Extending Feature Models with Types

Benjamin Callewaert
KU Leuven, De Nayer Campus
Dept. of Computer Science
Leuven.AI, Belgium
Flanders Make@KU Leuven
benjamin.callewaert@kuleuven.be

Simon Vandevelde
KU Leuven, De Nayer Campus
Dept. of Computer Science
Leuven.AI, Belgium
Flanders Make@KU Leuven
s.vandevelde@kuleuven.be

Nuno Comenda
Intelli-Select
nuno.comenda@intelli-select.com

Bart Coppens
Intelli-Select
bart.jan.coppens@intelli-select.com

Nicholas Decleyre
Intelli-Select
nicholas.decleyre@intelli-select.com

Joost Vennekens
KU Leuven, De Nayer Campus
Dept. of Computer Science
Leuven.AI, Belgium
Flanders Make@KU Leuven
joost.vennekens@kuleuven.be

Abstract

Feature models are diagrams representing the variability of a product. While they are beneficial in reducing time, costs, and risks, their application in several problem areas remains unexplored. For example, in the financial domain, feature models could be leveraged to represent the variability of complex financial products and aid users in managing them. However, existing feature modeling approaches have limitations w.r.t. representing large products with complex features. To address some of these limitations, this work proposes a new approach called typed feature modeling, which extends the expressiveness of feature modeling by associating features with types that cover specified or infinite domains. We demonstrate the effectiveness of this approach by applying it to represent complex financial products that follow a commonly used industry standard. Additionally, we present an interactive tool for typed feature modeling containing an implementation of the financial use case and demonstrate how it can assist users in managing their financial products.

Feature modeling, Domain modeling, Knowledge Representation and Reasoning

1. Introduction

Feature modeling (Kang et al., 1990a) has established itself as the widely accepted standard for variability modeling. Introduced nearly three decades ago, the original feature modeling notation has since seen numerous variations and extensions. The goal of feature modeling is to create a diagram (feature model) that represents all the features and components of a product, and how they relate to each other. In this way, this diagram provides a comprehensive view of all possible configurations of the product. The use of feature modeling brings several advantages, including reducing the time it takes to deliver a product, lowering costs, minimizing product risks, and streamlining labor efforts (Northrop, 2008).

Although feature models are commonly employed in the context of software product development, they also hold significant potential for many other domains (Hein et al., 2000). For example, they could be used in the development of design tools that assist engineers in creating accurate designs (Vandevelde, Callewaert, et al., 2022). In the world of finance, the potential of feature models is still untapped. Yet, feature models lend themselves well to representing the variability of complex financial products. This, in turn, could be leveraged to develop tooling to assist users in managing such products. However, in practical scenarios, feature modeling and related tools are too limited for these purposes. Financial products often involve numerous complex features, each with a large number of variations. On top of that, several of these complex features might take on the value from the same domain. Representing these complex features using basic feature modeling can lead to a rapid explosion of the size of the feature diagram, making it very hard to maintain and interpret. In addition, they lack the ability to represent knowledge from "outside" the product, such as background or contextual knowledge Hence, basic feature modeling alone may not be sufficient for complex use cases in diverse domains.

In this work, we propose an extension to feature modeling, called *typed feature modeling*, which aims to enhance the expressiveness of feature modeling while maintaining its intuitive nature. To do this, we allow features to be associated with types that can

URI: https://hdl.handle.net/10125/107010
978-0-9981331-7-1
(CC BY-NC-ND 4.0)

cover enumerated or infinite domains. We demonstrate the effectiveness of this extension by using it to represent complex financial products, showcasing the benefits it brings and its potential for application in various, complex domains. Furthermore, we present an interactive tool for typed feature modeling and illustrate how it, together with a model for the financial use case, can assist users in efficiently managing their financial products.

The paper is structured as followed: we begin by providing background information in Section 2. Afterwards, in Section 3, we present our extension of feature modeling, typed feature modeling, and define its semantics. Next, we illustrate typed feature modeling with a use case in the financial domain, in Section 4. We present our interactive typed feature modeling tool in Section 5 and show how it can leverage the typed feature model diagram from the previous chapter to help users manage financial products. In Section 6 we compare our extension to other feature modeling extensions and finally, we conclude in Section 7.

2. Preliminaries

2.1. Feature Modeling

The goal of feature modeling is to provide an overview of a product and its variability. It was introduced in 1990 for software product development (Kang et al., 1990a), and has since gained wide adoption within the industry. We will briefly go over the concepts and components of a feature model diagram, using Fig. 1 as an example. In this example, we have modeled the variability of a simple financial *security*, which consists of multiple *features* as denoted by its child nodes. There are four types of relationships possible between parents and their children:

- A **mandatory** feature is present in every configuration in which its parent feature is present. This is denoted by an edge ending in a filled-in circle, such as for *Issuer*.
- An **optional** feature may or may not be present if its parent is present. This is denoted by an edge ending in an empty circle, such as for *Rating*.
- Children can be **alternative** features to one another, meaning that only one of the child features can be present if the parent feature is present. This is denoted by a curve between the edges of the child nodes, as shown for *Equity*, *Bond*, and *Fund*.

