Semantic Model Differencing Utilizing Behavioral Semantics Specifications

Philip Langer, Tanja Mayerhofer, and Gerti Kappel

Business Informatics Group, Vienna University of Technology, Vienna, Austria
{langer,mayerhofer,gerti}@big.tuwien.ac.at

Abstract. Identifying differences among models is a crucial prerequisite for several development and change management tasks in model-driven engineering. The majority of existing model differencing approaches focus on revealing syntactic differences which can only approximate semantic differences among models. Significant advances in semantic model differencing have been recently made by Maoz *et al.* [16] who propose semantic diff operators for UML class and activity diagrams. In this paper, we present a generic semantic differencing approach which can be instantiated to realize semantic diff operators for specific modeling languages. Our approach utilizes the behavioral semantics specification of the considered modeling language, which enables to execute models and capture execution traces representing the models' semantic interpretation. Based on this semantic interpretation, semantic differences can be revealed.

1 Introduction

The identification of differences among independently developed or consecutive versions of software artifacts is not only a crucial prerequisite for several important development and change management tasks, such as merging and incremental testing, but also for enabling developers to efficiently comprehend an artifact's evolution. As in model-driven engineering the main software artifacts are models, techniques for identifying *differences among models* are of major importance.

The challenge of model differencing has attracted much research in the past years, which lead to significant advances and a variety of approaches. The majority of them use a *syntactic differencing* approach, which applies a fine-grained comparison of models based on their abstract syntax representation. As shown by Alanen and Porres [1] and later by Lin *et al.* [14], syntactic differencing algorithms can be designed in a generic manner—that is, they can be applied to models conforming to any modeling language. The result of such a differencing approach is a set of syntactic differences, such as model elements that only exist in one model. Although syntactic differences constitute valuable and efficiently processable information sufficient for several application domains, they are only an approximation of the *semantic differences* among models with respect to their meaning [10]. In fact, few syntactic differences among models may induce many semantic differences, whereas also syntactically different models may still exhibit the same semantics [16]. The identification of semantic differences is crucial for understanding the evolution of a model, as it enables to reason about the meaning of a change. Compared to syntactic differencing, semantic differencing enables several

J. Dingel et al. (Eds.): MODELS 2014, LNCS 8767, pp. 116–132, 2014.
© Springer International Publishing Switzerland 2014

additional analyses, such as the verification of the semantic preservation of changes like of refactorings and the identification of semantic conflicts among concurrent changes.

Significant advances towards *semantic model differencing* have been recently made by Maoz *et al.* [16]. They propose semantic diff operators yielding so-called *diff witnesses*, which are interpretations over a model that are valid in only one of the two compared models. The semantic diff operator has to be realized specifically for each modeling language by transforming models into an adequate semantic domain, performing dedicated analyses within this semantic domain, translating the results of the analyses back again, and representing them in the form of diff witnesses. Following this procedure, Maoz *et al.* presented dedicated diff operators for UML activity diagrams [17] and class diagrams [18]. Developing such diff operators for a specific modeling language, however, still remains a major challenge, as one has to develop often non-trivial transformations encoding the semantics of the modeling language into a semantic domain, perform analyses dedicated to semantic differencing in this semantic domain, and translate the results into diff witnesses on the level of the modeling language—notably, this challenging process has to be repeated for every modeling language.

To mitigate this challenge, we present a *generic* semantic differencing approach that can be instantiated to realize semantic diff operators for specific modeling languages. This approach follows the spirit of *generic* syntactic differencing, which utilizes metamodels to obtain the necessary information on the syntactic structure of the models to be compared. Accordingly, we propose to utilize the behavioral semantics of a modeling language to support the semantic model differencing. In particular, we exploit the executability of the behavioral semantics to obtain execution traces for the models to be compared. These traces are considered as semantic interpretations over the models and, thus, act as input to the semantic comparison. The actual comparison logics is specified in terms of dedicated match rules defining which differences among these interpretations constitute semantic differences. Semantic diff operators defined with our approach are enumerative yielding diff witnesses, which constitute manifestations of semantic differences among models and enable modeler's to reason about a model's evolution. Hence, the diff operators constitute a crucial basis for supporting collaborative work on models as well as for carrying out model management activities, such as model versioning and refactoring, which can be supported by an automated analysis of diff witnesses for identifying conflicting changes and causes of semantic differences.

