour. This results on some instance containing an invalid combination of driver's states due to missing reinit dependencies. Modelling driver states explicitly would be useful only if we can verify properties using a temporal logic. It would be extremely interesting to attempt to verify behavioral properties in all the different variants of an SMDC file and determine if there are some variants which violate one

The possibility of generating all the valid SMDC file instances using Clafer can be used to uncover inconsistencies due to changes in inclusion conditions or in the variability model. For instance, circular dependencies (A depends on B, and B depends on A) or dependencies to nonexistent drivers (derived from incorrect inclusion conditions).

4.7 Clafer missing constructs

Three VPO language constructs were missing in Clafer: Boolean variables, boolean literals and default values. The lack of boolean literals was solved by modelling them as an XOR group with a constraint selecting the *true* literal. In VPO models, the result of a boolean expression is stored in boolean variables (known as named expressions). Boolean variables make constraint expressions more concise; sub-expressions can be stored in variables and then used as part of a larger boolean equation. Because Clafer does not support boolean variables, translated named expressions were stored during the model transformation, and then, each variable use was replaced with its definition. The constraints in the output model are equivalent to those in the VPO files, but sometimes quite large and thus, less readable. The lack of Boolean variables and literals does not represent an issue for the model transformation; the information represented by these constructs was successfully translated to the target model.

On the other hand, the default values cannot be represented in Clafer directly. Two strategies were considered during the development of this project as indirect support for default values. The first strategy consists of generating an initial version of the variability model without default values. Then, use the Clafer instance generator to obtain all the valid model instances, from which the under-constrained VPs can be identified. Knowing the under-constrained set of variables, extra constraints are inserted into the initial model to assign default values to them. A similar strategy to this is what is currently implemented at ASML: when undefined VPs are detected in a—faulty—machine configuration, the configuration management module throws an error, prompting the user to run the `asm_upgrade_config` tool to assign default values to the undefined variables. A second strategy consists of evaluating the constraints during the model transformation to identify under-constrained VPs and then insert additional constraints to set default values where needed. This second alternative is more convenient from the user's perspective—the default values are assigned automatically without further steps—although the implementation of this strategy is far from trivial. VPO constraints only require propositional logic, and most of the VP variable assignments occurs either in single assignment constraints or on the right-hand side of an implication operator inside a guarded assignment constraint. Thus, this implementation may be feasible, but is left as future work.

4.8 Clafer's toolset evaluation

The Clafer tools deal with small to medium–size models properly, but they are not efficient enough to manage large models including many multi-variable constraint expressions. When all the VPs and constraints available from the VPO were included in the Clafer model, compilation time and instance generation time skyrocketed.

The Clafer toolset does not translate the counterexamples and conflicting constraints generated by the Choco-solver[1] back to Clafer. This is a crucial feature when troubleshooting large models. A conflicting constraint makes the constraint problem unsatisfiable and no instances are generated from the model. These problematic constraints are relatively easy to find manually on small models, but it is an insurmountable endeavor when dealing with large models like those at ASML.

Four valuable characteristics were found in Clafer tools. First, the toolset is freely available to run online; this is quite useful for rapid prototyping. Second, tool usage and implementation is well documented in the Clafer wikipage—which also has plenty of model examples. Third, Clafer's developers were quite responsive and open to answering questions—even thought the project is officially finished and they are not working on its development anymore. Lastly, the project is open source and has a permissive license; any company can use the source code with no obligation of sharing the modified code.

Based on the results obtained from this work, it is concluded that Clafer toolset is not robust enough to handle the variability model at ASML. Using it would require optimizing the compiler and probably integrating a SAT solver to analyze the ASML models in a reasonable time. Translating all the debugging information provided by the backend solvers—something that is not currently implemented—is important in order to facilitate the debugging process of variability models.

[1] the backend solver used in this study

4.9 Conclusions

In this chapter we implemented a model transformation from ASML's variability models to Clafer, and provided a conceptual mapping from SMDC specification to Clafer. ASML's variability model is composed of a large set of Variability Parameters (SVPs, PVPs and EVPs) with constraints and dependencies. All VP attributes are either *string* or *integer* type, which are directly supported in Clafer. Four abstract clafers were used to model SVP, protected PVP, non-protected PVP and EVP types. From these abstract objects, all the VPs were generated, using an *XOR* group to define the variants associated to each VP. Therefore, it was found that Clafer is expressive enough to represent all the variability constructs in the VPO models (RQ2).

