Model Construction with External Constraints: An Interactive Journey from Semantics to Syntax

Mikoláš Janota[1], Victoria Kuzina[2], and Andrzej Wąsowski[2]

[1] Lero, University College Dublin, Ireland
mikolas.janota@ucd.ie
[2] IT University of Copenhagen, Denmark
{victoria,wasowski}@itu.dk

Abstract. Mainstream development environments have recently assimilated guidance technologies based on constraint satisfaction. We investigate one class of such technologies, namely, interactive guided derivation of models, where the editing system assists a designer by providing hints about valid editing operations that maintain global correctness. We provide a semantics-based classification of such guidance systems and investigate concrete guidance algorithms for two kinds of modeling languages: a simple subset of class-diagram-like language and for feature models. Both algorithms are efficient and provide exhaustive guidance.

1 Introduction

Modern modeling and development environments, like Rational® Software Modeler, Visual Studio®, and Eclipse™ embrace interactive support. They offer context-sensitive hints of valid editing operations, such as name completion. These techniques are somewhat limited though. Proposed operations are locally correct with respect to a syntax definition, a metamodel, or a type system, but they do not guarantee global correctness, in the sense of satisfying more complex combinatorial constraints beyond the type system—a need more often seen in domain specific modeling languages (DSMLs). Recently, new techniques [24,11,22,20] have been proposed that increase adaptivity of guidance and strengthen the correctness guarantees.

All these techniques recognize the strength of interactive guidance as opposed to mere validation of constraints or batch synthesis of models without human intervention. Many constraints cannot be meaningfully validated on incomplete models. Effectively, validation provides feedback too late, when the model is ready, the errors are present and substantial revisions might be needed. Batch synthesis in turn favours purely mechanical constraint satisfaction, failing to uncover deep domain knowledge to achieve clarity characteristic to human-made artifacts. Interactive guidance is the midway path between the batch synthesis and validation. It is an incremental model transformation [6,19], which by providing early feedback eliminates the risk of inconsistencies. Still it puts the modeler in control, who shapes the model herself, striving for clarity and precision.
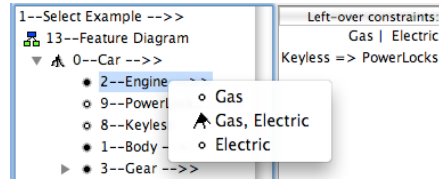
Fig. 1. Guidance with context menus (a screen from our prototype for feature models)

The aim of our work is twofold: 1) to advance the understanding of interactive model derivation from the semantics perspective and 2) to develop guidance algorithms for concrete modeling languages. Our contributions comprise:

– A rigorous definition of interactive model derivation, and a classification of derivation processes aligned with the conceptual modeling hierarchy.
– Identification of efficient and reliable existing technology supporting soundness preserving instantiation of a nontrivial subset of class diagrams.
– Novel algorithms for completeness-preserving and for semantics-preserving derivation of feature models. Crucially, our algorithms are terminating, complete and execute efficiently.
– A definiton of expressive power for guided derivation algorithms in general, and a corresponding evaluation of expressiveness for our new algorithms.

Our primary audience are builders of modeling tools in general and in particular of platforms for model driven engineering with DSMLs. We believe that the topic is also relevant for language designers and for researchers studying semantics of modeling languages and model transformation.

Outline. Section 2 introduces interactive model derivation, and its classification. Section 3 discusses soundness preserving derivation for class-diagram-like DSMLs. In Section 4 we recall basic background about feature models, used subsequently in Sections 5–6 to study completeness- and semantics-preserving derivation. We briefly describe our prototype in Section 6.3 and relate to existing literature in Section 7. We summarize and discuss future work in Section 8.

2 Interactive Model Derivation

Figure 1 shows a screen of our prototype model editor for feature models. A user has just asked for editing the Engine node and a list of suggestions has opened. Here, three choices are proposed, namely to create an or-group comprising Electric and Gas or append any of the features individually as an optional subfeature. In general, guidance is provided by offering valid editing steps. As the number of ways to proceed is typically large, the operations that are offered apply to a context determined by the user, e.g., a feature node, a class, or a state.

Figure 2 shows an overview of the process. The designer queries the editor for possible valid edit operations, the editor provides a list of such and the designer chooses one. The editor executes the operation and waits for the next input.
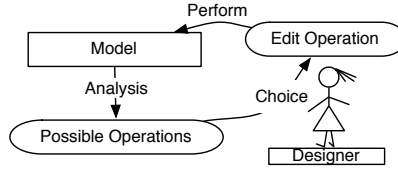
Fig. 2. An overview of the interactive model derivation process

How does the editor know which operations are valid? The operations are chosen so that they do not violate imposed constraints. The simplest constraints come from syntax and typing rules (well-formedness), the more intricate ones from combinatorial aspects of the domain, especially for DSMLs. A reasoning algorithm translates knowledge from constraints to an accessible form, such as a list of edit operations. Below we write VALID-OPERATIONS to denote such a reasoning algorithm. Naturally, even though we use a single name for all of them, the concrete algorithms differ depending on use cases and modeling languages.