In Section 2, we discuss existing work in the area of model differencing, before we introduce our semantic differencing approach in Section 3. Subsequently, we show in Section 4 how semantic diff operators can be implemented by applying our approach to an existing semantics specification approach. In Section 5, we address the issue of generating model inputs relevant to semantic differencing. Finally, we present an evaluation of the feasibility of our approach in Section 6 and draw conclusions in Section 7.

2 Related Work

Most of the existing model differencing approaches compare two models based on their *abstract syntax* representation (e.g., [1,2,14,25,26,31]). In particular, a match between two models is computed yielding the correspondences between their model elements,

before a fine-grained comparison of all corresponding model elements is performed. The result of this syntactic differencing is the set of model elements present in only one model and a description of differences among model elements present in both models.

However, to determine whether two syntactically different models also differ in their meaning, the semantics of the modeling language they conform to has to be taken into account [10]. Few *semantic model differencing* approaches have been proposed in the past. Generally, we can distinguish enumerative and non-enumerative approaches. *Enumerative* approaches calculate semantic interpretations of two compared models called *diff witnesses*, which are only valid for one of the two models and, hence, provide evidence about the existence of semantic differences among the models. *Non-enumerative* approaches do not calculate and enumerate diff witnesses directly, but instead compute an aggregated description of the semantic difference among the compared models [8].

Significant advances in semantic differencing have been achieved by Mazo *et al.*, who propose an approach for defining enumerative semantic diff operators [16]. In this approach, two models to be compared are translated into an adequate semantic domain whereupon dedicated algorithms are used to calculate semantic differences in the form of diff witnesses. Following this approach, they define the diff operators CDDiff [18] and ADDiff [17], for UML class diagrams and UML activity diagrams, respectively. CDDiff translates UML class diagrams into an Alloy module to generate object diagrams that are valid instances of one class diagram but not of the other. ADDiff translates two UML activity diagrams into SMV modules to identify execution traces which are possible only in one of the two activity diagrams. Gerth *et al.* [9] developed an enumerative semantic diff operator similar to ADDiff for detecting semantic differences among process models. Therefore, the process models are translated into process model terms, which are subsequently compared to identify execution traces valid only for one of the two compared models. Another approach for defining enumerative semantic diff operators was presented by Reiter *et al.* [28]. In their approach, two models that shall be compared are translated into a common semantic domain. The resulting so-called *semantic views* of the two models are subsequently compared by syntactic differencing techniques to identify semantic differences.

Unlike the approaches discussed so far, Fahrenberg *et al.* [8] propose an approach for defining non-enumerative semantic diff operators. Therefore, the models to be compared are mapped into a semantic domain having an algebraic structure that enables to define the difference among two models in the form of an operator on the semantic domain. Thereby, the difference is captured in the form of a model conform to the same modeling language as the two compared models. Fahrenberg *et al.* applied this approach for defining semantic diff operators for feature models as well as automata specifications [8], and later also for UML class diagrams [7].

3 Overview

Developing semantic diff operators using the discussed existing semantic differencing approaches poses a major challenge, because one has to develop non-trivial transformations encoding the semantics of the modeling language into an adequate semantic domain, in which then specific semantic comparison algorithms have to be imple-

