Vision Paper: Make a Difference! (Semantically)

Uli Fahrenberg, Axel Legay, Andrzej Wasowski

Vision Paper: Make a Difference! (Semantically)*

Uli Fahrenberg[1], Axel Legay[1], and Andrzej Wąsowski[2]

[1] INRIA / Irisa Rennes, France, {ulrich.fahrenberg,axel.legay}@irisa.fr
[2] IT University of Copenhagen, Denmark, wasowski@itu.dk

Abstract. Syntactic difference between models is a wide research area with applications in tools for model evolution, model synchronization and version control. On the other hand, semantic difference between models is rarely discussed. We point out to main use cases of semantic difference between models, and then propose a framework for defining well-formed difference operators on model semantics as adjoints of model combinators such as conjunction, disjunction and structural composition. The framework is defined by properties other then constructively. We instantiate the framework for two rather different modeling languages: feature models and automata specifications. We believe that the algebraic theory of semantic difference will allow to define practical model differencing tools in the future.

1 Introduction

The notion of syntactic difference is well established in software engineering. Textual and graphical algorithms are used to identify differences between text files (source code) and models, and then employed to construct versioning systems, which support comparison and merging of files. *Semantic* difference between models is rarely discussed in the modeling community. This is surprising given the wide recognition of importance of software evolution; semantic difference can support evolution scenarios like bug localization, or incremental verification, and enable model merging that does not fail on ad-hoc syntactic conflicts.

While working on specification theories, within the realm of concurrency and verification, we have observed that many familiar operators on specifications also apply to other models: conjunction – superposition of requirements; parallel composition – structural composition of models; refinement – subtyping, just to mention the most important ones. However the notion of *difference*, as a form of (partial) inverse to the above operators, does not attract nearly as much interest in software engineering.

Our objective is to define and present semantic difference between models in a general fashion. We propose an unambiguous definition of difference which emphasizes its algebraic properties. We instantiate it both for a very simple modeling language, *feature models* [13], and also for the mode complicated language of *automata specifications* [14].

Finally, we also try to explain how difference operators can be used to make formal software development more iterative. It is a common belief that development by stepwise refinement, or use of component algebras, requires using a highly planned and waterfall-like development process. See for example the following quote:

* Supported by MT-LAB: a VKR Centre of Excellence in Modeling of IT Systems

> An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code. Based on the assumption that your mathematical transformations are correct, you can therefore make a strong argument that a program generated in this way is consistent with its specification. [22, p.32]

We will point out uses of difference between models involving flow of information between the stages of the development process and abstraction layers in either way. This allows to run the formal development process in a more agile and iterative manner.

Let us give a teaser of our approach to difference with an extremely simple example: the difference operator for integer numbers. Observe that given two integers t and s, the difference $t - s$ can be defined as the maximum integer x for which $s + x \leq t$. More succinctly: x is a difference of t by s if it holds that for any other integer y:

$$s + y \leq t \quad \text{iff} \quad y \leq x.$$

It is then easy to see that this defines a unique notion of difference. Now observe that we have here defined $t - s$ by *property* rather than *construction*. To show that such a difference actually exists, one has to do more work; but if it does, we already know that it is unique. We will repeatedly use constructions like the above for defining differences with respect to other binary operators and for other objects than integers.

A similar algebraic structure can be uncovered in the area of *software verification*: In programming languages, there is a long established notion of weakest precondition, as the proof obligation on the context of a piece of code which suffices to conclude a given goal [8,11]. Let P be a fragment of imperative code consisting of a number of sequentially composed statements $s_1, \ldots, s_k$. Let the axiomatic semantics for each statement be expressed by a Hoare triple $\{\varphi_i\}s_i\{\psi_i\}$, where φ_i is a precondition and ψ_i is a postcondition, and let ψ be a desired property of the state after executing P. Proving that P is correct, i.e. that $\{\mathsf{true}\}P\{\psi\}$ describes P, amounts to showing that $\mathsf{true} \to \varphi_1, \varphi_1 \to \psi_1, \psi_1 \to \varphi_2, \ldots, \psi_k \to \psi$.