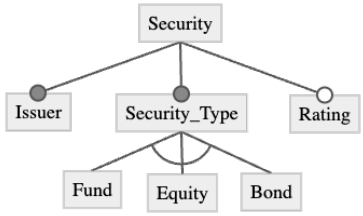

Figure 1. Example of a feature model

- Children can be in an (inclusive) **or** relationship, meaning that at least one child is present if the parent feature is present. This relationship is not shown in the example but is denoted by a filled-in curve between the edges.

In addition to the parent-child relationships, there are also two types of cross-tree constraints. The first type is known as a **require** constraint, where one feature requires the inclusion of another feature. The second type is an **exclude** constraint, which specifies that certain features cannot coexist in the same configuration. These cross-tree constraints are usually visualized within the feature model by drawing an arrow between the respective nodes or providing textual explanations separately. In the visual representation, a single arrow denotes a required constraint between the nodes, whereas a double arrow indicates an excluded constraint.

2.1.1. Feature modeling extensions Since its introduction, several extensions for feature modeling have been proposed to make feature models more suitable for practical applications and to provide better "conceptual completeness". The two most prominent ones are cardinality-based feature modeling (Czarnecki et al., 2004) and extended feature modeling (Benavides et al., 2005).

Cardinality-based feature modeling Some authors propose extending feature models with UML-like multiplicities (so-called cardinalities) (Czarnecki et al., 2004; Riebisch et al., 2002). The new relationships introduced in this notation are defined as follows:

- A **feature cardinality** is an interval denoted $[n \dots m]$ with n as the lower bound and m as the upper bound. These intervals determine the number of instances of the feature that can be part of a product. This relationship may be used as a generalization of the mandatory ($[1 \dots 1]$) and optional ($[0 \dots 1]$) relationships.
- A **group cardinality** is an interval denoted

$\langle$n . . . m$\rangle$, with n as the lower bound and m as the upper bound limiting the number of child features that can be part of a product when its parent feature is selected. This relationship generalizes the "alternative" relationship ($\langle 1 \ldots 1 \rangle$) and the "or"-relationship ($\langle 1 \ldots$ n$\rangle$, with n the number of features in the relationship).

Extended Feature Modeling Sometimes, it becomes necessary to enhance feature models by incorporating additional information about a feature. This additional information is referred to as feature attributes. When a feature model includes these attributes, it is known as an extended, advanced, or attributed feature model. A feature attribute represents a measurable characteristic of a feature, and each attribute is associated with a specific domain, which defines its range of possible values. Such a domain can be both finite (e.g., an interval) or infinite (e.g., all real numbers). Attributes can represent various characteristics of a feature, such as the cost associated with a feature. Extended feature models also enable the involvement of attributes in cross-tree relations to express complex constraints. Several groups of authors (Benavides et al., 2005; Czarnecki et al., 2005; Kang et al., 1990b) have suggested the inclusion of attributes or "non-functional" features in feature models, but there seems to be no unanimous agreement on the specific notation for defining attributes.

2.2. FO(·), the IDP system and the Interactive Consultant

Chapter 5 presents a tool for our feature modeling extension, based on a state-of-the-art reasoning engine. To give sufficient background, we now will briefly discuss this reasoning engine and related concepts.

The **IDP system** (De Cat et al., 2018) is a reasoning engine for FO(·), a rich extension of First-Order Logic (FOL). It implements the philosophy of the Knowledge Base Paradigm (Denecker & Vennekens, 2008): knowledge is represented in a purely declarative manner, independent from how it is used. This is done by storing the knowledge in a Knowledge Base (KB), to which then various inference tasks can be applied to put it to practical use. This approach has two main advantages. Firstly, declaratively representing the knowledge is often easier than developing algorithmic solutions to specific tasks. Secondly, the split between knowledge and its application facilitates the re-use of the knowledge for multiple purposes. In this way, different problems in the same domain can typically be solved using the same KB with different inference tasks.

FO(·) extends FOL with types, aggregates, inductive definitions, arithmetic, partial functions, and intensional objects. In this way, FO(·) is an expressive and versatile representation language, well-suited for modeling problems from many domains.

FO(·) also allows for partial functions. Normally, functions are total: they assign an output value to each of the input values. On the other hand, partial functions do not necessarily have this property. A partial function f with domain D and range R is defined as followed: for every $d \in D$ there exists at most one element $y \in R$ such that $(x, y) \in f$. In this case, $f(x)$ may not exist.