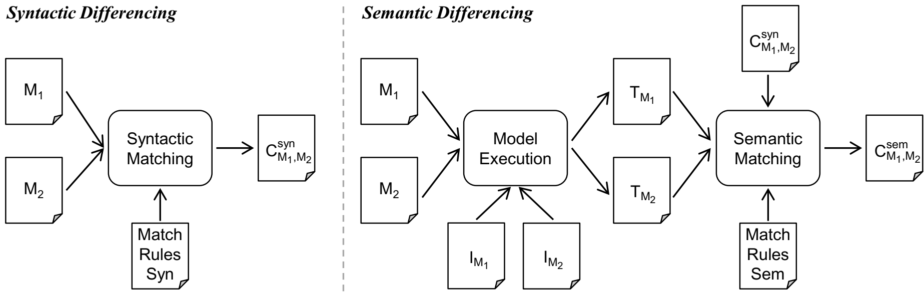

Fig. 1. Overview of semantic differencing approach

mented. To mitigate this challenge, we propose a *generic* semantic differencing approach that can be instantiated to realize semantic diff operators for specific modeling languages. Therefore, we utilize the *behavioral semantics specification* of a modeling language, which can be defined using existing semantics specification approaches, such as xMOF [20], Kermeta [22], or DMM [5]. Such semantics specifications can be used for various application domains, ranging from model simulation, verification, through to validation. In this work, we aim at reusing such semantics specifications also for semantic model differencing. In particular, we exploit the fact that behavioral semantics specifications enable the execution of models and that the identification of semantic differences among models is possible based on *execution traces*, since they reflect the models' behavior and, hence, constitute the semantic interpretation of the models.

Figure 1 depicts an overview of our semantic model differencing approach consisting of three steps: syntactic matching, model execution, and semantic matching. In the *syntactic matching* step, syntactically corresponding elements of the two compared models M_1 and M_2 are identified based on syntactic match rules $MatchRulesSyn$ for establishing syntactic correspondences $C^{syn}_{M_1,M_2}$ between the models. In the *model execution* step, the models M_1 and M_2 are executed for relevant inputs I_{M_1} and I_{M_2} based on the behavioral semantics specification of the modeling language. During model execution, the traces T_{M_1} and T_{M_2} are captured, which constitute the semantic interpretation of the executed models M_1 and M_2. We assume that the model execution is deterministic, meaning that the model execution yields for a given input always the same execution trace, and that the number of possible traces is finite. In the *semantic matching* step, the captured execution traces T_{M_1} and T_{M_2} are compared based on semantic match rules $MatchRulesSem$, which define the semantic equivalence criteria, to establish semantic correspondences $C^{sem}_{M_1,M_2}$ between the models M_1 and M_2. Thereby, two models M_1 and M_2 are semantically equivalent, if the traces captured during their execution T_{M_1} and T_{M_2} match according to the semantic match rules. In the semantic matching, also the syntactic correspondences of the examined models $C^{syn}_{M_1,M_2}$ may be taken into account.

Our semantic model differencing approach is *generic*, because it enables to implement semantic diff operators for any modeling languages whose behavioral semantics is defined such that conforming models can be executed and execution traces can be obtained. From all artifacts involved in the semantic differencing, only the semantic

match rules are specific to the realization of a semantic diff operator for a modeling language. This is an important differentiator of our approach compared to currently existing semantic model differencing approaches.

4 Semantic Model Differencing

In this section, we show how the proposed semantic model differencing approach can be realized for the behavioral semantics specification language xMOF [20]. Therefore, we first introduce how the behavioral semantics of modeling languages can be defined with xMOF and how this definition is used to execute models. Second, we present which trace information is needed for semantically differencing models. Third, we show how semantic match rules can be defined for semantically comparing models based on execution traces. For illustrating the presented techniques, we use the Petri net language as a running example throughout the paper.

4.1 Behavioral Semantics Specification with xMOF

The semantics specification language xMOF integrates existing metamodeling languages, in particular Ecore, with the action language of UML [23]. This enables the definition of the behavioral semantics of the concepts of a modeling language by introducing operations for the respective metaclasses and defining their behavior with UML activities. UML's action language for defining activities provides a predefined set of actions for expressing the manipulation of objects and links (e.g., CreateObjectAction) and the communication between objects (e.g., CallOperationAction), as well as a model library of primitive behaviors (e.g., IntegerPlus). xMOF enables the execution of models by executing the activities defined in the modeling language's semantics specification based on the fUML virtual machine. fUML [24] is a standard of the OMG, which defines the semantics of a subset of UML activity diagrams formally and provides a virtual machine enabling the execution of fUML-compliant models.

Example. In the upper left part of Figure 2, the *Ecore-based metamodel* of the Petri net language is depicted. A Petri net (Net) consists of a set of uniquely named places (Place) and transitions (Transition), whereas transitions reference their input and output places (input, output). The initial marking of the Petri net is captured by an Integer attribute (initialTokens) of the Place metaclass.