Consider a modeling language $\mathcal{M}$ with the syntax defined using a metamodel. The semantics of $\mathcal{M}$ is given by assigning to each model $M : \mathcal{M}$ a logic description denoted $[\![M]\!]$. VALID-OPERATIONS returns a set of editing operations, which are mappings from models to models.

Naturally, a transformation applied to a model yields a change in the pertaining semantics. In this article we focus only on algorithms that identify transformations that *refine* the model. Semantically, a refinement is characterized by strengthening the semantics of the model (making the model more restrictive). Intuitively, any instance of the refined model is also an instance of the original one. More formally, for each $O \in$ VALID-OPERATIONS(M) the following implication holds: $[\![O(M)]\!] \rightarrow [\![M]\!]$.

Focusing on refinement is motivated by the goal of a *gradual* construction of the model. Hence we will be concerned with *sequences* of editing operations, most often with sequences that begin with an empty model and lead to a fully constructed one. A limit of such sequences of refinements is typically a point beyond which refinement is not desirable. For instance, for a model with semantics expressed as a set, a singleton set is a possible limit.

In general, we aim for guidance algorithms to propose enough operations so all such limits are attainable, not preventing the user from deriving any useful models. We will call this property *exhaustiveness of advice*. In a similar spirit we will require *validity of advice*, i.e., that *no* sequence of operations selected from the proposed choice is leading to an invalid model.

Apart from the refinement requirement, we also consider a constraint, φ, capturing additional domain knowledge. We have identified three different model derivation scenarios differentiated by what role the constraint φ has. The first scenario is mostly known under the term *modeling*: instantiation of metamodels. It aims at constructing a single instance of a metamodel, for example an object diagram, and can be seen as a gradual narrowing of a set of possible instances to
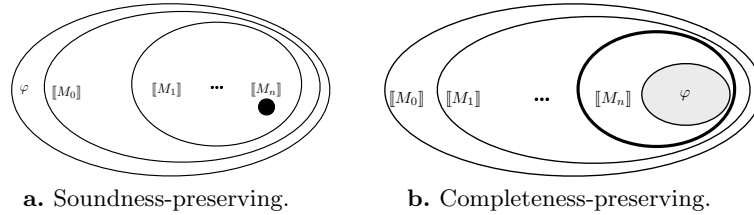
a. Soundness-preserving.   **b.** Completeness-preserving.

Fig. 3. Limits of evolution in different classes of model derivation

a single instance of interest. We shall call such a process a *soundness-preserving model derivation*. Consider the visualization in 3a. The designer seeks for a particular instance that satisfies φ and each of the transformations brings her closer to the desired solution. The ellipses represent intermediate models and the goal instance is depicted as ●. A very simple example of soundness-preserving derivation is interactive configuration [10,14] of feature-models [5], where one specific product, a configuration, from a domain is selected. Other applications include instantiation of highly combinatorial DSMLs as seen in [24].

We say that $\{O_i\}_{i=1..n}$ is a sequence of valid operations and $\{M_i\}_{i=0..n}$ is its associated sequence of intermediate models iff $O_i \in$ VALID-OPERATIONS(M_{i-1}) and $M_{i+1} = O_{i+1}(M_i)$. For the derivation to be soundness-preserving all the intermediate models must be within the bounds of the constraint φ which represents the domain (φ can amount to a metamodel and a set of OCL constraints). More specifically, $[\![M_i]\!] \rightarrow \varphi$ holds for all i.

In soundness-preserving derivation any VALID-OPERATIONS algorithm has the *exhaustive advice* property iff it enables deriving all valid instances, i.e., for any model M respecting the constraints ($[\![M]\!] \rightarrow \varphi$) there exists a sequence of valid operations and its associated sequence of models $\{M_i\}_{i=1..n}$ and $M = M_n$.

The second model derivation scenario takes us from modeling to metamodeling: modeling of domains, as opposed to single artifacts. Here the model derivation refines the entire universum of models to a *subset* that describes a certain domain as precisely as possible in the given language. We shall call such model derivations *completeness-preserving* since they focus not on finding a specific valid instance (as in soundness-preserving), but on *removing all unsuitable* instances. Figure 3b gives an intuition: The derivation starts with a very loose model M_0 and gradually approaches a model that tightly over-approximates φ.

Completeness-preserving model derivation is less common than the soundness-preserving one, in the same way as metamodeling is a less common activity than modeling. It can be used to reverse-engineer models from logic descriptions of constraints—a functionality particularly needed in modeling of non-software products, where designers are trained in product design but not in the kind of modeling commonly seen in IT. What is perhaps more interesting, it can be used to refine existing models of domains on which new constraints are being imposed. Such constraints may arise from the context of the model. For instance, in our
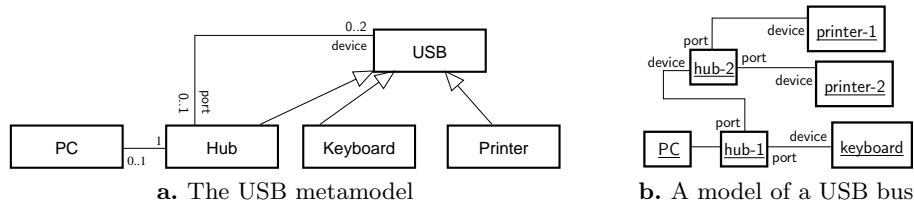
a. The USB metamodel   **b.** A model of a USB bus

Fig. 4. The USB language and its instance. Inspired by [23]

previous work we have investigated how constraints in an architecture model propagate to constraints of the feature model [13].