The KB itself consists of three types of blocks: *vocabularies*, *structures* and *theories*, each representing the corresponding concept from classical logic.

A *vocabulary* specifies a set of *type*, *predicate*, or *function* symbols. A type is a domain of values, such as a list of strings or the domain of real numbers $\mathbb{R}$. A predicate symbol expresses a relation on zero or more types. A proposition is a 0-ary predicate. Lastly, a function symbol expresses a function from the Cartesian product of a number of types $T_1 \times \cdots \times T_n$ to a type T_{n+1}. A function is also called a *constant* if it is 0-ary, i.e., has no input arguments.

A *structure* provides an interpretation for the symbols in its vocabulary. If it provides an interpretation for each symbol in the vocabulary, it is called a *full* interpretation. Otherwise, it is called a partial interpretation.

A *theory* contains a set of logical formulas, written in FO(·).

By itself, the KB is not executable: it merely represents the knowledge of a domain. To put this knowledge to use, the IDP system offers multiple inference tasks, like propagation and model expansion.

In the past, the IDP system has already proven itself as a suitable tool for configuration problems (Aerts et al., 2022; Carbonnelle et al., 2022; Van Hertum et al., 2017; Vlaeminck et al., 2009). Its approach works well to tackle complex configuration problems in an interactive way. Such applications can be found in many domains, such as manufacturing, finance, and logistics. The IDP system typically performs best if there are many constraints over relatively small domains, and if the numerical calculations that are involved are not too complex. The latest version of the IDP system, and the one used in this work, is IDP-Z3 (Carbonnelle et al., 2023).

The **Interactive Consultant** is a user-friendly, interactive interface for the IDP-Z3 system. It is fully generic, in the sense that it can generate an interface for any syntactically correct FO(·) KB. The interface displays a *tile* for each symbol in the KB, which

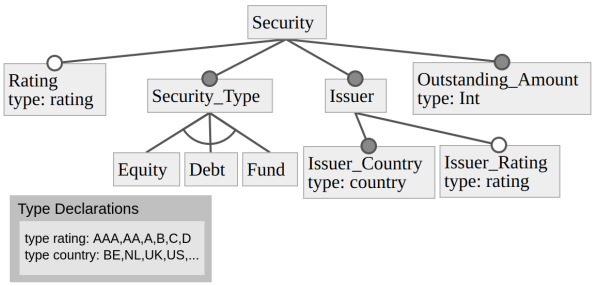

Figure 2. Example of a typed feature model

allows users to assign values to these symbols. In the background, IDP-Z3 then derives the consequences of that value assignment and updates the other symbol tiles accordingly. This process continues until all symbols have been assigned a value (either by the user or by the system), ensuring compliance with the formulas in the KB. Moreover, users can request explanations for derived consequences by clicking on values. This makes the tool *explainable*: in such a case, it will return a list of relevant value assignments and formulas.

One of the design goals of the Interactive Consultant is to empower users to interactively explore a problem space. Indeed, due to its intuitive interface and powerful solver, users can easily interact with the knowledge in the KB. This has already proven useful in several successful past use cases, such as for component design (Aerts et al., 2022), to support notaries with legislation (Deryck et al., 2019), and to select suitable adhesives in a manufacturing context (Jordens et al., 2022; Vandevelde, Jordens, et al., 2022).

3. Typed Feature Modeling

Typed feature modeling is an extension of feature modeling that aims to represent complex products in a compact manner while maintaining intuitiveness. It allows a feature to be associated with a type that defines its domain – i.e., the typed feature is limited to a value from that domain. In this way, the variability of complex features can be represented in a compact way. Without this extension, such features could only be represented by adding each element from the domain as a distinct subfeature. This leads to a massive increase in the size of the overall feature diagram, resulting in a cluttered and unclear representation. Moreover, this approach is not possible for features that have a type with an infinite domain.

Fig. 2 shows an example of a typed feature model (TFM) that captures the variability of a simple financial security. In the TFM, two types are defined: *rating*, which contains different ratings that can be assigned to a security (ranging from AAA to D), and *country*, which contains a list of countries. The defined types are listed in a compact legend, shown under the feature tree. Next to the defined types, a TFM also supports built-in types covering infinite domains, namely integers, real numbers and dates.

The example in Fig. 2 contains four typed features. Both *Rating* and *Issuer_Rating* have the type *rating*; they both take on one value from the domain if they are present. The feature *Issuer_Country* represents the country of origin of the security's issuer and is therefore associated with the *country* type. Finally, the feature *Outstanding_Amount* denotes the number of available securities and is associated with the (infinite) integer domain.

3.1. Formal semantics

We define the semantics of typed feature models by means of a translation to FO(·).