In xMOF, the behavioral semantics of a modeling language is defined in an *xMOF-based configuration*. This xMOF-based configuration contains for each metaclass in the metamodel a *configuration class*, which extends the respective metaclass with its behavioral semantics by introducing additional attributes and references for capturing runtime information, as well as additional operations and activities for defining behavior. In the lower left part of Figure 2, the configuration classes contained by the xMOF-based configuration of the Petri net language are shown. To capture the number of held tokens of a Petri net during its execution, the Integer attribute tokens is introduced in the configuration class PlaceConfiguration. The main() operation of NetConfiguration serves as entry point for executing a Petri net model. It first calls the operation initializeMarking(), which initializes the tokens attribute of each Place instance with the value of the

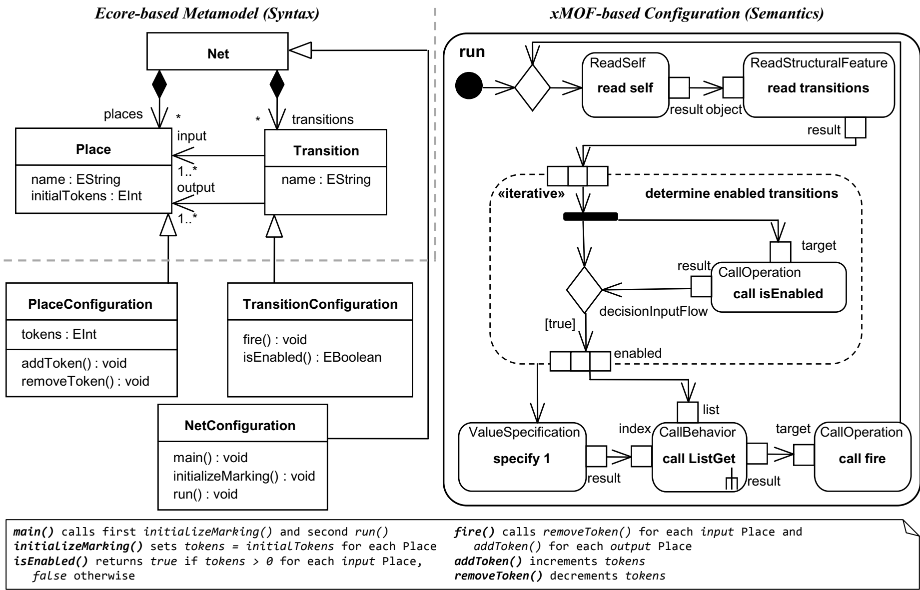

Fig. 2. Petri net language specification

initialTokens attribute, before the operation run() is invoked. The activity defining the behavior of run() is depicted in the right part of Figure 2. It determines in a loop the set of enabled transitions (ExpansionRegion "determine enabled transitions"), selects the first enabled transition (CallBehaviorAction "call ListGet"), and calls the operation fire() (CallOperationAction "call fire") for this transition. Subsequently, the operation fire() calls for each input place of the transition the operation removeToken() to decrement its tokens value and addToken() for each output place to increment its tokens value.

Based on this behavioral semantics specification, a Petri net model can be executed. Therefore, the configuration classes are instantiated for each model element in the Petri net model and the resulting instances are initialized according to the attribute values and references of the respective model element. These instances are then provided to the

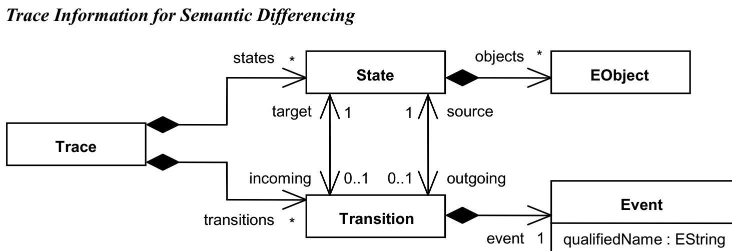

Fig. 3. Trace information format specification

fUML virtual machine as input before the main() operation is invoked. Consequently, during the execution, the values of the tokens attribute of the PlaceConfiguration instances are updated accordingly by the fUML virtual machine.