However this may not always be possible, since it enforces correctness regardless of the initial state. Instead it is more reasonable to *synthesize* an assumption X for which $X \to \varphi_1, \varphi_1 \to \psi_1, \psi_1 \to \varphi_2, \ldots, \psi_k \to \psi$. The property X is called a sufficient precondition for P to guarantee ψ. We say that X is the *weakest precondition* if it is also necessary, i.e. if it holds for all formulae Y that

$$Y \to \varphi_1, \varphi_1 \to \psi_1, \psi_1 \to \varphi_2, \ldots, \psi_k \to \psi \quad \text{iff} \quad X \to Y.$$

The precondition X informs the *user* of P on what conditions she has to meet. Dually if the precondition φ for P is fixed by the users of the program, the strongest post-condition shows the developer what can be guaranteed with P. If this conclusion is unsatisfactory, the *developer* can use it to improve P, to give stronger guarantees.

In the following section we will see that this weakest precondition structure, will also appear in differencing feature models.

2 Case Study: Difference for Feature Models

We will now define difference for the language of feature models [13]. To the best of our knowledge, semantic differences for feature models have not been studied before.

Definition 1. *A feature model is a tuple $M = (F, H, G, \varphi)$. F is a finite set of features, $H \subseteq F \times F$ is a set of directed edges, $G \subseteq 2^F$ is a set of or-groups, and φ is a Boolean formula over F expressing so-called* cross-tree constraints*. We demand that i) (F, H) is a forest*[3] *and write parent(f), for $f \in F$, for the unique $p \in F$ for which $(p, f) \in H$, and that ii) all states in an or-group share the same parent, so for all $e, f \in g \in G$, parent$(e) =$ parent(f).*

Fig. 1 presents feature models of two applets (in the spirit of [1]) which we will use as examples. We will use single letter names for features (underlined in the diagram). In applet_1, the root feature is a and represents the concept of an applet itself. The diagram says that the applet is decomposed into three smaller features (m, d, t). The empty circles above the names of d and t mean that implementing these two features is optional: an applet *may*, but does not *have to* override d and t. However, each applet *must* override (m) at least one of the methods p, s, and i; this necessity is denoted by the filled circle above the feature m and the filled arc in the concrete syntax. In the abstract syntax, this is expressed by or-groups $\{\mathsf{m}\}, \{\mathsf{p}, \mathsf{s}, \mathsf{i}\} \in G$. Moreover the cross-tree constraint (placed under the diagram) requires that any applet overriding d or s must also override i.

The variant of feature models presented above is among the simplest (and perhaps most popular) in use. The semantics of the language is defined in terms of translation to Boolean logics, see [2]. Let $M = (F, H, G, \varphi)$ be a feature model, then

$$[\![M]\!] = \varphi \wedge \Big(\bigwedge_{(p,c)\in H} c \to p \Big) \wedge \bigwedge_{\{f_1,\ldots,f_k\}\in G} \big(\mathit{parent}(f_1) \to \bigvee_{i=1}^{k} f_i\big).$$

The generated formula describes the configurations allowed by M. All of them need to satisfy the cross-tree constraint φ. Also, whenever a feature f is included in a configuration, its parent must be included, too. Finally, for each group at least one of its members must be present as soon as its (unique) parent is present. The semantics of our example is hence

$$\big((\mathsf{d} \to \mathsf{i}) \wedge (\mathsf{t} \to \mathsf{i})\big) \wedge \big((\mathsf{m} \to \mathsf{a}) \wedge (\mathsf{d} \to \mathsf{a}) \wedge (\mathsf{t} \to \mathsf{a}) \\ \wedge (\mathsf{p} \to \mathsf{m}) \wedge (\mathsf{s} \to \mathsf{m}) \wedge (\mathsf{i} \to \mathsf{m})\big) \wedge \big((\mathsf{a} \to \mathsf{m}) \wedge (\mathsf{m} \to (\mathsf{p} \vee \mathsf{s} \vee \mathsf{i}))\big).$$

Analysis techniques for feature models often rely on SAT solving or BDDs [23,16,24,17].