Types First, we define the user-defined types in our vocabulary. The domain of a type can either be finite or infinite. For each type t_i, with domain d_i, declared in the typed feature model, we define a type in the FO(·) vocabulary:

$$\text{type } t_i \coloneqq \{v_j : v_j \in d_i\}$$

Features For the translation of untyped features we use the semantics defined by Batory (Batory, 2005), in which each feature is represented by a proposition. For each typed feature, we introduce a partial 0-ary function on its type. In FO(·), we can represent these features as:

- $f_i : () \to Bool$ if f_i is an untyped feature
- partial $f_i : () \to t_i$ if f_i is a feature that is associated with type t_i

Next, we also introduce a type $Feature$, consisting of all features $F_1 \dots F_n$ in the TFM, which we define as:

$$\text{type } Feature \coloneqq \{F_1 \dots F_n\}$$

In other words, we introduce for each feature f_i a constant, and a value in the domain of the type $Feature$.

Relations The predicate $present : Feature \to Bool$ indicates which features are present in the configuration. We define the function $present$ for every typed feature

Relation	Translation to FO(·)
f_2 is mandatory for f_1	$present(F_1) \Leftrightarrow present(F_2)$
f_2 is optional for f_1	$present(F_1) \Leftarrow present(F_2)$
f_1 has alternative $f_0 \dots f_m$	$present(F_1) \Leftrightarrow$ $\bigvee_{i=0}^{m} present(F_i) \wedge \bigwedge_{0 \leq j \leq k \leq m} \neg(present(F_j) \wedge present(F_k))$
f_1 has OR $f_0 \dots f_m$	$present(F_1) \Leftrightarrow (present(F_o) \vee \dots \vee present(F_m))$
f_1 requires f_2	$present(F_1) \Rightarrow present(F_2)$
f_1 excludes f_2	$present(F_1) \Rightarrow \neg present(F_2)$

Table 1. Translation between feature relations and FO(·)

$f_1 \dots f_n$, associated with their respective type $t_1 \dots t_n$:

$$\begin{aligned} present(F_1) &\Leftrightarrow \exists x \in t_1 : f_1() = x. \\ &\dots \\ present(F_n) &\Leftrightarrow \exists x \in t_n : f_n() = x. \end{aligned} \tag{1}$$

For each untyped feature f_i we define the function $present$ as:

$$present(F_i) \Leftrightarrow f_i() = true \tag{2}$$

We then define the semantics of the relationships between features as in Table 1.

4. Financial use case

ISDA CDM The financial world relies on computing standards to meet the very high demands of today's global, always-on financial environment. Without standards, the lifecycle processing of trades would be a manual, error-prone, and time-consuming process. However, the trading and management of securities so far have not used this standardization to its fullest extent yet. To address this issue, the International Swaps and Derivatives Association (ISDA) developed the *Common Domain Model (CDM)* (2020) as the industry's first solution to establish standard conventions for securities. The CDM is a standardised, machine-readable and machine-executable blueprint for how securities are traded and managed across the transaction lifecycle. It encompasses all dimensions of the security lifecycle, ranging from product description to defining the potential events that may occur while trading securities. The CDM also introduces data rules and calculation artefacts to ensure compliance and reduce ambiguities in implementing the standard. This approach is a significant step towards realizing the intended benefits, promoting consistency, and enabling greater automation and efficiency at scale.

CDM as TFM To enhance the understanding of the ISDA CDM and management of securities following the standard, we propose to partially represent it as a typed feature model. While the ISDA CDM encompasses the entire transactional lifecycle of financial securities, our focus is on representing the variability of a security according to the standard. By using the intuitiveness of typed feature modeling, this approach has the potential to greatly benefit financial experts by providing a clear representation of the complex model and assisting them in effectively managing their CDM-compliant securities. Fig. 3 shows a typed feature model that models the variability of these securities following the CDM standard. In the model, 14 types are defined. For instance, the *exchange* type encompasses all the stock exchanges on which a security may be listed. Although the model remains relatively large, it effectively captures a significant amount of information. By allowing features to be associated with types, we can represent the entire variability of CDM-compliant securities in a concise and intuitive manner. This approach is essential as complex features such as *sector* and *rating* have numerous possible values, and representing them as individual sub-features would overcrowd the feature model and hinder a clear understanding of the overall variability.