Default values support is missing in Clafer—other constructs like boolean variables and literals are indirectly supported. As it is, the lack of default values support is the weakest point of this variability language in the context of ASML. It is important to note that this is a missing configuration support capability (as already identified in Chapter 2, Table 2.1) and not a variability expressiveness issue.

A comprehensive evaluation of a variability modelling language demands robust tools that enable model analysis to derive useful information. A variability language's suitability cannot be determined only by its capacity to express variability of a domain. It must also provide insights about the system by performing computer-aided analysis (based on the variability specification) that would otherwise be impossible—or quite difficult—to be derived manually.

The Clafer compiler and instance generator were used to analyze the output models. The Clafer toolset handles large models without failures, but it turned out to be extremely slow for our use case. Skipping the name resolution step in the compiler helps speed up the process, but a more comprehensive analysis—such as compiler's profiling—is required for further optimization. Thus, the tools' speed is the most important limiting factor that makes them not robust enough for their use in ASML's context (RQ3).

Clafer's constraint language is very expressive, supporting propositional, relational, arithmetic and first-order logic. For ASML's variability model, propositional logic suffices. Under certain scenarios, SAT solvers deal with propositional satisfiability problems more efficiently than more advanced CSP solvers [83], such as those used by the Clafer toolset as backends. An interesting enhancement of Clafer tools would be to add a SAT solver that is optionally used when the analyzed models contain propositional logic only.

Clafer claims to be expressive enough to integrate structural modelling with variability. This aspect was explored in Section 4.6 by modelling the System Manager Driver Control specification in Clafer. The Clafer language can be used to represent all the elements in a SMDC file and integrate the variability information as inclusion conditions. The instance generator can be used to find all the possible variations of an SMDC file based on its inclusion conditions. The file variations found are then used to identify structural inconsistencies in the original file.

The analysis of this exercise proves that Clafer 0.4.5 is not fully useful in the ASML context. Even though it was found this is not a suitable language, this exercise's relevance lies in the identification of issues in ASML's current models. For instance, domain experts consulted pointed out that the current variability models have missing constraints, making it possible for invalid machine configurations to be derived. One approach to tackling this issue could be to generate all the possible configurations from the current variability model; then, using these configurations, use ASML's simulators to detect the invalid ones. Another approach, could be to use both behavioural and variability modelling to verify the behavioural properties among all the possible configurations of a system. All the configurations that violate those behavioural properties would be marked invalid. The invalid configurations detected using any of these approaches, could then be a starting point to adding some of the missing constraints to the variability model.

This chapter presented the last component of our evaluation proposal to determine whether or not the selected VML, could be used at ASML. The evaluation result is negative due to missing configuration support and lack of tool robustness. However, our evaluation results point to alternative strategies for configuration support and specific tool enhancements that would make the proposed VML applicable for ASML.

Chapter 5

Conclusions

A fundamental question in variability management—addressed by both academia and industry—is *how to represent variability*. Many ideas on the subject have been developed in academia over the last 30 years. At the same time, variability management has become an increasingly complex problem in the industry. Academia takes on this challenge by generating a myriad of new ideas and methodologies. Companies tackle variability management by developing customized processes and tools that evolve as their specific variability requirements change. Looking beyond their specific needs and incorporating academic ideas is often experienced as disruptive or risky to implement.

In this context, our goal (RQ1) was to conduct an effective evaluation of the existing VMLs in three stages: 1) a literature review, 2) a formal framework to evaluate variability language expressiveness and 3) a practical assessment of one VML using ASML variability models. This approach proved effective in evaluating existing VMLS; we argue its usefulness as follows.

The practical evaluation component of our proposal answers the question: *can an existing VML represent all the constructs in our variability models?* Using these models a definite answer is effectively reached, an answer that is only valid for a particular domain in a specific point in time. Our conclusion was that Clafer is expressive enough to represent variability constructs and constraints, but the current lack of default values support—a configuration capability—must be integrated into the language to fully express all the constructs found in ASML's variability models (RQ2). Furthermore, this practical assessment also showed that the evaluated Clafer toolset (0.4.5) is not robust enough to efficiently deal with the number of multi-variable constraints in the ASML variability models used in our evaluation (RQ3).