Consider now the feature model applet_2 of Fig. 1, which could emerge as a result of the same concept being modeled by another engineer. To focus attention we will assume that this model has been created by a designer of a component that needs to satisfy model applet_1 as a requirement. A few questions arise: How do these two models differ? Are they equivalent? If not, what is the actual difference?

[3] A forest is a finite disjunction of rooted trees, so technically we capture sets of feature models.

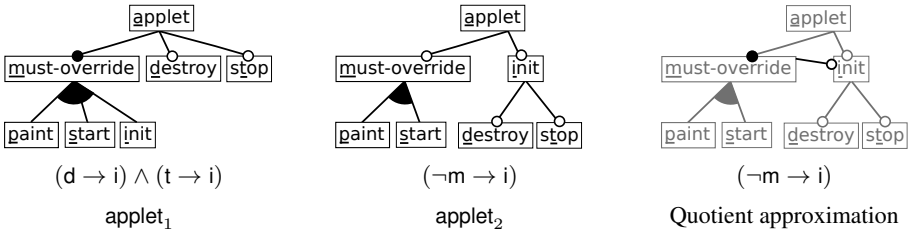

Fig. 1. The two example feature models and an over-approximation of their quotient

Syntactic difference algorithms cannot address these questions. A textual difference algorithm applied to the cross-tree constraint would just say that they differ, being unable to qualitatively explain the difference. An edit-distance based algorithm applied to the tree diagram could likely discover that i has been moved to become a parent of d and t, but not more – tree difference algorithms inform about the editing steps, but they cannot explain their impact. Admittedly, syntactic difference has a proven record of usefulness in many situations. However a modeler trying to understand the difference between the two diagrams, would likely ask a non-syntactic question: *What does this change mean?* Such question is best addressed semantically.

Following the pattern of the examples in the introduction, we will define the semantic difference of formulae φ and ψ as the "weakest" solution X to the implication $\varphi \wedge X \to \psi$. Hence:

Definition 2. *Given two formulae φ and ψ, a formula X is an* adjoint to the conjunction *$\varphi \wedge \psi$ if it holds for all formulae Y that*

$$\varphi \wedge Y \to \psi \quad \textit{iff} \quad Y \to X.$$

Thus X satisfies $\varphi \wedge X \to \psi$ and is implied by any Y which also solves this "equation". The next lemma shows that adjoints to conjunction are defined uniquely up to bi-implication, hence we may speak of *the* adjoint to a conjunction $\varphi \wedge \psi$ and denote it $X = \psi \setminus\!\!\!\wedge\, \varphi$ (provided that it exists, which we shall show below):

Lemma 1. *If X_1 and X_2 are adjoints to the conjunction $\varphi \wedge \psi$, then $X_1 \leftrightarrow X_2$.*

Proof. $X_1 \to X_1$ entails $\varphi \wedge X_1 \to \psi$ and hence $X_1 \to X_2$. Similarly for $X_2 \to X_1$. □

Existence of adjoints to conjunction is settled by the following lemma, whose proof is a routine verification of the property in the definition.

Lemma 2. *For formulae φ, ψ, we have $\psi \setminus\!\!\!\wedge\, \varphi \equiv \varphi \to \psi$.*

Coming back to our example, a routine computation shows that $[\![\mathsf{applet}_1]\!] \setminus\!\!\!\wedge\, [\![\mathsf{applet}_2]\!] = m \vee\ p \vee\ s \vee \neg\, a \vee \neg\, i$. We have computed the weakest cross-tree constraint which needs to be added to applet_2 for it to act like applet_1. In other cases it might not be useful just to compute a cross-tree constraint as the difference of two feature models; instead one might want a representation which is closer to the concrete feature-model syntax.