5. TFM-IDP tool

The TFM-IDP tool combines the simplicity and intuitiveness of typed feature modeling with the interactivity of the Interactive Consultant. It aims to support interactive exploration of the variability of a product complemented with background knowledge, which is useful both for an expert to validate typed feature models and for making concrete, error-free configurations. The tool's interface is split into two tabs: a typed feature model editor, where diagrams can be created and background knowledge can be added, and a user interface to interact with the feature model. The tool is an extension of the FM-IDP tool developed by Vandevelde, Callewaert, et al. (2022) and can be

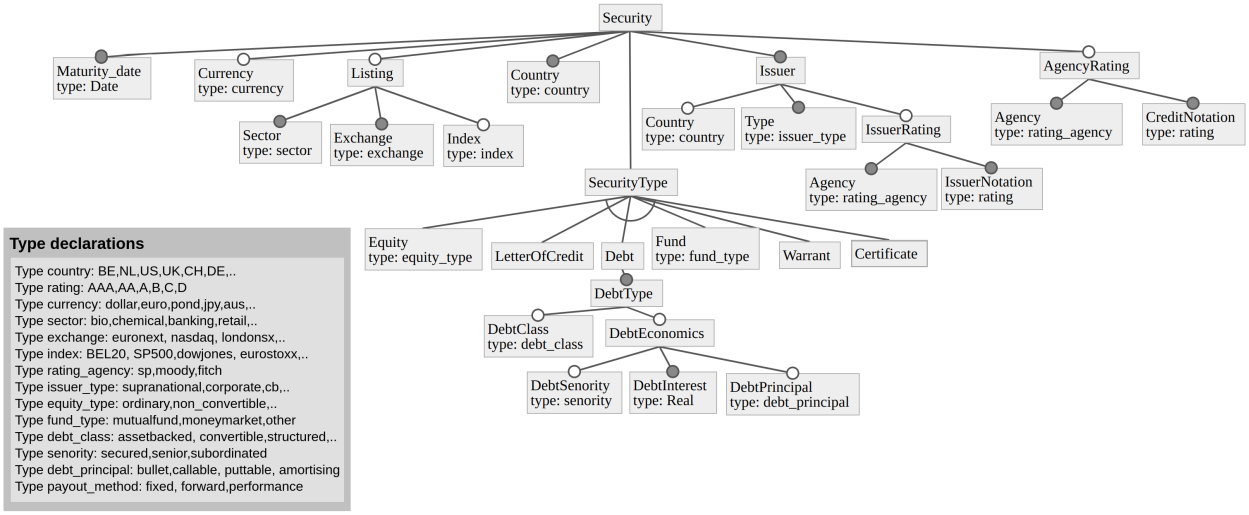

Figure 3. Typed feature model of ISDA CDM security

accessed online[1].

5.1. Typed Feature Model editor

The primary component of the tool is a visual editor for creating typed feature models. This editor is displayed on the left side of the screenshot in Fig. 4. To create a feature diagram, users simply click on the canvas to create a new node, assign a name to it, establish a connection by clicking on its parent node, and specify the connection type. Types can be defined in the legend located at the bottom left corner of the editor. In this legend, types are declared by giving them a name and specifying their domain of values. Once a type is defined, features can be associated with it.

Once all features have been added and the types have been declared, the model is converted into an FO(·) vocabulary and theory. These are visible to the user through a read-only text editor, present in the bottom-right of the screen. Since IDP-Z3 does not support partial functions, in contrast to the previous version of the IDP system (IDP3 (De Cat et al., 2018)), we slightly deviate from the semantics defined in Section 2. Instead of introducing a partial function, we define a total function for each typed feature. To overcome this difference, we dismiss the definition of the $present$ function for typed features (Eq. 1). This means that every typed feature f_i will always have a value, but this value should be ignored if $present(F_i)$ is false.

Typed feature models represent the variability of a product in an intuitive way but they lack the ability to represent knowledge from "outside" the product, limiting their use in real-life applications. Therefore, we also allow for background knowledge to be specified in our tool. This background knowledge can be written in an FO(·) editor, present on the top-right of the screen. The editor allows declaring additional concepts outside the product. We could, for example, introduce the concept *fed_rate*, representing the Federal funds rate, as this is not a feature of a security but could influence its behaviour. The editor also allows expressing additional FO(·) formulae in the theory. In this case, we could specify that the maturity date should always be a date that comes after the current day using the built-in literal #TODAY.

$$Maturity_date() > \#TODAY.$$

Behind the scenes, the vocabulary and theory blocks are merged with their counterparts generated from the feature model. In this way, they form one complete KB containing both the knowledge of the feature model and its required background knowledge.

5.2. Knowledge interaction

Interaction with the knowledge is facilitated through the Interactive Consultant (Carbonnelle et al., 2019), as shown in the screenshot in Fig. 5. Every time a feature is selected or is given a value, IDP-Z3 automatically derives the consequences of this assignment. This process of selecting features in the interface and immediately viewing the consequences results in a tight feedback loop between the user and the knowledge. Furthermore, users can request explanations for derived