The systematic literature review component in our proposal yields a set of VML capabilities considered relevant based on academic research and an initial tool support assessment. This approach provides a reference model that can be used to reduce the VML options based on our requirements. In our assessment, it yielded Clafer and IVML as the two most promising options. Moreover, as a reference model, our approach has the added benefit that it also presents variability language capabilities that were not being considered as part of the re-

quirements of our domain, prompting the question: *are we following the best approach to model variability?*

As we consider the challenge of variability representation from a wider perspective, it becomes inevitable to consider the matter in more depth and ask ourselves: *what are all the forms of variability existing in the real-world that we are trying to represent?* The question of *what exist in the world* is the subject of ontologies, hence the inclusion of ontological expressiveness theories as part of our evaluation proposal. Ontological analysis pointed to the existence of the dynamic dimension of variability; a paradigm shift from the static view of variability modelling prevailing in most VMLs. With this shifted perspective, we can look back to our domain-specific variability requirements and recognize new possibilities for addressing our current challenges and foresee new paths forward.

Future work

Both Asadi's et. al evaluation framework and the Clafer modelling language point to the same aspect: the dynamic dimension of variability. ATFV does it by proposing process variability and Clafer by adding a temporal dimension to the language, yet in both cases, the applicability of this work is not clearly defined. The current ASML variability model allows the derivation of invalid configurations; so the possibility of verifying dynamic properties among all the variations of a specification would enable a systemic way of detecting these invalid configurations. This is what the results of our evaluation point to, but it is a possibility that requires further research and tools development in order to be validated.

Appendix A. Example SMDC file modelled in Clafer

This appendix includes an imaginary SMDC specification file (Listing 5.1). This specification file is translated to a Clafer model shown Listing 5.2.

```
1  # SMDC imaginary example file
2
3  DRIVER_DEFINITIONS
4      DEFINE: AC "Alignment Scan Control" 300 # Initialize timeout of 5
5      # minutes, default timeout
6      # (30 seconds) for terminate.
7      DEFINE: AF "Alignment Subsystem" 300 600 # Initialize timeout 5 min,
8      # terminate timeout 10 min.
9      DEFINE: AM "Air Mounts"
10     DEFINE: ID "Image Sensor Scan Control"
11     DEFINE: IF "Image Sensor Subsystem"
12     DEFINE: KS "Swap Control"
13     DEFINE: MI "M. System Interferometers"
14
15     # Drivers IR, PS and RV are only conditionally available.
16     if INT_RETICLE_INSP_SYSTEM == PRESENT
17     DEFINE: IR "Int. Rtcl Inspect System"
18     DEFINE: PS "Rtcl Particle Scanner"
19     DEFINE: RV "Reticle Mover"
20     endif
21     DEFINE: RH "Reticle Handling"
22     DEFINE: SNM "Synchronisation Control"
23     DEFINE: SOM "Synchronisation Driver (T) M"
24     DEFINE: WH "Wafer Handler"
25     DEFINE: WS "Wafer Stage"
26     "phase_A" 20, phase_B" 90, "phase_C" 30 60 #term timeout on C
27
28 VIRTUAL_DEFINITIONS
29     WW "ww" : WH, WS
30
31 OPER_DEPENDENCIES
32     AC: AF, SNM, WS
33     AF: MI, SOM
34
```

```
35 INIT_DEPENDENCIES
36     ID: IF
37
38     # Driver RS may only start if:
39     # Drivers AM and MI have already been initialized.
40     RS: AM, MI
41
42     # Driver WS has been initialized if:
43     # Steps "phase_A", "phase_B" and "phase_C" are initialized.
44     # Step "phase_B" may be init'ed after MI and phase_A have been initialized
45     # Step "phase_C" may be init'ed after AM and phase_B have been initialized
46     WS "phase_B": MI
47     WS "phase_C": AM
48
49 REINIT_DEPENDENCIES
50     ID: IF 19
51     KS: WS "phase_B"
```