For this the algorithm displayed on the right can be used. It takes as input two formulae φ and ψ in conjunctive normal form (note that semantics of feature models are easily converted to CNF) and then finds for the quotient all clauses in ψ which are not entailed by φ (through the satisfiability check of $\varphi \wedge \neg c$ in line 3). This is clearly an over-approximation of the quotient, but might fail at maximality. It can still be useful in the software development process as a more syntactic representation.

QUOTIENT-AND
Input: φ, ψ : formulae in CNF
Output: an over-approx. to $\psi \setminus^{\wedge} \varphi$
1 Let $X = \emptyset$
2 **for each** clause $c \in \psi$ **do**
3 **if** SAT($\varphi \wedge \neg c$) **then** add c to X
4 **return** X

As an example, the approximation computed for the quotient of applet_1 by applet_2 is $(\neg \mathsf{a} \vee \mathsf{m}) \wedge (\neg \mathsf{i} \vee \mathsf{m}) = (\mathsf{a} \to \mathsf{m}) \wedge (\mathsf{i} \to \mathsf{m})$, and this can easily be added to the syntactic representation of applet_2 to signal the changes necessary, see Fig. 1 (rightmost).

In [6] we have presented a general feature model synthesis algorithm. The concrete syntax for the difference in the example above could be automatically computed by this algorithm. In general the algorithm could be used in a modeling tool visualizing semantic differences between feature models.

3 A Categorical Intermezzo

We will now generalize the considerations on adjoints and difference.

Definition 3. *A* preorder category *is a class C of* objects *and a* morphism *relation $\to_C \subseteq C \times C$ which is reflexive and transitive. A* functor *of preorder categories C, D is a mapping $F : C \to D$ which respects the morphisms: if $x \to_C y$ then $F(x) \to_D F(y)$.*

A preorder category is just a usual preorder, and a functor is a preorder homomorphism. We use categorical language here because *adjoints* are categorical concepts:

Definition 4. *Let C, D be preorder categories and $L : C \to D$, $R : D \to C$ functors. Then (L, R) is called an* adjoint pair *if it holds for all $x \in C$, $y \in D$ that*

$$L(x) \to_D y \quad \textit{iff} \quad x \to_C R(y).$$

In an adjoint pair (L, R), L is called the *left* and R the *right* adjoint. The notion of adjoints is important in category theory; note that we have simplified things here by only working in preorder categories, see e.g. [15, Ch. 4] for the full story. We can generalize the proof of Lemma 1 to show that up to isomorphism, one half of an adjoint pair determines the other:

Lemma 3. *If (L_1, R_1), (L_1, R_2), and (L_2, R_1) are adjoint pairs between preorder categories C, D, then $R_1(y) \leftrightarrow_C R_2(y)$ and $L_1(x) \leftrightarrow_D L_2(x)$ for all $x \in C$, $y \in D$.*

To apply these considerations to the setting of Section 2, we need only notice that we are working there in the category $\mathcal{F}$ with logical formulae as objects and implications as morphisms. If we denote by A_φ and I_φ, for $\varphi \in \mathcal{F}$, the mappings $\mathcal{F} \to \mathcal{F}$ given by $A_\varphi(\psi) = \varphi \wedge \psi$, $I_\varphi(\psi) = \psi \setminus^{\wedge} \varphi$, then the biimplication of Definition 2 reads

$$A_\varphi(Y) \to \psi \quad \text{iff} \quad Y \to I_\varphi(\psi),$$

hence we are defining an adjoint pair (A_φ, I_φ) for all formulae φ. Lemma 2 then says that such an adjoint pair exists for each formula φ. Another way to state this is that with tensor product $\wedge$, the category $\mathcal{F}$ is (strict symmetric) *closed monoidal*; in this context, the adjoint $\backslash^\wedge$ is also called the *exponential* to $\wedge$.

4 Difference and Development Processes

The adjoint to conjunction is useful in a top-down development scenario, when a general requirements model is given (applet_1) and a refinement is developed by a component designer (applet_2). By visualizing the difference $[\![\mathsf{applet}_1]\!] \backslash^\wedge [\![\mathsf{applet}_2]\!]$, the designer can monitor his refinement, and see how to constrain it to meet the general requirements.