4.2 Trace Information

In our approach, trace information obtained from executing the two models to be compared constitutes the basis for reasoning about semantic differences among the models. The format of the trace information is defined by the metamodel depicted in Figure 3. A trace (Trace) consists of states (State) capturing the runtime state of the executed model (objects) at a specific point in time of the execution. Transitions (Transition) are labeled with the event (Event) that caused a state change of the executed model leading from one state (source) to another (target). This trace information format constitutes the interface for using our semantic differencing approach. Hence, our approach does not directly depend on a specific behavioral semantics specification language or on a specific virtual machine; it only operates on traces conforming to this very basic trace format.

4.3 Semantic Model Differencing Based on Trace Information

For semantically differencing two models, the trace information captured by executing these models are compared according to semantic match rules. These match rules decide based on the states of the compared models and based on the events causing state transitions which model elements semantically correspond to each other and whether the two models are semantically equivalent. The semantic match rules are specific to the considered modeling language as well as to the relevant semantic equivalence criteria. Thereby our approach is flexible in the sense that match rules can be expressed for different equivalence criteria. This is an important property because, depending on the usage scenario of a modeling language, different equivalence criteria for models may apply. For Petri nets, for instance, different equivalence criteria are *marking equivalence*, *trace equivalence*, and *bisimulation equivalence* [6]. If Petri nets are used to define production processes, where the tokens residing in places represent production resources, the marking equivalence criteria might be the most suitable equivalence criteria. However, if Petri nets are used to define business processes, the trace equivalence criteria might be more adequate.

For defining match rules, our implementation integrates the model comparison language ECL [12]. In ECL, model comparison algorithms are specified with declarative rules which are used to identify pairs of matching elements in two models.

Example. We now consider two different semantic equivalence criteria for Petri nets: final marking equivalence (which we defined for illustration purposes) and marking equivalence (adopted from literature [6]). Two Petri net models with the same set of places are *final marking equivalent* if they have the same final marking, whereas they are *marking equivalent*, if they have the same set of reachable markings. The example Petri net models $PN1$ and $PN2$ depicted in Figure 4 are not final marking equivalent because their final markings $M_{2,PN1}$ and $M_{2,PN2}$ are different. However, they are

marking equivalent, as they have the same set of reachable markings ($M_{0,PN1}$ matches with $M_{1,PN2}$, $M_{1,PN1}$ with $M_{2,PN2}$, and $M_{2,PN1}$ with $M_{0,PN2}$).

Listing 1 shows the semantic match rules expressed in ECL for determining whether two Petri net models are final marking equivalent. The semantic match rule $Match\ Trace$ (lines 1-8) is responsible for matching the traces captured for the execution of two compared Petri net models. If the models are final marking equivalent, this rule has to return true, otherwise false. Therefore, the respective final states of the two traces $markingStatesLeft$ and $markingStatesRight$ are obtained using the operation $getMarkingStates()$ (lines 10-12). These retrieved final states are then matched with each other (line 6) and if they match, the Petri net models are final marking equivalent and true is returned. The two final states are matched by the rule $MatchState$ (line 14-22). Therefore, the final runtime states of the PlaceConfiguration instances from the respective final state are retrieved by calling the operation $getPlaceConfigurations()$ (lines 24-30) and true is returned if the PlaceConfiguration instances match (line 20). The PlaceConfiguration instances are matched by the rule $MatchPlaceConfiguration$ (lines 32-37), which defines that two PlaceConfiguration instances match, if they match syntactically (this is checked by the extended syntactic match rule $MatchPlace$ not shown here, which defines that two Place instances match if they have the same name) and if they contain the same amount of tokens (line 36). Thus, in the end, the match rule $MatchTrace$ returns true, if the two compared Petri net models have the same markings in the end of the execution and are, hence, final marking equivalent.