Listing 5.1: Example SMDC specification

```
1  /*
2   *  Clafer instance generator found
3   *  two different instances of this model.
4   */
5
6  /* Abstract clafers for SMDC drivers definitions */
7
8  abstract Timeout
9      timeout -> integer
10         [ this > 0 ]
11
12
13 abstract DRIVER_DEFINE
14         Desc -> string
15
16
17 abstract DRIVER_phase
18
19 abstract VIRTUAL_DRIVER_DEFINE
20     Desc -> string
21
22
23 /*Abstract Clafer for VP definitions*/
24 abstract VP_DEFINE
25         VP_name -> string
26         Description -> string
27
28
29 /*VP definitions*/
30 INT_RETICLE_INSP_SYSTEM: VP_DEFINE
31         [VP_name = "Reticle"]
32         [Description = "Internal reticle inspection system"]
33
34         xor variants
```

```
35                  PRESENT
36                  ABSENT
37
38  /*
39   *   Driver definitions
40   */
41
42  abstract ABS_AC : DRIVER_DEFINE
43          [Desc = "Alignment Scan Control"]
44
45          init: Timeout
46                  [init.timeout = 300]
47
48          OPER_DEPS
49                  OD01 -> ABS_AF
50                  OD02 -> ABS_SNM
51                  OD03 -> ABS_WS
52
53          INIT_DEPS       /*NONE*/
54          REINIT_DEPS /*NONE*/
55
56
57  abstract ABS_AF : DRIVER_DEFINE
58          [Desc = "Alignment Subsystem"]
59
60          init: Timeout
61                  [init.timeout = 300]
62
63          end: Timeout
64                  [end.timeout = 600]
65
66          OPER_DEPS
67                  OD01 -> ABS_MI
68                  OD02 -> ABS_SOM
69
70          INIT_DEPS       /*NONE*/
71          REINIT_DEPS /*NONE*/
72
73  abstract ABS_AM : DRIVER_DEFINE
74          [Desc = "Air Mounts"]
75
76          OPER_DEPS       /*NONE*/
77          INIT_DEPS       /*NONE*/
78          REINIT_DEPS /*NONE*/
79
80  abstract ABS_ID : DRIVER_DEFINE
81          [Desc = "Image Sensor Scan Control"]
82
83          OPER_DEPS       /*NONE*/
84          INIT_DEPS
85                  ID01 -> ABS_IF
86          REINIT_DEPS
87                  RD01 -> ABS_IF
88
89  abstract ABS_IF : DRIVER_DEFINE
90          [Desc = "Image Sensor Subsystem"]
```

```
91
92          OPER_DEPS       /*NONE*/
93          INIT_DEPS       /*NONE*/
94          REINIT_DEPS /*NONE*/
95
96
97  abstract ABS_KS : DRIVER_DEFINE
98          [Desc = "Swap Control"]
99
100         OPER_DEPS       /*NONE*/
101         INIT_DEPS
102                 ID01 -> ABS_AM
103                 ID02 -> ABS_MI
104                 ID03 -> ABS_WS.Phase_B
105         REINIT_DEPS /*NONE*/
106
107 abstract ABS_MI : DRIVER_DEFINE
108         [Desc = "M. System Interferometers"]
109
110         OPER_DEPS       /*NONE*/
111         INIT_DEPS       /*NONE*/
112         REINIT_DEPS /*NONE*/
113
114 abstract ABS_IR : DRIVER_DEFINE
115         [Desc = "Int. Rtcl Inspect System"]
116
117         OPER_DEPS       /*NONE*/
118         INIT_DEPS       /*NONE*/
119         REINIT_DEPS /*NONE*/
120
121 abstract ABS_PS : DRIVER_DEFINE
122         [Desc = "Rtcl Particle Scanner"]
123
124         OPER_DEPS       /*NONE*/
125         INIT_DEPS       /*NONE*/
126         REINIT_DEPS /*NONE*/
127
128 abstract ABS_RV : DRIVER_DEFINE
129         [Desc = "Reticle Mover"]
130
131         OPER_DEPS       /*NONE*/
132         INIT_DEPS       /*NONE*/
133         REINIT_DEPS /*NONE*/
134
135 abstract ABS_RH : DRIVER_DEFINE
136         [Desc = "Reticle Handling"]
137
138         OPER_DEPS       /*NONE*/
139         INIT_DEPS       /*NONE*/
140         REINIT_DEPS /*NONE*/
141
142 abstract ABS_SNM : DRIVER_DEFINE
143         [Desc = "Synchronisation Control"]
144
145         OPER_DEPS       /*NONE*/
146         INIT_DEPS       /*NONE*/
```

```
147         REINIT_DEPS /*NONE*/
148
149 abstract ABS_SOM : DRIVER_DEFINE
150         [Desc = "Synchronisation Driver (T) M"]
151
152         OPER_DEPS       /*NONE*/
153         INIT_DEPS       /*NONE*/
154         REINIT_DEPS /*NONE*/
155
156 abstract ABS_WH : DRIVER_DEFINE
157         [Desc = "Wafer Handler"]
158
159         OPER_DEPS       /*NONE*/
160         INIT_DEPS       /*NONE*/
161         REINIT_DEPS /*NONE*/
162
163
164 abstract ABS_WS : DRIVER_DEFINE
165         [Desc = "Wafer Stage"]
166
167         Phase_A: DRIVER_phase
168                 Phase_A_init:Timeout
169                         [Phase_A_init.timeout = 20]
170
171                 INIT_DEPS /*NONE*/
172
173         Phase_B: DRIVER_phase
174                 Phase_B_init:Timeout
175                         [Phase_B_init.timeout = 90]
176
177                 INIT_DEPS
178                         ID01 -> ABS_MI
179
180
181         Phase_C: DRIVER_phase
182                 Phase_C_init:Timeout
183                         [Phase_C_init.timeout = 30]
184                 Phase_C_end: Timeout
185                         [Phase_C_end.timeout = 60]
186
187                 INIT_DEPS
188                         ID02 -> ABS_AM
189
190
191 DRIVER_DEFINITIONS
192         AC:ABS_AC
193         AF:ABS_AF
194         AM:ABS_AM
195         ID:ABS_ID
196         IF:ABS_IF
197         KS:ABS_KS
198         MI:ABS_MI
199
200         IR:ABS_IR ?
201         [INT_RETICLE_INSP_SYSTEM.variants.PRESENT <=> IR]
202         PS:ABS_PS ?
```

```
203         [INT_RETICLE_INSP_SYSTEM.variants.PRESENT <=> PS]
204         RV:ABS_RV ?
205         [INT_RETICLE_INSP_SYSTEM.variants.PRESENT <=> RV]
206
207         RH:ABS_RH
208         SNM:ABS_SNM
209         SOM:ABS_SOM
210         WH:ABS_WH
211         WS:ABS_WS
212
213 /*
214  *      VIRTUAL DRIVERS DEFINITIONS
215  */
216 abstract ABS_WW : VIRTUAL_DRIVER_DEFINE
217         [Desc = "ww"]
218         REF_WH -> DRIVER_DEFINITIONS.WH
219         REF_WS -> DRIVER_DEFINITIONS.WS
220
221
222 VIRTUAL_DRIVER_DEFINITIONS
223         WW:ABS_WW
```