In this scenario, *information flows top-down* – as in the quote in the introduction. The difference is used to refine models at lower abstraction levels. As much as this is useful, this is not fully satisfactory. In software engineering processes, *information flows both ways*. Especially in iterative processes the implementations are continuously adjusted to meet requirements, while requirements themselves are also continuously adjusted as a result of changing business conditions, and learning from experience in implementing the previous iterations. So we need to not only have ways for communicating model changes top-down in the refinement hierarchy, but *also* bottom-up.

Let us link these observations to differencing feature models. Observe that $\varphi \wedge X \to \psi$ is equivalent to $\varphi \to \neg X \vee \psi$. Moreover, if X is the weakest constraint that makes the former valid, then $\neg X$ is the strongest constraint that makes the latter valid. If interpreted in modeling terms, $\neg X$ represents the least amount of weakening that needs to be added to the model whose semantics is given by ψ (in the example applet_1) in order for the requirements to be possible to meet with components satisfying φ (applet_2). So $\neg X$ represents the information that *flows upwards* in the refinement hierarchy whenever it is not the component that needs to be 'fixed', but the requirements that need to be relaxed.

In our example, the negation of the difference formula is $\neg\mathsf{m} \wedge \neg\mathsf{p} \wedge \neg\mathsf{s} \wedge \mathsf{a} \wedge \mathsf{i}$. It directly describes a configuration of applet_2 that needs to be admitted by applet_1 in order to make the two models equivalent. In general this negation encodes all configurations of applet_2 that need to be admitted by applet_1 in order to make the two models equivalent.

We define the adjoint to disjunction using a universal property as in Def˙ 2: Given formulae φ, ψ, say that a formula X is an adjoint to the disjunction $\varphi \vee \psi$ if it holds that

$$\varphi \to Y \vee \psi \quad \text{iff} \quad X \to Y$$

for all formulae Y; hence X is now to be the "strongest" (with respect to implication ordering) solution to the implication $\varphi \to X \vee \psi$.

If we denote by O_ψ the mapping $O_\psi(\varphi) = \varphi \vee \psi$, the above bi-implication defines a *left adjoint* J_ψ to O_ψ, i.e. an adjoint pair (J_ψ, O_ψ). By the considerations of Section 3 we know that such left adjoint, if it exists, is unique; using Lemma 2 and self-duality of the category $\mathcal{F}$ we can conclude that $\psi \backslash^\vee \varphi := J_\psi(\varphi) = \neg(\psi \backslash^\wedge \varphi) = \neg(\varphi \to \psi)$, hence the adjoint to disjunction always exists.

Use Cases for Semantic Difference of Feature Models. Let us conclude the feature modeling example with a list of concrete applications for the difference of feature models, seen as a difference of their semantics (some of them already suggested above):

– Visualizing and explaining difference between models as specifications.
– The difference is a *debugging information*. Instances satisfying applet_2 but not the adjoint, are examples of configurations that are illegal in the requirements model.
– Dually they can be shown to the designer of applet_1 as examples of possible configurations, which might be used to expand requirements.
– If system configurations in $[\![\mathsf{applet}_2]\!] \wedge \neg([\![\mathsf{applet}_1]\!] \mathbin{\backslash^{\wedge}} [\![\mathsf{applet}_2]\!])$ pass correctness tests then the modeler should consider communicating them upward, to negotiate relaxation of these (otherwise reuse may be hindered).

5 Difference for Automata Specifications

We will now briefly show that the same construction of adjoint is applicable (and in fact known) for automata specifications. Assume a fixed alphabet of actions Σ.