Formally, for any sequence of valid operations $\{O_i\}_{i=1..n}$ and its sequence of intermediate models $\{M_i\}_{i=0..n}$ it holds that $\varphi \rightarrow [\![M_i]\!]$ for any i. We will say that a completeness-preserving VALID-OPERATIONS algorithm has the *exhaustive advice* property iff it enables to derive all the models that are weaker than φ, i.e., for any M, such that $\varphi \rightarrow [\![M]\!]$, there exists a sequence of valid operations and their associated intermediate models $\{M_i\}_{i=0..n}$ with $M = M_n$.

The final scenario is a very well known class of model transformations that lies in the intersection of semantics- and completeness-preserving cases: the class of *semantics-preserving* transformations or *refactorings*. Any refactoring O has the property that $[\![O(M)]\!] = [\![M]\!]$ and for any sequence of valid refactorings $\{O_i\}_i = 1..n$ with the intermediate models $\{M_i\}_{i=0..n}$ we have that $[\![M_i]\!] = [\![M_{i+1}]\!]$. We easily see that sequences of operations that satisfy this condition are both semantics- and completeness preserving.

Below we detail interactive derivation algorithms for concrete languages and corresponding use cases, one for each class out of the three defined above.

3 Soundness-Preserving Model Derivation

We have characterized soundness-preserving model derivation as an editing process proceeding via a sequence of models with decreasing space of possible completions, so that for any M_i we have that $[\![M_i]\!] \rightarrow \varphi$. Below we consider an instance of this problem for a DSML describing USB buses, defined by the metamodel in Fig. 4a. Each model in this language describes a USB bus comprising a PC to which a hub is connected. Each hub has two ports. Each port can be used as a socket to which another USB device is connected; either another hub, or a keyboard, or a printer. Figure 4b shows a simple model.

As it is often the case, the metamodel as such cannot fully express the syntactic restrictions on the model, so it is accompanied by a set of constraints. The constraints are typically formalized, but for the sake of simplicity we express them in plain English here: (C1) each model must contain exactly one instance of PC, (C2) every USB device is connected to a port or to the PC instance, and (C3) every bus has a keyboard connected or a free port to connect one. We want to assist the designer in creating USB models satisfying these constraints.
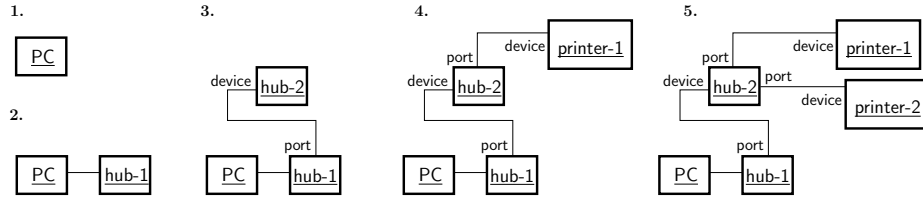
Fig. 5. Stages of editing the example from 4b.

Out of many derivation processes possible, in this work we focus on syntactically monotonic ones, i.e., beginning with an empty model and refining the model syntactically by adding new elements step by step. The new elements must be added to the existing ones, which means that whenever we instantiate a new device on the bus, we need to connect it to one of the existing ports.

Figure 5 shows a derivation of a simple model. We start with an empty diagram. As a consequence of (C1) and (C2), the editor[3] proposes a single possible edit step: the instantiation of a PC node. Constraint (C2) yields a possible set of choices in the second step: instantiate and connect a hub (step 2). In step 3 the tool suggests instantiating and connecting a USB device object to any of the two ports of hub-1. All three types of devices are offered (a hub, a keyboard, and a printer). We choose the hub, and then similarly two printers in steps 4–5. The final step of the example is perhaps the most interesting one. We are in a state, in which only one port is free, in the hub-1 object. Possible edit steps for this port are to instantiate either a hub or a keyboard. Observe that a printer object cannot be suggested at this point as connecting a printer would exclude the possibility of connecting any other device to the bus. Consequently constraint (C3) would be violated. Selecting a hub is safe as it still leaves the possibility of connecting a keyboard later. Select a keyboard to obtaining the model in Fig. 4b.

### 3.1 Soundness-Preserving Derivation as Modular Configuration

Our USB language is in fact an instance of the *modular configuration problem*. In [23] algorithms for monotonic syntactic derivation of instances of these problems are presented, guaranteeing validity and exhaustiveness of advice.