Listing 5.2: SMDC specification example modelled in Clafer

References

[1] M. Galster, D. Weyns, M. Goedicke, U. Zdun, J. Cunha, and J. Chavarriaga, "Variability and complexity in software design: Towards quality through modeling and testing," *ACM SIGSOFT Software Engineering Notes*, vol. 42, pp. 35–37, Jan. 2018.

[2] P. C. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, ser. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[3] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Berlin, Heidelberg: Springer-Verlag, 2005.

[4] Wikipedia contributors, "Complex system — Wikipedia, the free encyclopedia," 2020, [Online; accessed 2-February-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Complex_system&oldid=935566621

[5] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli, "Modeling cyber–physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, Jan 2012.

[6] H. Eichelberger and K. Schmid, "Mapping the design-space of textual variability modeling languages: a refined analysis," *International Journal on Software Tools for Technology Transfer*, vol. 17, pp. 559–584, Oct. 2015.

[7] M. H. t. Beek, K. Schmid, and H. Eichelberger, "Textual variability modeling languages: An overview and considerations," in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B.* New York, NY, USA: Association for Computing Machinery, 2019, p. 151–157. [Online]. Available: https://doi.org/10.1145/3307630.3342398

[8] J. van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001, pp. 45–54.

[9] K. Pohl and A. Metzger, "Variability management in software product line engineering," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 1049–1050. [Online]. Available: https://doi.org/10.1145/1134285.1134499

[10] A.-L. Lamprecht, S. Naujokat, and I. Schaefer, "Variability management beyond feature models," *Computer*, vol. 46, pp. 48–54, Nov. 2013.

[11] K. Schmid, R. Rabiser, and P. Grünbacher, "A comparison of decision modeling approaches in product lines," in *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, ser. VaMoS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 119–126. [Online]. Available: https://doi.org/10.1145/1944892.1944907

[12] K. Deelstra and M. Sinnema, "Managing the complexity of variability in software product families," 2008, date_submitted:2008 Rights: University of Groningen (Publisher).