Definition 5 ([14]). *A modal specification (MS) is a tuple $\mathcal{R} = (P, \lambda^0, \Delta^m, \Delta^M)$ where P is a set of states, $\lambda^0 \in P$ is the initial state and $\Delta^M \subseteq \Delta^m \subseteq P \times \Sigma \times P$. Δ^M and Δ^m are respectively* must- *and* may-*transitions, both deterministic and total: for every state $p \in P$ and action $a \in \Sigma$, there is exactly one $\lambda \in P$ such that $(p, a, \lambda) \in \Delta^m$.*

An automaton is a MS where $\Delta^M = \Delta^m$. An *instance* of a MS is an automaton that is obtained by unfolding the modal specification and cutting some may transitions while ensuring that all the must transitions stay present. Formally, let $R = (P, \lambda^0, \Delta^m, \Delta^M)$ be a MS and $A = (M, m^0, \Delta)$ an automaton. A is an instance of R, written $A \models R$, if there exists a binary relation $\rho \subseteq M \times P$ such that $(m^0, \lambda^0) \in \rho$, and for all $(m, p) \in \rho$:
(1) for every $(p, a, \lambda) \in \Delta^M$ there is a transition $(m, a, m') \in \Delta$ with $(m', \lambda) \in \rho$
(2) for every $(m, a, m') \in \Delta$ there is a transition $(p, a, \lambda) \in \Delta^m$ with $(m', \lambda) \in \rho$.
We write $[\![R]\!]$ for the set of instances of a MS R and say that a MS S refines another MS T, written $S \leq T$, iff $[\![S]\!] \subseteq [\![T]\!]$.

Two modal specifications over the same alphabet can be composed by synchronizing on common actions, similarly to composition for regular transition systems, but with the provision that the composition of two may-transitions is again a may-transition, and the composition of two must-transitions is a must-transition. The composition $M_1 \parallel M_2$ accepts all compositions between models of M_1 and of M_2, so $[\![M_1 \parallel M_2]\!] = \{(m_1 \parallel m_2) \mid m_1 \in [\![M_1]\!], m_2 \in [\![M_2]\!]\}$.

Given specifications S and T, the *quotient* operation $\mathbin{\backslash\!\!\parallel}$ computes the greatest specification X (with respect to the refinement order) such that $S \parallel X \leq T$. So $T \mathbin{\backslash\!\!\parallel} S$ is essentially the difference between S and T with respect to structural composition – it describes the component that is missing in order to provide T. In a more succinct way we can say that X is a quotient of T by S if it holds that

$$S \parallel Y \leq T \quad \text{iff} \quad Y \leq X$$

for all specifications Y. In the spirit of Section 3, we can note that modal specifications and refinements form a preorder category $\mathcal{M}$, and then the bi-implication above means that quotient is the *right adjoint* to structural composition, i.e. that for any specification S, the functors $P_S(T) = S \parallel T$ and $Q_S(T) = T \mathbin{\backslash\!\!\parallel} S$ form an adjoint pair (P_S, Q_S).

Algorithms for computing these quotients are known for many behavioral component algebras [7,4,19,3,10].

6 Discussion: Towards Difference Between Languages

We have characterized semantic distance as an adjoint of a composition operator, and exemplified it for conjunction, disjunction, and parallel composition. In this section we want to illustrate an interesting direction into discussing semantic difference, namely characterizing distance between an instance of a modeling language and a subclass of this language.

This problem appears often in practice. For instance model-checkers for automata-like models may assume that models are deterministic to improve efficiency. Similarly, analysis tools for class diagrams may assume use of a subset of OCL, in order to make the validity (or consistency) problem decidable. For feature models, it is sometimes interesting to look at a class of models that are possible to represent purely diagrammatically (i.e. without cross-tree constraints). However modeling using the full power of the language is usually easier. It is efficient to abstract behaviors with nondeterminism; it is easier to write constraints in full OCL; and it is often natural to express some cross-tree constraints in propositional logics. So the problem arises, whether the full-featured instance of the language is far, or not far, from the subclass of models which are easy to analyse. Is it easy to translate into this subclass? How much expressivity is lost (if any)?