Modular configuration problems can be equated to a decidable class of object-diagram-like languages with propositional constraints interpreted over associations. More precisely, a constraint associated with a class is interpreted separately for each of its objects, and can refer only to variables of the object itself, and to variables of objects directly reachable via a navigation step from that object. No quantification or iteration is allowed in constraints, and all associations have bounded multiplicity. While the metamodels can contain cycles, only acyclic and connected instances are allowed. As the constraints are interpreted over *instantiations* of associations they can encode many inductive properties normally reserved for first order logics—like having a keyboard on each bus.

---

[3] Such an editor is feasible, as [23] demonstrates.
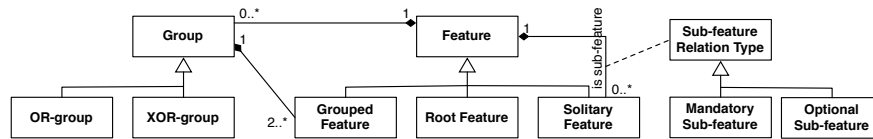
a. The metamodel
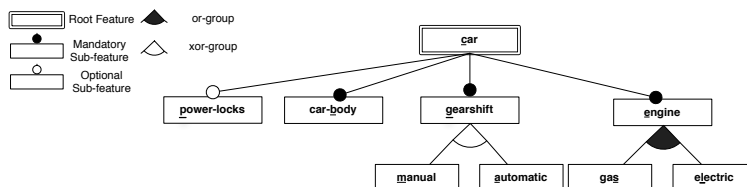
b. An example of a feature diagram, inspired by [7]

Fig. 6. The Language of Propositional Feature Diagrams

4 Background: Propositional Feature Models

Let us recall the language of feature models, exploited as an example in the upcoming sections. Feature models [15] are used to systematically describe variability and commonality in product line engineering [4]. In a nutshell, a feature corresponds to a specific functionality of a system. A feature model records available features, together with constraints and dependencies relating them.

A variety of feature diagram languages is found in the literature, mostly with propositional semantics [21]. We should note, however, that other semantics exist, for instance using grammars [2], higher-order [12] and probabilistic [8] logic. In this article we operate on the combinatorial core of feature models, the propositional models [7]. A propositional feature model comprises a *feature diagram* and a *constraint*. The diagram is the centerpiece. It records the features and dependencies in a graph-based notation. An additional constraint is appended if it cannot be expressed in the diagrammatic language itself.

A feature diagram organizes features in a tree, containing a node for each feature. A child node, or a *sub-feature*, is either *optional*, *mandatory*, or *grouped*. A grouped feature belongs either to an *or-group* or an *xor-group* with other sub-features of the same parent. Fig. 6a shows the metamodel of this language.

Apart from hierarchically organizing the features, the purpose of the diagram is to determine which combinations of features, so-called *feature configurations*, are permitted. The root feature must be present in all configurations. A sub-feature must not be selected into a configuration not containing its parent. A mandatory feature is required by its parent, whereas an optional one is not. From features grouped in an or-group (resp. xor-group) at least one (resp. exactly one) must be selected whenever the parent is selected. The feature diagram in Fig. 6b describes possible configurations of a car. Each car must have a body, gearshift, and engine; an engine is electric *or* gas (selecting both corresponds to a hybrid engine); a gearshift is *either* automatic or manual.

The semantics $[\![\cdot]\!]$ of a propositional feature diagram is defined as the set of permitted configurations. Symbolically such a set is identified with the set of satisfiable assignments of its characteristic formula in propositional logic. The formula is created by binding a single variable to each feature, and translating the syntax of a feature diagram in a natural manner. More specifically, the formula is a conjunction of the formulas yielded by the individual constructs, which are defined as follows. The root r is always selected, hence yields the formula r. Features $f_1, \ldots, f_n$ grouped in an or-group under the parent p yield the formula $f_1 \vee \cdots \vee f_n \leftrightarrow p$. An xor-group $f_1, \ldots, f_n$ yields $f_1 \vee \cdots \vee f_n \leftrightarrow p$ and additionally mutually disables the grouped features: $\bigwedge_{i \neq j} \neg(f_i \wedge f_j)$. Finally, each optional feature c with the parent p yields the formula $c \rightarrow p$ and each mandatory feature m yields $m \leftrightarrow p$. The formula representing the semantics of the model in Fig. 6b is: $c \wedge (p \rightarrow c) \wedge (b \leftrightarrow c) \wedge (g \leftrightarrow c) \wedge (e \leftrightarrow c) \wedge (g \leftrightarrow m \,\mathsf{xor}\, a) \wedge (e \leftrightarrow s \vee l)$. In order to obtain semantics of the entire feature model (M, ψ) we lift the semantics of the feature diagram M and conjoin it with the constraint ψ: $[\![(M, \psi)]\!] = [\![M]\!] \wedge \psi$.

Some authors extend the diagram language with *excludes* and *requires* edges that may cross the tree hierarchy. Such extended diagrams are technically fully expressive, in the sense that they can encode any propositional formula [21]. However, when it comes to expressiveness, there is little difference between the use of an explicit constraint (ψ) and the two additional constructs. Indeed, our results from Sections 5–6 can easily be extended to account for the two.