[1]https://fm-idp.onrender.com/

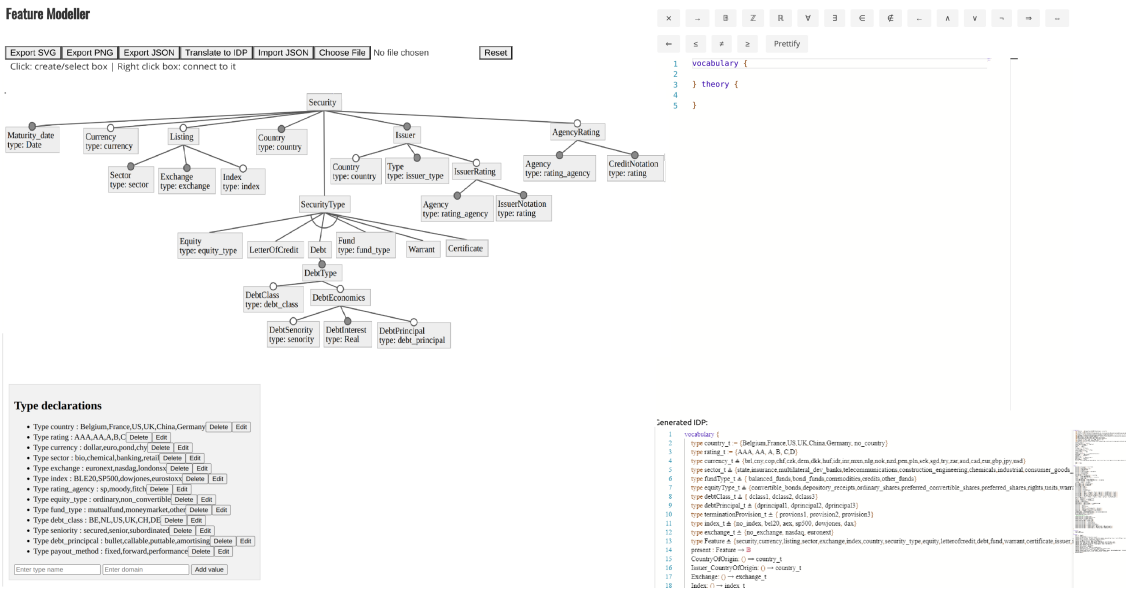

Figure 4. Screenshot of the CDM security use case in the typed feature model editor

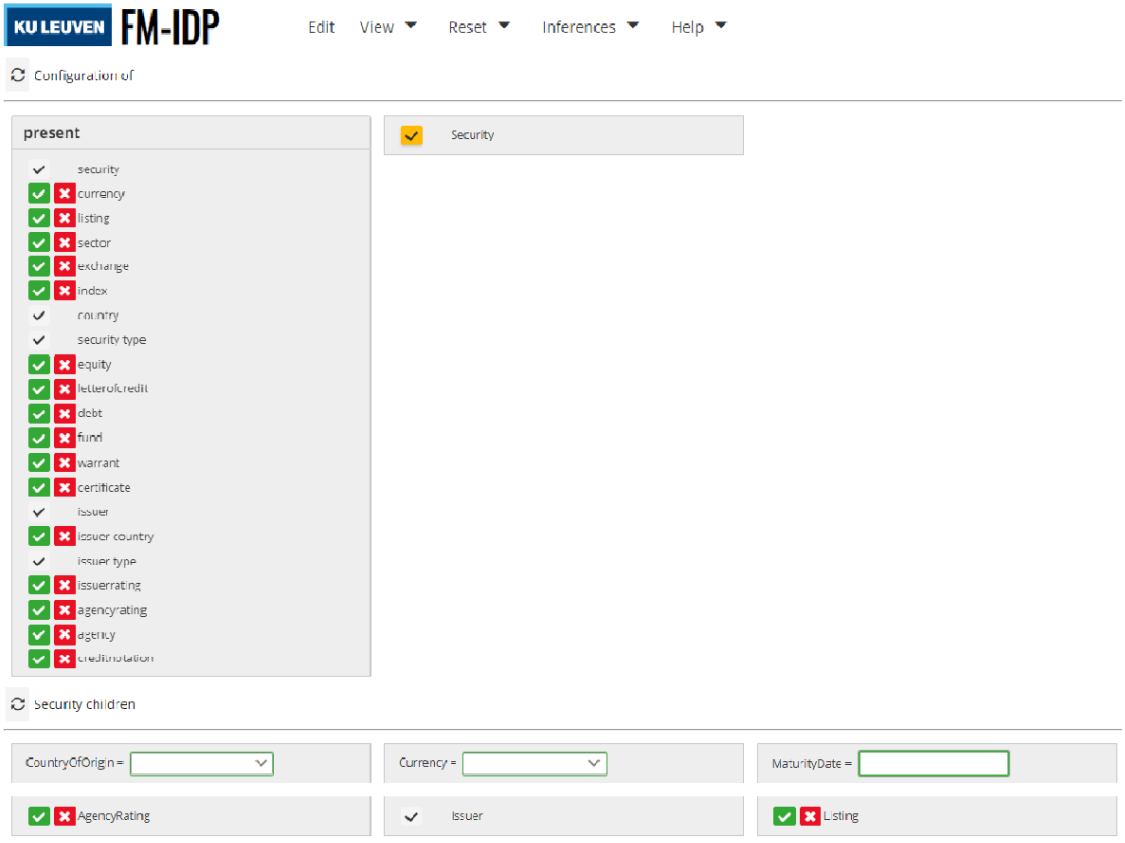

Figure 5. Screenshot of the CDM security use case in the configurator

consequences by clicking on values.