Such translation is usually performed by an abstraction operation. Automata can be determinized; OCL (and propositional) constraints can be weakened to approximate their semantics within the sublanguage. Interestingly such an abstraction is also an adjoint, manifesting the same abstract structure as the instance-to-instance differences. Below we detail this for the example of determinization of modal automata.

The essence of a *determinization* operator det for (non-deterministic) modal specifications is that for any specification S, $\mathsf{det}(S)$ is the *smallest deterministic over-approximation* of S. Hence $\mathsf{det}(S)$ is deterministic, $S \leq \mathsf{det}(S)$, and for any deterministic specification D, $S \leq D$ implies $\mathsf{det}(S) \leq D$. Now the last two properties can be combined by demanding that

$$S \leq D \quad \text{iff} \quad \mathsf{det}(S) \leq D$$

for all deterministic D, which is almost the property we have encountered earlier.

Now let $\mathcal{M}$ be the preorder category of deterministic modal specifications as before, and let $\mathcal{N}$ be the larger category of non-deterministic specifications. We have a functor $I : \mathcal{M} \to \mathcal{N}$ (which "forgets" that the specification is deterministic; hence called a *forgetful* functor), and det is a functor $\mathcal{N} \to \mathcal{M}$. The equation above then becomes

$$\mathsf{det}(S) \leq_{\mathcal{M}} D \quad \text{iff} \quad S \leq_{\mathcal{N}} I(D)$$

for all $S \in \mathcal{N}$, $D \in \mathcal{M}$. Hence the determinization functor det is *left adjoint* to the forgetful functor I; this type of functors is usually called *free*.

We see in this example that existence of a faithful abstraction to the subclass of our modeling language, which maps a model to an abstraction which is "not too far" away, is the same as a *free* functor from the language to the subclass, left adjoint to the forgetful functor. This is indeed characteristic of a number of other examples, and motivates the search for free functors also in other areas.

7 Final Remarks and Related Work

We have described a formal approach to defining semantic difference between models. Perhaps somewhat unexpectedly, our proposal relies on using a preorder on models, instead of using equality (equivalence) and attempting to construct some sort of counterpart of subtraction. Our difference is an operator that is defined as an adjoint. In modeling it makes sense to consider differencing with respect to various composition operators, with conjunction and structural composition being the two main contenders.

Let us briefly summarize the process of defining a semantic difference:

1. Identify a set of models $\mathbf{S}$ and a preorder $\leq$ on $\mathbf{S} \times \mathbf{S}$ (here this was a refinement on automata, or implication of formulae; in other contexts it could be subtyping).
2. Choose a binary composition operator (merge) $\otimes : \mathbf{S} \times \mathbf{S} \to \mathbf{S}$. We have used entailment, parallel composition, conjunction and disjunction in this role.
3. The semantics of models is given as a mapping $[\![\cdot]\!] : \mathbf{S} \to \mathbf{D}$ to a semantic domain.
4. Usually the semantic domain $\mathbf{D}$ has better algebraic structure than the syntactic domain $\mathbf{S}$. Thus it is easier to define the difference, as an operator $\backslash^{\otimes}$ on the semantic domain: $\backslash^{\otimes} : \mathbf{D} \times \mathbf{D} \to \mathbf{D}$. By definition $T \backslash^{\otimes} S$ returns the maximum X for which $S \otimes X \leq T$, or (as for disjunction) the minimum X for which $S \leq X \otimes T$. Not in all semantic domains such a maximum, or minimum, may exist, but if it does, it is unique (up to the equivalence relation induced by the preorder $\leq$).

In the future we intend to work on semantic differences for other modeling languages, including UML class diagrams. Providing a difference for this language requires that we are able to compute differences for a substantial fragment of first order logics.

Related Work Semantic difference is discussed in [21], which defines the difference operator between models $T - S$ as a set of *witnesses*, which are instances of T but not instances of S. While this definition is natural, and can be useful in many practical cases (for example it directly allows providing counterexamples for non-emptiness of difference), it also has drawbacks. Unlike our proposal, such definition of difference defines an operator which has a different co-domain than the domains of operands. A difference between models is no longer a model. Secondly, in most practical cases, the set of witnesses is infinite and cannot easily be enumerated.