5 Completeness-Preserving Model Derivation

Section 2 characterizes completeness-preserving derivation as a sequence of transformations in which the semantics of intermediate models covers the given constraint at all times, so $\varphi \rightarrow [\![M_i]\!]$ for all intermediate M_i. In this section we develop a completeness-preserving VALID-OPERATIONS algorithm for a specific case—derivation of feature diagrams, or feature models without the constraint.

5.1 Analysis

Consider a scenario in which instead of a feature model of a car we are given a collection of constraints; for example that a car cannot have both manual and automatic gearshift. Let φ be a conjunction of such constraints. The constraints are proposed by various stakeholders involved in car development. Now a modeler needs to construct a feature diagram that faithfully captures φ.

How do we characterize a diagram that *faithfully* captures a constraint? On the one hand, the desired feature diagram should not be more restrictive than φ — otherwise it would have ruled out configurations that do not violate the constraints. On the other hand, the diagram should capture as much of the constraints as possible, ideally precisely those that are configurations of φ. In general, since a propositional feature diagram alone is not expressive enough to capture arbitrary constraints, we strive for capturing strongest possible overapproximations of φ, hoping that these would most clearly represent the intrinsic

VALID-OPERATIONS(M, n, φ) : set of operations

    ▷ M is a (partially constructed) feature diagram
    ▷ n is a node in M, present iff M is nonempty

1 **if** M is empty **then return** $\{\mathsf{Root}(r) \mid \varphi, M \models r\}$

2 $solitary \leftarrow \{\mathsf{Mandatory}(n, c) \mid \varphi, M \models c \leftrightarrow n$ and c not instantiated in $M\}$
       $\cup \, \{\mathsf{Optional}(n, c) \mid \varphi, M \models c \rightarrow n$ and c not instantiated in $M\}$

3 $groups \leftarrow \{\mathsf{OrGroup}(n, m_1 \ldots m_k) \mid n \leftrightarrow \varphi, M \models \bigvee_{i \in 1 \ldots k} m_i,$
       $k > 1$ and all m_i are not instantiated in $M\}$
       $\cup \, \{\mathsf{XorGroup}(n, m_1 \ldots m_k) \mid \varphi, M \models n \leftrightarrow \bigvee_{i \in 1 \ldots k} m_i \wedge \bigwedge_{i \neq j} \neg(m_i \wedge m_j),$
       $k > 1$ and all m_i are not instantiated in $M\}$

4 $refine \leftarrow \{\mathsf{RefineOR\text{-}group}(n, m_1, \ldots, m_k) \mid$
       $\{m_1, \ldots, m_k\}$ is an or-group of p in M and $\varphi, M \models \bigwedge_{i \neq j} \neg(m_i \wedge m_j)\}$
       $\cup \, \{\mathsf{RefineOptional}(n) \mid n$ is an optional child of p in M, and $\varphi, M \models n \rightarrow p\}$

5 **return** $solitary \cup groups \cup refine$

Fig. 7. Computing completeness-preserving operations for propositional feature models

combinatorial structure of φ. So we seek for a diagram M that conforms to the metamodel (Fig. 6a) and whose semantics is weaker than the provided constraints, so: $\varphi \rightarrow [\![M]\!]$, and which is refined as much as possible.

In another use case, a new constraint should to be imposed on an already existing feature model. Consider again the example of Fig. 6b. The designer adds a feature keyless-entry which requires power-locks. So an additional constraint keyless-entry $\rightarrow$ power-locks arises. The editor computes that the keyless-entry can be a sub-feature of power-locks and if the user opts for this operation, the implication can be removed. The editor also computes that keyless-entry can be made a sub-feature of car, due to transitivity of implication. However, if the user chooses this operation, the formula keyless-entry $\rightarrow$ power-locks is not captured in the diagram and must be appended as a new constraint to the model.

Similarly as in Sect. 3 we choose to work with syntactically monotonic derivations. For feature diagrams it means that our editing steps add new syntactic constructs to existing parts of the diagram, as opposed to modifying them. Strictly speaking, we do supply a few simple non-monotonic steps for usability reasons. These extensions however do not influence the expressiveness of the available transformations when it comes to validity and exhaustiveness of advice.

5.2 Completeness-Preserving Derivation of Feature Diagrams

The process starts with a feature diagram M satisfying $\varphi \rightarrow [\![M]\!]$. The algorithm VALID-OPERATIONS (see Fig. 7) suggests editing operations. As we do not want to overwhelm the user with an inundation of possible operations, the algorithm provides editing operations for a specific node, denoted n in Fig. 7.

For the sake of clarity, let us first assume that we have a magic oracle determining queries of the form $\varphi, M \models \psi$, meaning that assuming the constraint φ

and the semantics of diagram M, the proposition ψ holds: $\varphi \wedge [\![M]\!] \rightarrow \psi$. The algorithm mirrors, in a sense, the semantics of feature diagrams. For a given node n, it needs to find *all* possible mandatory sub-features, i.e., features c that are not in the diagram yet and for which the bi-implication $n \leftrightarrow c$ holds (line 2). Other primitives of the diagram meta-model are covered analogously.

Line 4 is somewhat different from the preceding ones. Whereas the previous lines *add* to the diagram, this one *modifies* the existing constructs. Any or-group may be refined to an xor-group if a sufficient constraint is provided, i.e., mutual exclusion of the grouped features. Analogously for the optional feature.