For realizing the marking equivalence criteria, the operation $getMarkingStates()$ has to be adapted as shown in Listing 2. It retrieves the state after the initializeMarking activity has been executed for the NetConfiguration instance (line 3) and after each execution of the fire activity for any TransitionConfiguration instance (line 4). Therefore, the operation $getStatesAfterEvent()$ provided by the trace is used, which retrieves the states caused by an event corresponding to the provided qualified name. Thus, the operation $getMarkingStates()$ returns the runtime states of the compared models after initializing the marking of the net and after each transition firing, i.e., each state after reaching a new marking. These sets of states $markingStatesLeft$ and $markingStatesRight$ match (cf. line 6 in Listing 1), if each state in $markingStatesLeft$ has a corresponding state in $markingStatesRight$ and vice versa; that is, if each marking reachable in $PN1$ is also reachable in $PN2$ and vice versa. Please note that we restrict ourselves in this example to conflict-free and terminating Petri nets.

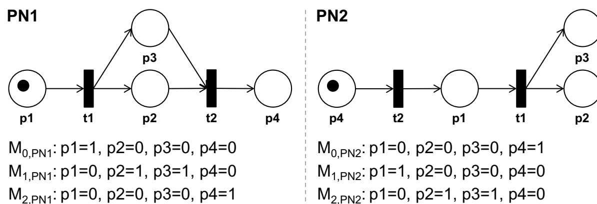

Fig. 4. Example Petri net models

```
1  rule MatchTrace
2      match left : Left!Trace with right : Right!Trace {
3        compare {
4              var markingStatesLeft : Set = left.getMarkingStates();
5              var markingStatesRight : Set = right.getMarkingStates();
6              return markingStatesLeft.matches(markingStatesRight);
7        }
8      }
9
10 operation Trace getMarkingStates() : Set {
11       return self.states.at(self.states.size() − 1).asSet();
12 }
13
14 @lazy
15 rule MatchState
16     match left : Left!State with right : Right!State {
17       compare {
18             var placeConfsLeft : Set = left.getPlaceConfigurations();
19             var placeConfsRight : Set = right.getPlaceConfigurations();
20             return placeConfsLeft.matches(placeConfsRight);
21       }
22     }
23
24 operation State getPlaceConfigurations() : Set {
25       var placeConfs : Set = new Set();
26       for (object : Any in self.objects)
27             if (object.isKindOf(PlaceConfiguration))
28                   placeConfs.add(object);
29       return placeConfs;
30 }
31
32 @lazy
33 rule MatchPlaceConfiguration
34     match left : Left!PlaceConfiguration with right : Right!PlaceConfiguration
35     extends MatchPlace {
36       compare : left.tokens = right.tokens
37     }
```

Listing 1. Semantic match rules for Petri net final marking equivalence

```
1  operation Trace getMarkingStates() : Set {
2        var markingStates : Set = new Set();
3        markingStates.addAll(self.getStatesAfterEvent("petrinetConfiguration.NetConfiguration.
             ↪initializeMarking"));
4        markingStates.addAll(self.getStatesAfterEvent("petrinetConfiguration.
             ↪TransitionConfiguration.fire"));
5        return markingStates;
6  }
```

Listing 2. Adaptation of semantic match rules for Petri net marking equivalence

5 Input Generation Using Symbolic Execution

In Section 4, we showed how a semantic diff operator can be specified with match rules that are applied to concrete execution traces for determining whether two models are semantically equivalent. However, for several modeling languages additional input is required—alongside the actual model—before it can be executed. For instance, the Petri net language depicted in Figure 2 could take the initial token distribution as input instead of representing it in the model directly (Place.initialTokens). Enumerating all possible inputs and performing the semantic differencing for all resulting traces is not feasible for several scenarios, as the number of possible inputs may quickly become

large or even infinite. In fact, we are interested only in inputs that cause *distinct and for the semantic differencing relevant execution traces*. Having obtained such inputs, the models to be compared can be executed for these inputs and the semantic match rules can be applied on the captured traces for semantically differencing the models. Thereby, two models are semantically equivalent, if they exhibit the same behavior for the same inputs as defined by the semantic match rules. If they behave differently for the same input, they differ semantically and the respective input constitutes a diff witness.