[13] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski, "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*, ser. VaMoS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 173–182. [Online]. Available: https://doi.org/10.1145/2110147.2110167

[14] R. Bashroush, M. Garba, R. Rabiser, I. Groher, and G. Botterweck, "Case tool support for variability management in software product lines," *ACM Computing Surveys*, vol. 50, no. 1, Mar. 2017. [Online]. Available: https://doi.org/10.1145/3034827

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-021, 1990. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[16] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '13*. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2430502.2430513

[17] D. Lettner, M. Petruzelka, R. Rabiser, F. Angerer, H. Prähofer, and P. Grünbacher, "Custom-developed vs. model-based configuration tools: Experiences from an industrial automation ecosystem," in *Proceedings of the 17th International Software Product Line Conference Co-Located Workshops, SPLC '13 Workshops*. New York, NY, USA: Association for Computing Machinery, 2013, p. 52–58. [Online]. Available: https://doi.org/10.1145/2499777.2500713

[18] "Kconfig language specification," https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt, accessed: 2020-02-11.

[19] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the Linux kernel variability model," in *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, SPLC'10*. Berlin, Heidelberg: Springer-Verlag, 2010, p. 136–150.

[20] M. Acher, H. Martin, J. Alves Pereira, A. Blouin, D. Eddine Khelladi, and J.-M. Jézéquel, "Learning From Thousands of Build Failures of Linux Kernel Configurations," Inria ; IRISA, Technical Report, Jun. 2019. [Online]. Available: https://hal.inria.fr/hal-02147012

[21] A. Jaksic, R. France, P. Collet, and S. Ghosh, "Evaluating the usability of a visual feature modeling notation." 7th International Conference, SLE 2014, Sept. 2014, pp. 122–140.

[22] H. Eichelberger and K. Schmid, "A systematic analysis of textual variability modeling languages," in *Proceedings of the 17th International Software Product Line Conference, SPLC '13*. New York, NY, USA: Association for Computing Machinery, 2013, p. 12–21. [Online]. Available: https://doi.org/10.1145/2491627.2491652

[23] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615 – 636, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437910000025

[24] K. Schmid, R. Rabiser, and P. Grünbacher, "A comparison of decision modeling approaches in product lines," in *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, ser. VaMoS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 119–126. [Online]. Available: https://doi.org/10.1145/1944892.1944907

[25] A. Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *Journal of Computing and Information Technology*, vol. 10, Jan. 2002.

[26] T. Asikainen, T. Mannisto, and T. Soininen, "A unified conceptual foundation for feature modelling," in *10th International Software Product Line Conference (SPLC'06)*, Aug 2006, pp. 31–40.

[27] V. Myllärniemi, T. Asikainen, T. Männistö, and T. Soininen, "Kumbang configurator–a configuration tool for software product families," 01 2005.

[28] D. Batory, "Feature models, grammars, and propositional formulas," in *Software Product Lines*, H. Obbink and K. Pohl, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 7–20.

[29] A. Abele, Y. Papadopoulos, D. Servat, M. Törngren, and M. Weber, "The cvm framework - a prototype tool for compositional variability management." Jan. 2010, pp. 101–105.

[30] M. Mendonça, M. Branco, and D. Cowan, "S.P.L.O.T. - Software product lines online tools," Oct. 2009, pp. 761–762.

[31] M. Acher, P. Collet, P. Lahire, and R. France, "A domain-specific language for managing feature models," Jan. 2011, pp. 1333–1340.

[32] A. Classen, Q. Boucher, and P. Heymans, "A text-based approach to feature modelling: Syntax and semantics of TVL," *Science of Computer Programming*, vol. 76, no. 12, pp. 1130 – 1143, 2011, special Issue on Software Evolution, Adaptability and Variability. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642310001899

[33] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte, "Variability modelling in the abs language," vol. 6957, Nov. 2010, pp. 204–224.