An implementation of the oracle $\varphi, M \models \psi$ is actually presented in [7], where a technique is developed for compiling a boolean formula into a so-called *feature graph*. When using a feature graph as an oracle, necessary queries can be evaluated in time polynomial in the number of features. Indeed, most of the queries used in the above algorithm can be established during a single traversal of the feature graph, which makes this algorithm very efficient. All this despite that most of the queries would require solving NP-complete problems, when applied to a textual representation of φ. Efficiency during derivation is possible, because the hardness of the problem is shifted to a startup phase, when the feature graph is generated (generation is NP-hard). The generation runs only once per an editing session (as opposed to at every editing step) and it can be efficiently implemented using binary decision diagrams [3].

Valid advice. The algorithm of Fig. 7 suggests only the operations that are completeness-preserving. More specifically, for any returned operation O and a model M s.t. $\varphi \rightarrow [\![M]\!]$ it holds that $\varphi \rightarrow [\![O(M)]\!]$. The proof of this fact proceeds by comparing the algorithm to the semantics defined in Sect. 4.

Exhaustive advice. This algorithm provides exhaustive advice, meaning that any feature diagram weaker than φ can be derived from the empty diagram in a finite number of steps by solely applying operations suggested by the algorithm. For the lack of space we do not provide the whole proof, but only illustrate the basic idea. Assume that an adversary gives us a diagram M and a constraint φ such that $\varphi \rightarrow [\![M]\!]$. Our goal is to re-construct M using the editor. We start at the root and descend towards the leaves. First, let us look at the root r: Because it is a root and $[\![M]\!]$ is weaker than φ, we have $\varphi \rightarrow [\![M]\!]$ and $[\![M]\!] \equiv r \wedge \psi$ for some ψ (see the definition of $[\![\cdot]\!]$ in Sect. 4). Together this yields that $\varphi \rightarrow r$, so r will be offered to the user (Line 1) as an option when she starts editing.

Let c be any optional sub-feature that has not been reconstructed so far and p be a re-constructed parent of c. We have $[\![M]\!] \equiv (c \rightarrow p) \wedge \psi$ and hence by the same reasoning as before $(\varphi \wedge p) \rightarrow c$, therefore c will be offered to the user as an optional child of p. The rest of the proof is carried out in an analogous fashion.

On the side, note that the algorithm enables us to start the derivation from an arbitrary diagram, not necessarily an empty one. Hence a stronger form of the exhaustive advice property can be shown for it: any diagram M' such that $([\![M]\!] \wedge \varphi) \rightarrow [\![M']\!]$ and that respects the parent-child relation imposed by M, can be derived from M using the suggested steps.

$$(M, \psi) \xrightarrow{\hspace{3cm}} (M', \psi')$$

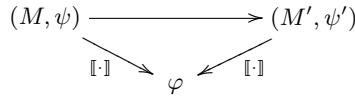$$\llbracket \cdot \rrbracket \searrow \quad \swarrow \llbracket \cdot \rrbracket$$

$$\varphi$$

Fig. 8. A refactoring step for a feature model (M, ψ): $\llbracket M' \rrbracket \to \llbracket M \rrbracket$, $\llbracket \psi \rrbracket \to \llbracket \psi' \rrbracket$

6 Semantics-preserving Editing Feature Models

Recall that a propositional feature model comprises a diagram and a boolean formula. The semantics of this pair is defined as a conjunction of the semantics of the diagram and the formula. For feature models, semantics-preserving transformations are those that preserve the satisfying assignments of that conjunction. One may imagine an amplitude of transformations that are semantic-preserving. Below we describe the type of transformation that we have targeted in our work.

6.1 Feature Diagram Refinement by Model Refactoring

Let us replay the scenario from Sect. 5.1, this time considering both a diagram *and* a constraint. As previously, the goal is to approximate the given constraint φ with a feature diagram, which is not likely to capture the constraint completely leaving a remainder in the second component of the model.

The editing process starts with the empty diagram and φ as the constraint component. The constraint is weakened gradually, whenever the diagram is refined. The semantics of the entire model is preserved throughout. Figure 8 schematically depicts what we expect from each single editing operation.

The algorithm in Sect. 5.2 lets us assist the user with refining the diagram while keeping its semantics weaker than φ. In here, the editor should also provide for automatic transformation of the constraint. For instance, if the constraint contains the formula $c \to p$ and the user makes c to be an optional sub-feature of p, one would predict that the formula in question will be removed from the constraints as the sub-feature relation already captures the implication (see Sect. 4). These simplifications are important for the user as they give her an intuition of how much of the left-over constraint is alread incorporated into the diagram.

We should note that there is no single obvious notion of simplification. Simplification cannot be defined as weakening. Changing $x \wedge y$ to x weakens and arguably simplifies, whereas changing x to $x \vee y$ weakens but hardly classifies as simplification. Hence, by simplification we mean an operation that reduces the size of the syntactic representation of the formula and keeps as much of the original structure as possible while preserving the pertaining semantics.