For *automatically* generating relevant inputs from the semantics specification of the modeling language for the two models to be compared, we apply an adaptation of symbolic execution [3]. The basic idea behind symbolic execution, as introduced by Clarke [4], is to execute a program—in our case, the semantics specification for a specific model—with *symbolic values* in place of concrete values and to record a *path constraint*, which is a quantifier-free first-order formula, for each conditional statement that is evaluated over symbolic values along an execution path. For each symbolic value, a symbolic state is maintained during the symbolic execution, which maps symbolic values to symbolic expressions. After executing a path symbolically, we obtain a sequence of path constraints, which can be conjuncted and solved by a constraint solver to obtain concrete inputs. An execution with these inputs will consequently exercise the path that has been recorded symbolically. If a conjunction of path constraints is unsatisfiable, the execution path can never occur. Using backtracking and negations of path constraints, we may further obtain *all feasible paths* represented as an *execution tree*, which is a binary tree, where each node denotes a path constraint and each edge a Boolean value.

More recently, several extensions and flavors of traditional symbolic execution have been proposed (cf. [3] for a survey). For this work, we apply a combination of concolic execution [29] and generalized symbolic execution [11]. *Concolic execution* significantly decreases the number of path constraints by distinguishing between concrete and symbolic values. The program is essentially executed as normal and only statements that depend on symbolic values are handled differently. As we execute the semantics specification with a concrete model (to be compared) and additional input, we may consider only the additional input as symbolic values—statements that interact with the executed model itself are executed as normal. One of the key ideas behind *generalized symbolic execution* also used in this work is to use lazy initialization of symbolic values. Thus, we execute the model as normal and initialize empty objects for symbolic values only when the execution accesses the object for the first time. Similarly, attribute values of objects are only initialized on their first access during the execution with dedicated values to induce a certain path during the execution.

Example: Initial tokens as input. Before we discuss how we apply symbolic execution on our running example, we have to slightly modify the Petri net language depicted in Figure 2 such that it takes the initial token distribution as input. Therefore, we add a class Token, which owns a reference named place to Place denoting the token's initial place. Additionally, we change the operation NetConfiguration.main() and add a parameter initialTokens of type EList<Token> to this operation. The activity specifying the behavior of the operation main() passes the list of initial tokens to the operation initializeMarking(), which in turn sets the number of tokens for each place p to `p.tokens = initialTokens->select(t | t.place = p).size()`; note that we define

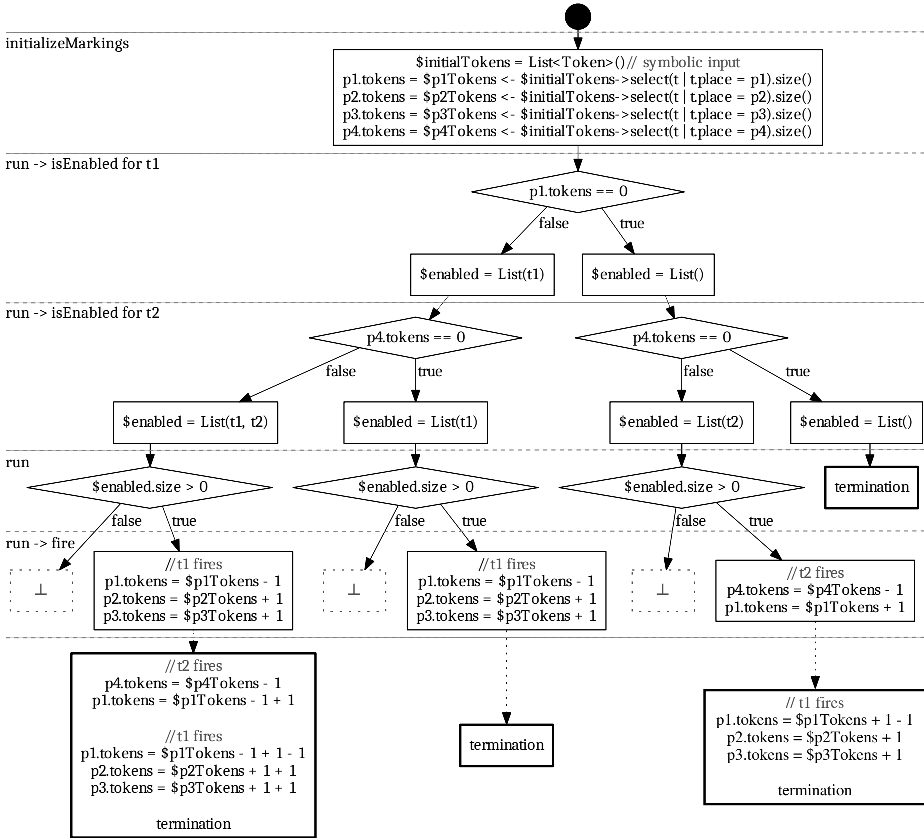

Fig. 5. Excerpt of the execution tree for $PN2$ (cf. Figure 4)

this assignment here in OCL syntax for the sake of brevity, in an operational semantics specification, this assignment is specified in terms of an action language, such as fUML.