Model merging [5] is composing overlapping models, typically, without prior computation of differences between them. In [18] a semantics oriented merge operation is discussed for statecharts. It would be interesting to see whether this work could be extended to provide visualization of semantic differences for statecharts.

Gerth and co-authors [9] present a semantic-based notion between change operations in a version control scenario. Two operations are equivalent if they lead to equivalent business process models (in the sense of trace inclusion). They are not concerned with synthesizing difference models, but with detecting and avoiding merge conflicts. Our operator, could potentially be used in conflict resolution or visualizing changelogs.

Segura et al. [20] define a syntactic merge operator for feature models using graph transformations. Closer to semantics, Thüm et al. [23] discuss semantic differences of edits to feature models. They do not compute differences but simply classify them as strengthening, weakening, refactoring, and incomparable.

Semantic difference for programs is understood better than for models. For instance, in [12] differences between procedures are approximated by dependence relations.

Acknowledgments. We thank Krzysztof Czarnecki for indicating the semantic difference problem to us, and Jose Fiadeiro for an encouraging discussion on the subject.

References

1. M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MoDELS*, volume 4199 of *LNCS*. Springer, 2006.
2. D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *LNCS*. Springer, 2005.
3. N. Bertrand, A. Legay, S. Pinchinat, and J.-B. Raclet. A compositional approach on modal specifications for timed systems. In *ICFEM*, volume 5885 of *LNCS*. Springer, 2009.
4. P. Bhaduri and S. Ramesh. Synthesis of synchronous interfaces. In *ACSD*. IEEE, 2006.
5. G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *GaMMa*. ACM, 2006.
6. K. Czarnecki and A. Wąsowski. Feature diagrams and logics: There and back again. In *SPLC*, pages 23–34. IEEE Computer Society, 2007.
7. A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wąsowski. Timed I/O automata: a complete specification theory for real-time systems. In *HSCC*. ACM, 2010.
8. E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics*. Springer, 1990.
9. C. Gerth, J. M. Küster, M. Luckey, and G. Engels. Precise detection of conflicting change operations using process model terms. In *MoDELS (2)*, volume 6395 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2010.
10. G. Goessler and J.-B. Raclet. Modal contracts for component-based design. In D. V. Hung and P. Krishnan, editors, *SEFM*. IEEE Computer Society, 2009.
11. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 1969.
12. D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In H. A. Müller and M. Georges, editors, *ICSM*. IEEE Computer Society, 1994.
13. K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.
14. K. G. Larsen. Modal specifications. In *AVMS*, volume 407 of *LNCS*, 1989.
15. S. Mac Lane. *Categories for the Working Mathematician*. Graduate texts in mathematics. Springer, second edition, 1998.
16. M. Mendonca, A. Wąsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *SPLC'09*. IEEE Computer Society, 2009.
17. M. Mendonça, A. Wąsowski, K. Czarnecki, and D. D. Cowan. Efficient compilation techniques for large scale feature models. In *GPCE*, 2008.
18. S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *ICSE*. IEEE Computer Society, 2007.
19. J.-B. Raclet. Residual for component specifications. *ENTCS*, 215:93–110, 2008.
20. S. Segura, D. Benavides, A. R. Cortés, and P. Trinidad. Automated merging of feature models using graph transformations. In *GTTSE*, volume 5235 of *LNCS*. Springer, 2007.
21. J. R. Shahar Maoz and B. Rumpe. A manifesto for semantic model differencing. In *International Workshop on Models and Evolution*, 2010.
22. I. Sommerville. *Software Engineering, 9/E*. Addison-Wesley, 2011.
23. T. Thüm, D. S. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE*, pages 254–264. IEEE Computer Society, 2009.
24. P. Trinidad, D. Benavides, A. R. Cortés, S. Segura, and A. Jimenez. FAMA framework. In *SPLC*, page 359. IEEE Computer Society, 2008.