6.2 Adapting Constraint for Diagram Refinement

The diagram is refined by the user by choosing an operation from a set suggested by VALID-OPERATIONS (see Fig. 7). It is clear how these operations affect the diagram. Here we explain how to adapt the left-over constraint.

Our prototype accepts the input in the Conjunctive Normal Form (CNF) with enhancements that enable writing for instance IFF f_1 f_2 for the two clauses $\neg f_1 \vee f_2$ and $\neg f_2 \vee \neg f_1$. Each editing operation results in a set of clauses $\mathcal{C}$ that represents the constraints and a boolean formula ψ that corresponds to the semantics of the current feature diagram. Hence, the problem now is to simplify $\mathcal{C}$ under the assumption that ψ holds.

The first simplification we perform reduces $\mathcal{C}$ to such form that no clause $c \in \mathcal{C}$ is implied by the conjunct of the rest of the clauses and ψ, i.e., there is no $c \in \mathcal{C}$ such that $\bigwedge_{d \in \mathcal{C} \setminus \{c\}} (d \wedge \psi) \to c$. This reduction is implemented by repeatedly choosing and removing a clause that violates this property until no such clause exists[4]. Binary Decision Diagrams (BDDs) [3] are used to decide the individual implications but a SAT solver could be used as well.

Apart from performing the operations suggested by the algorithm VALID-OPERATIONS (Fig. 7), the user chooses the root of the feature diagram when the derivation process begins. A feature r can be chosen as a root if it is implied by all the other features while assuming φ. The choice of the root r corresponds to setting r to *true*. That lets us simplify $\mathcal{C}$ by repeatedly using the tautologies $(true \vee C) \leftrightarrow true$ and $(false \vee C) \leftrightarrow C$, a technique known as *unit propagation*.

Our motivation for using CNF was that it enables performing simplifications in a rather uniform manner while it is still fully expressive. Many useful constructs, e.g., f_1 *excludes* f_2, map directly to their clausal form ($\neg f_1 \vee \neg f_2$ in this case). Nevertheless, both types of simplification could be extended to operate on a general boolean form. Instead of operating on a set of clauses, one would operate on a set of boolean expressions. Then, the reduction of this set (first simplification) can be done in the same fashion. The unit propagation of the root would be replaced by substituting *true* for the root in each expression and subsequently applying standard boolean tautologies, such as $(a \wedge true) \leftrightarrow a$.

6.3 Implementation

We have embedded the algorithm VALID-OPERATIONS (see Fig. 7) into an Eclipse plug-in that lets the user load a file with a constraint and derive the desired feature diagram. The editing process follows the principles described in Sect. 2 and the simplification of the constraint is performed as described in Sect. 6.2. The constraint, in Conjunctive Normal Form, is translated to a BDD representation that is further used to compute the feature graph — a structure that answers queries for VALID-OPERATIONS (see [7] for details).

We have applied the implementation to models described in the literature, including the largest publicly available example known to us, a model of an ecommerce system from [18], for which the editor responded with imperceptible time delays. Details in [17]. The main bottleneck of our solution lies in the algorithm of [7], and its prime implicants computation. Preliminary experiments indicate that on typical formulae this algorithm scales up to 200-300 variables.

---

[4] Heuristics could be used for deciding which clauses should be removed first. For instance, one could start with the longer ones.

7 Related Work

The algorithm that compiles propositional constraints to a feature graph, which is used as an oracle in Sect. 5.2, originates in our previous work [7]. The present paper really completes the agenda of loc. cit.

Our work shares many goals with [24,11,22,20]. Still it differs significantly in several respects. First, they mostly focus on algorithms for concrete problems, while we also try to understand problems in a more general framework of preservation classes and advice properties. Second, the practical part of our work focuses on efficient terminating algorithms, with a particular performance gain in use of the offline compilation phase. In contrast, [24,11,22] rely on online query execution using Prolog, which does not even guarantee termination. We admit though that all report efficient execution times in practice, for the constraints they consider.

Third, we achieve exhaustiveness of advice, while the quality of advice is hard to assess in loc. cit. SmartEMF [11] provides a weak form of exhaustive advice, given that it terminates. The algorithm of Sen and coauthors [22] is a very good example of a solution which does *not* guarantee exhaustiveness of advice. In fact the latter algorithm has very little control over the kind of advice. It only provides a fixed number of hints, computed by running a Prolog engine several times on a constraint system representing valid choices. The constraints are permuted randomly in order to allow for finding different solutions.

Concrete modeling languages used in these works differ. While [24,11] focus on class diagram-like metamodels (e.g. Eclipse Modeling Framework), we work with modular configuration problems and feature models. The two languages we have chosen are less expressive than these in some sense, but this is what enables us to provide support that is more robust and automated. In [22] languages expressible using hyperedge replacement grammars are targeted.

SmartEMF [11] does not consider guidance beyond attributes. In particular, there are no hints given about the need to instantiate new objects in the model, which constitute most of the operations in our derivation processes.