[34] M. Rosenmüller, N. Siegmund, T. Thüm, and G. Saake, "Multi-dimensional variability modeling," in *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, ser. VaMoS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 11–20. [Online]. Available: https://doi.org/10.1145/1944892.1944894

[35] K. Schmid, C. Kröher, and S. El-Sharkawy, "Variability Modeling with the Integrated Variability Modeling Language (IVML) and EASy-Producer," in *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC '18*. New York, NY, USA: Association for Computing Machinery, 2018, p. 306. [Online]. Available: https://doi.org/10.1145/3233027.3233057

[36] Eichelberger and El-Sharkawy, Sascha and Holger and Kröher, Christian and Schmid, Klaus, "Integrated Variability Modeling Language: Language specification – version 1.30," 2015, [Online; accessed 12-May-2020]. [Online]. Available: https://github.com/SSEHUB/EASyProducer/blob/master/doc/IVML%20Language%20Spec.docx

[37] P. Juodisius, A. Sarkar, R. R. Mukkamala, M. Antkiewicz, K. Czarnecki, and A. Wasowski, "Clafer: Lightweight modeling of structure, behaviour, and variability," *CoRR*, vol. abs/1807.08576, 2018. [Online]. Available: http://arxiv.org/abs/1807.08576

[38] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski, "Clafer: Unifying class and feature modeling," *Software Systems Modeling*, vol. 15, Dec. 2014.

[39] A. A.F, "PYFML - A Textual Language for Feature Modeling," *International Journal of Software Engineering Applications*, vol. 9, pp. 41–53, Jan. 2018.

[40] E. Alférez Salinas, M. Acher, J. Galindo, B. Baudry, and D. Benavides, "Modeling variability in the video domain: Language and experience report," *Software Quality Journal*, vol. 27, p. 307–347, March, 2019.

[41] A. Villota, R. Mazo, and C. Salinesi, "The high-level variability language: An ontological approach," in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, ser. SPLC '19. New York, NY, USA: ACM, 2019, pp. 162–169. [Online]. Available: http://doi.acm.org/10.1145/3307630.3342401

[42] K. Schmid, "Variability support for variability-rich software ecosystems," in *2013 4th International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*, 2013.

[43] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, pp. 7–29, Jan. 2005.

[44] ——, "Staged configuration using feature models," in *Software Product Lines*, R. L. Nord, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 266–283.

[45] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '13*. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2430502.2430513

[46] M. Jaring and J. Bosch, "Representing variability in software product lines: A case study," in *Software Product Lines*, G. J. Chastek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 15–36.

[47] O. Djebbi and C. Salinesi, "Criteria for comparing requirements variability modeling notations for product lines," in *Fourth International Workshop on Comparative Evaluation in Requirements Engineering (CERE'06 - RE'06 Workshop)*, 2006, pp. 20–35.

[48] M.-O. Reiser, *Core Concepts of the Compositional Variability Management Framework (CVM): A Practitioner's Guide*. Technische Universität Berlin, 2009.

[49] J. Liang, "Solving clafer models with choco," no. GSDLab-TR 2012-12-30, 12/2012 2012.

[50] T. Asikainen, T. Männistö, and T. Soininen, "Kumbang: A domain ontology for modelling variability in software product families," *Advanced Engineering Informatics*, vol. 21, no. 1, pp. 23 – 40, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S147403460600067X

[51] OMG, *Object Constraint Language Specification, version 2.0*, Object Modeling Group, June 2005. [Online]. Available: http://fparreiras/papers/OCLSpec.pdf

[52] J. Meinicke, T. Thüm, R. Schrter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability with FeatureIDE, pp. 58*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[53] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93–95, April 2008.

[54] J. Bosch and R. Capilla, "Dynamic variability in software-intensive embedded system families," *Computer*, vol. 45, no. 10, pp. 28–35, Oct 2012.

[55] M. Hinchey, S. Park, and K. Schmid, "Building dynamic software product lines," *Computer*, vol. 45, no. 10, pp. 22–26, Oct 2012.

[56] R. Olaechea, S. Stewart, K. Czarnecki, and D. Rayside, "Modelling and multi-objective optimization of quality attributes in variability-rich software," in *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*, ser. NFPinDSML '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2420942.2420944

[57] S. Nadi and S. Krüger, "Variability modeling of cryptographic components: Clafer experience report," Jan. 2016, pp. 105–112.

[58] J. A. Ross, A. Murashkin, J. H. Liang, M. Antkiewicz, and K. Czarnecki, "Synthesis and exploration of multi-level, multi-perspective architectures of automotive embedded systems (sosym abstract)," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2017.

[59] Wikipedia contributors, "Expressive power (computer science) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Expressive_power_(computer_science)&oldid=929184095, 2019, [Online; accessed 16-March-2020].