Representing the CDM standard as a typed feature model and implementing it in the TFM-IDP tool can help financial experts in several ways.

- The tool allows **interactive exploration** of the different variants of securities that comply with the CDM standard. This enables users to enhance their understanding of the standard itself and the associated constraints.
- Financial companies often have diverse ways of modeling financial securities within their own specific service environment. With the TFM tool, users can **construct securities** that were modeled differently by dynamically selecting features and assigning values to typed features. In this way, users are guided to model securities that follow the constraints of the CDM standard.
- Users can employ the tool to check whether their securities conform to the CDM standard. If not, the provided explanations can help them alter their security specification to conform to the CDM standard. The tool therefore also serves as a **verification mechanism**.

6. Related Work

Several extensions of feature modeling have been proposed throughout the years to allow the notation to be applied in different application domains. Although original feature models are expressively complete, as Schobbens et al. (2007) point out, extending them can greatly improve their ability to model more complex products in a concise and intuitive way. This principle applies equally to typed feature modelling: in standard FM, we could represent the values of a domain as a set of subfeatures that have an alternative relationship with the parent feature. However, this approach is not practical in reality, as it would quickly lead to a cluttered feature diagram. Moreover, it is not possible for features that are associated with infinite domains.

Various groups of authors proposed the inclusion of attributes in feature models to capture further information about the characteristics of features. Benavides et al. (2005) were the first to propose a full notation for extended feature models. In this notation, attributes can take on the value of a domain. These domains can encompass (ranges of) infinite domains, but the notation lacks an intuitive way to represent user-defined types composed of a large number of enumerated values. Therefore, it would not be possible to represent complex concepts, like *Country* or *CreditNotation* of the CDM standard, as attributes. Without our proposed extension, the only option would be to represent each potential value as a subfeature in an alternative relationship, as discussed in the previous paragraph. The ability to define complex types and depict them in a comprehensible legend ensures the possibility of representing complex products compactly and intuitively. Attributes are thus a great way to represent measurable characteristics of features, but are not suitable to represent complex features in a concise way.

The idea of representing each attribute as a subfeature and associating it to the desired type (such as integer) was previously introduced by Bednasch (2002). A collection of attributes can then be modeled as a number of subfeatures. While this proposal comes close to our proposed extension, it does not encompass the possibility to define new types. By contrast, our proposed extension allows us to associate several features with the same defined type. This can greatly increase the compactness of the diagram and is essential to represent products consisting of complex features that take on the same values in a concise way. To the best of our knowledge, our extension is the first to allow features to be associated with types in an intuitive way.

Next to associating features with types, Hein et al. (2000) introduced the idea of relationship types. They defined several different typed relationships, such as the *consists of* relationship, to increase the expressivity of feature modeling and make them more applicable for industrial applications.

7. Conclusion & Future Work

While feature models have proven to be very useful in software product development, they have limitations to represent complex products in diverse domains like finance. To address this, we proposed an extension called typed feature modeling. This extension enhances the expressiveness of feature modeling by allowing features to be associated with types that cover specified or infinite domains. We have demonstrated the effectiveness of our extension by representing complex financial products that conform to the industry standard as a typed feature model. While several extensions of feature modeling exist, our proposed approach stands out due to the possibility to define complex types for complex features that can take on a large number of possible values. By presenting these types in a comprehensible legend, we make sure that the intuitive nature of feature modeling remains. In this way, our approach is crucial to represent the financial use case in a concise way.

Furthermore, we presented an interactive tool

for typed feature modeling and implemented the financial use case within it. We have illustrated how implementing the CDM standard in our TFM-IDP tool can assist users in efficiently managing their financial products and increasing their understanding of the standard itself.

In the future, we aim to enhance our feature modeling tool by incorporating additional extensions, such as cardinality-based feature modeling and extended feature modeling. Moreover, we intend to integrate various analysis operations into the tool for typed feature modeling. These operations will help users to effectively verify and validate the correctness of their typed feature diagrams.

In summary, this work contributes to the advancement of feature modeling by introducing typed feature modeling and demonstrating its potential for diverse domains. The presented tool and financial use case serve as practical demonstrations of the benefits and effectiveness of the proposed extension.

Acknowledgments

This research received funding from the Flemish Government, through Flanders Innovation & Entrepreneurship (VLAIO, project HBC.2022.0477) and under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme.