Alves et al. [1] discuss refactorings of feature models, introducing a measure of quality for them. While we strongly disagree with this definition (which is basically a weakening relation) we state that semantics-preserving feature model derivation, as presented in here, falls under their definition of refactoring. In [1] a catalog of refactorings is proposed, but no firm claims of its exhaustiveness are made. In that sense our results are stronger, as we have argued that any models can be derived with our editing operations (exhaustiveness of advice). Being incremental, our operations do not enable refactoring *any* model to *any* model though. Such goal would require a mixture of both weakening and strengthening steps. We intend to explore this possibility in future.

Massoni et al. [19] have investigated refactorings that are done in two tiers: at the program (code) level and the object model level. The need for such refactorings arises in the evolution of projects developed under the model-driven paradigm. As in our work, the authors recognize the importance of formally sound refactorings and aim to establish such.

8   Summary and Future Work

Interactive model derivation is a process of constructing models and meta-models in the presence of automatic adaptive guidance on possible modeling steps. We have classified uses of model derivation into soundness-preserving, completeness-preserving and semantics-preserving, relating these classes to modeling and metamodeling activities. Then we have proposed guidance algorithms for two kinds of languages (pointing to existing work in one case). Our development was rigorous, exploiting logic constraints and reasoning algorithms. We have demonstrated the feasibility of such approaches by implementing a prototype for feature models and by testing it on existing examples.

While mainstream tools adopt interactive support technologies, the ultimate open problem in the field remains to be: how to generically and robustly provide exhaustive and valid guidance for an arbitrary language defined by a metamodel (for example in MOF or EMF) and a set of reasonably expressive constraints (like a large decidable subset of OCL). While others approached this problem by retaining an expressive language, but relying on a potentially nonterminating search, we have taken a different route. We considered simpler languages, but also more robust algorithms. In future we intend to investigate possibilities of supporting richer languages.

Even though we emphasize the importance of interactive support, in some cases it is still useful to automatically perform multiple editing operations. This desire poses several interesting problems ranging from HCI issues (how to present a choice of sequences of operations?) to optimization issues (how to efficiently discover sequences of interest?).

References

1. Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos José Pereira de Lucena. Refactoring product lines. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *GPCE*, pages 201–210. ACM, 2006.
2. Don Batory. Feature models, grammars, and propositional formulas. In H. Obbink and K. Pohl, editors, *SPLC '05*, LNCS. Springer-Verlag, 2005.
3. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.
4. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison–Wesley Publishing Company, 2002.
5. Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. fmp and fmp2rsm: eclipse plug-ins for modeling features using model templates. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 200–201, New York, NY, USA, 2005. ACM.

6. Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In R. L. Nord, editor, *SPLC '04*, volume 3154 of *LNCS*. Springer-Verlag, August 2004.

7. Krzysztof Czarnecki and Andrzej Wąsowski. Feature diagrams and logics: There and back again. In Kellenberger [16].

8. Krzysztof Czarnecki and Andrzej Wąsowski. Sample spaces and feature models: There and back again. In *Proceedings of the 12th International Software Product Line Conference, SPLC '08*. IEEE Computer Society, 2007.

9. *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

10. T. Hadzic, R. Jensen, and H. Reif Andersen. Notes on calculating valid domains. Manuscript online http://www.itu.dk/~tarik/cvd/cvd.pdf, 2006.

11. Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wąsowski. Guided development with multiple domain-specific languages. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *LNCS*, pages 46–60. Springer, 2007.

12. Mikoláš Janota and Joseph Kiniry. Reasoning about feature models in higher-order logic. In Kellenberger [16].

13. Mikoláš Janota and Goetz Botterweck. Formal approach to integrating feature and architecture models. In FASE '08 [9].

14. Ulrich Junker. Configuration. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*. Elsevier Science Inc., 2006.

15. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA), feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, November 1990.

16. P. Kellenberger, editor. *Software Product Lines*. IEEE Computer Society, 2007.

17. Victoria Kuzina. Interactive derivation of feature diagrams. Master's thesis, IT University of Copenhagen, 2008. To appear.

18. Sean Quan Lau. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, Dept. Electrical and Computer Engineering, University of Waterloo, Canada, 2006. Available at: http://gp.uwaterloo.ca/files/2006-lau-masc-thesis.pdf.

19. Tiago Massoni, Rohit Gheyi, and Paulo Borba. Formal model-driven program refactoring. In FASE '08 [9].

20. Steffen Mazanek, Sonja Meier, and Mark Minas. Auto-completion for diagram editors based on graph grammars. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Computer Society, 2008.

21. Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceeding of 14th IEEE International Requirements Engineering Conference (RE)*. IEEE Computer Society, 2006.

22. Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Domain-specific model editors with model completion. In P. J. Mosterman, T. Levandowszky, and J. de Lara, editors, *The 2nd International Workshop on Multi-Paradigm Modeling*, 2007.

23. Erik Roland van der Meer, Andrzej Wąsowski, and Henrik Reif Andersen. Efficient interactive configuration of unbounded modular systems. In Hisham Haddad, editor, *SAC*, pages 409–414. ACM, 2006.

24. Jules White, Douglas Schmidt, Andrey Nechypurenko, and Egon Wuchner. Domain-Specific Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains. In *GPCE4QoS*, October 2006.