[60] A. Villota, R. Mazo, and C. Salinesi, "On the ontological expressiveness of the high-level constraint language for product line specification," in *System Analysis and Modeling. Languages, Methods, and Tools for Systems Engineering*, F. Khendek and R. Gotzhein, Eds. Cham: Springer International Publishing, 2018, pp. 46–66.

[61] Y. Wand and R. Weber, "On the deep structure of information systems," *Information Systems Jorunal*, vol. 5, pp. 203–223, 1995.

[62] ——, "On the ontological expressiveness of information systems analysis and design grammars," *Information Systems Journal*, vol. 3, no. 4, pp. 217–237, 1993. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2575.1993.tb00127.x

[63] J. Evermann and Y. Wand, "Ontology based object-oriented domain modelling: Fundamental concepts," *Requirements Engineering*, vol. 10, pp. 146–160, May 2005.

[64] M. Asadi, D. Gasevic, Y. Wand, and M. Hatala, "Deriving variability patterns in software product lines by ontological considerations," in *Conceptual Modeling*, P. Atzeni, D. Cheung, and S. Ram, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 397–408.

[65] M. Bunge, *Treatise on Basic Philosophy (Vol. 3): Ontology I: The Furniture of the World*. Boston, MA: D. Reidel Publishing Company, 1977.

[66] G. Guizzardi, "Ontological foundations for structural conceptual models," Ph.D. dissertation, University of Twente, Nov. 2005.

[67] R. Guizzardi, X. Franch, and G. Guizzardi, "Applying a foundational ontology to analyze means-end links in the i framework," 05 2012, pp. 1–11.

[68] G. Guizzardi, *Ontology-Based Evaluation and Design of Visual Conceptual Modeling Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 317–347. [Online]. Available: https://doi.org/10.1007/978-3-642-36654-3_13

[69] I. Reinhartz-Berger, A. Sturm, and Y. Wand, "External variability of software: Classification and ontological foundations," in *Conceptual Modeling – ER 2011*, M. Jeusfeld, L. Delcambre, and T.-W. Ling, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 275–289.

[70] P. Green and M. Rosemann, "An ontological analysis of integrated process modelling," in *Advanced Information Systems Engineering*, M. Jarke and A. Oberweis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 225–240.

[71] H. Gregersen and C. S. Jensen, "On the ontological expressiveness of temporal extensions to the entity-relationship model," in *Advances in Conceptual Modeling*, P. P. Chen, D. W. Embley, J. Kouloumdjian, S. W. Liddle, and J. F. Roddick, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 110–121.

[72] J. Mylopoulos, "Conceptual modeling and telos," in *Conceptual Modeling, Databases, and Cases*. New York, NY, USA: Wiley, 1992.

[73] Y. Wand, V. C. Storey, and R. Weber, "An ontological analysis of the relationship construct in conceptual modeling," *ACM Trans. Database Syst.*, vol. 24, no. 4, p. 494–528, Dec. 1999. [Online]. Available: https://doi.org/10.1145/331983.331989

[74] Y. Wand and R. Weber, "An ontological evaluation of systems analysis and design methods," in *Information system concepts: an in-depth analysis.* Elsevier science publishers, BV, 1989.

[75] P. Soffer, B. Golany, D. Dori, and Y. Wand, "Modelling off-the-shelf information systems requirements: An ontological approach," *Requirements Engineering*, vol. 6, pp. 183–199, Oct. 2001.

[76] M. Bashari, E. Bagheri, and W. Du, "Dynamic software product line engineering: A reference framework," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, pp. 191–234, Mar. 2017.

[77] M. A. Khalidi, *Kinds (Natural Kinds vs. Human Kinds), in B. Kaldis (ed.)Encyclopedia of Philosophy and the Social Sciences.* Thousand Oaks,CA.: Sage, 2013.

[78] M. Becker, "Xml-enhanced product family engineering," 2002.

[79] P. Heymans, P. Y. Schobbens, J.-C. Trigaux, R. Matulevičius, A. Classen, and Y. Bontemps, "Towards the comparative evaluation of feature diagram languages," Jan. 2007.

[80] "MS Windows NT kernel description," https://www.eclipse.org/xtend/index.html, online. Accessed: 2020-05-28.

[81] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach, Chapter 6*, 3rd ed. Prentice Hall, 2010.

[82] G. Grozdev, *EPS System Manager Driver Control (D000559086/02)*, ASML.

[83] J. Petke, *Bridging Constraint Satisfaction and Boolean Satisfiability.* Springer, 2015.