References

Aerts, B., Deryck, M., & Vennekens, J. (2022). Knowledge-based decision support for machine component design: A case study. *Expert Systems with Applications*, *187*, 115869. https://doi.org/10.1016/j.eswa.2021.115869

Batory, D. (2005). Feature models, grammars, and propositional formulas. In H. Obbink & K. Pohl (Eds.), *Software product lines* (pp. 7–20). Springer Berlin Heidelberg.

Bednasch, T. (2002). *Konzept und implementierung eines konfigurierbaren metamodells für die merkmalmodellierung* (Master's thesis). Fachhochschule Kaiserslautern, Standort Zweibrücken, Germany.

Benavides, D., Trinidad, P., & Ruiz-Cortés, A. (2005). Automated reasoning on feature models. In O. Pastor & J. Falcão e Cunha (Eds.), *Advanced information systems engineering* (pp. 491–503). Springer Berlin Heidelberg.

Carbonnelle, P., Bogaerts, B., Vennekens, J., & Denecker, M. (2022). Interactive Configuration Problems in Observable Environments, 8.

Carbonnelle, P., Deryck, M., Vennekens, J., et al. (2019). An interactive consultant. *BNAIC, Date: 2019/11/06-2019/11/08, Location: Bruxelles*.

Carbonnelle, P., Vandevelde, S., Vennekens, J., & Denecker, M. (2023). Interactive configurator with fo(.) and idp-z3.

Common domain model (cdm) (tech. rep.). (2020). International Swaps and Derivatives Association (ISDA). https://www.icmagroup.org/Regulatory-Policy-and-Market-Practice/repo-andcollateral-markets/fintech/common-domain-model-cdm/

Czarnecki, K., Helsen, S., & Eisenecker, U. (2004). Staged Configuration Using Feature Models. In R. L. Nord (Ed.), *Software Product Lines* (pp. 266–283). Springer Berlin Heidelberg.

Czarnecki, K., Helsen, S., & Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization [Publisher: Wiley Online Library]. *Software process: Improvement and practice*, *10*(1), 7–29.

De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., & Denecker, M. (2018). Predicate logic as a modeling language: The IDP system. In M. Kifer & Y. A. Liu (Eds.), *Declarative Logic Programming: Theory, Systems, and Applications* (pp. 279–323). ACM. https://doi.org/10.1145/3191315.3191321

Denecker, M., & Vennekens, J. (2008). Building a Knowledge Base System for an Integration of Logic Programming and Classical Logic [Series Title: Lecture Notes in Computer Science]. In M. Garcia de la Banda & E. Pontelli (Eds.), *Logic Programming* (pp. 71–76). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-89982-2_12

Deryck, M., Devriendt, J., Marynissen, S., & Vennekens, J. (2019). Legislation in the knowledge base paradigm: Interactive decision enactment for registration duties. *Proceedings of the 13th IEEE Conference on Semantic Computing*, 174–177.

Hein, A., Schlick, M., & Vinga-Martins, R. (2000). Applying feature models in industrial settings. *Software Product Lines: Experience and Research Directions*, 47–70.

Jordens, J., Vandevelde, S., Van Doninck, B., Witters, M., & Vennekens, J. (2022). Adhesive selection via an interactive, user-friendly

system based on symbolic AI. *Proceedings of CIRP DESIGN 2022*.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990a). *Feature-oriented domain analysis (FODA) feasibility study* (tech. rep.). Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990b). *Feature-oriented domain analysis (foda) feasibility study* (tech. rep.). Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Northrop, L. (2008). *Software product lines essentials* (tech. rep.). Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

Riebisch, M., Böllert, K., Streitferdt, D., & Philippow, I. (2002). Extending feature diagrams with uml multiplicities. *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, *23*, 1–7.

Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., & Bontemps, Y. (2007). Generic semantics of feature diagrams. *Computer Networks*, *51*(2), 456–479. https://doi.org/https://doi.org/10.1016/j.comnet.2006.08.008

Van Hertum, P., Dasseville, I., Janssens, G., & Denecker, M. (2017). The KB paradigm and its application to interactive configuration. *Theory and Practice of Logic Programming*, *17*(1), 91–117.

Vandevelde, S., Callewaert, B., & Vennekens, J. (2022). Interactive feature modeling with background knowledge for validation and configuration. *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume B*, 209–216.

Vandevelde, S., Jordens, J., Van Doninck, B., Witters, M., & Vennekens, J. (2022). Knowledge-based support for adhesive selection. In G. Gottlob, D. Inclezan, & M. Maratea (Eds.), *Logic programming and nonmonotonic reasoning* (pp. 445–455). Springer International Publishing.

Vlaeminck, H., Vennekens, J., & Denecker, M. (2009). A logical framework for configuration software. *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 141–148. https://doi.org/10.1145/1599410.1599428