Example: Symbolic execution. To derive input values that cause all distinct execution traces, we symbolically execute the operational semantics of the Petri net language with both models to be compared, whereas the input of the execution—that is, the parameter initialTokens of type EList<Token>—is represented as a symbolic value. Figure 5 shows an excerpt of the resulting execution tree for $PN2$ (cf. Figure 4). Note that we bound the symbolic execution to at most one initial token per place in this example. We depict path constraints as diamonds and the symbolic states of symbolic values in boxes; symbolic values are prefixed with a $ symbol. The uppermost box shows the symbolic states after executing the operation initializeMarking(initialTokens). As this operation assigns the number of tokens based on the symbolic input initialTokens to the tokens attribute of each place, also the values assigned to this attribute are handled symbolically. The initial symbolic values for this attribute are mapped to the symbolic expression `$initialTokens->select(t | t.place`

`= pX).size()`. In Figure 5, this expression is abbreviated with `$pXTokens`. After the markings are initialized, the operation run() is called (cf. Figure 2). This operation iterates through the transitions of the net and checks whether they are enabled. Therefore, in the first iteration, isEnabled() is called on transition t1, which in turn iterates through all of its incoming places (p1 in our example) and checks whether there is an incoming place without tokens. Therefore, the symbolic value p1.tokens is accessed and the condition `p1.tokens == 0` is evaluated. We do not interfere with the concrete execution except for the access of the symbolic value and the evaluation of the condition in order to record the path condition, update the execution tree (cf. Figure 5), and solve the constraint to compute concrete values for the involved symbolic values inducing the `true` branch and the `false` branch, respectively. After that, we continue with the concrete execution with the respective concrete values for both branches. Depending on which branch is taken (i.e., t1 is enabled or not), t1 is added to the output expansion node enabled of the expansion region in the activity run (cf. Figure 2). In the symbolic execution of activities, we handle expansion nodes as list variables. As the addition of t1 to the expansion node depends on symbolic values, we also consider the list variable, denoted with `$enabled`, as symbolic. In the next iteration of run, the same procedure is applied to transition t2 and its input place p4; thus, the execution tree is updated accordingly. Next, the execution checks whether the list of enabled transitions contains at least one element with the condition `$enabled.size > 0`. As this condition accesses `$enabled`, which is considered as symbolic value, we record it in the execution tree and try to produce values for the `true` and `false` branch. However, the constraint solver cannot find a solution for the `false` branch, denoted with $\perp$ in Figure 5, because in three paths `$enabled` will always contain at least one transition according to the path conditions and symbolic states. Thus, in three of the six branches, fire() is called for the first transition in the list `$enabled` causing changes in the tokens attribute of the incoming and outgoing places. As the tokens attribute is considered as symbolic, we update their symbolic states. Finally, the execution proceeds with iterating through transitions again and firing them, if they are enabled. As we bound the symbolic execution to at most one initial token per place, all branches either terminate eventually or lead to an unsatisfiable state (e.g., violating the bound condition).

The final execution tree contains four satisfiable execution paths. The path conditions of these paths represent symbolically all relevant initial token markings for this net inducing all distinct execution traces. Using a constraint solver, we can generate Token objects with corresponding links to the places in the net such that the initial markings of the four inputs are: $\{p1 = 1\}$, $\{p1 = 1, p4 = 1\}$, $\{p4 = 1\}$, and $\{\}$ (no tokens at all). When repeating the symbolic execution for the Petri net $PN1$ in Figure 4, we obtain two additional inputs: $\{p1 = 1, p2 = 1\}$ (or p3 instead of p2), and $\{p1 = 1, p2 = 1, p3 = 1\}$. With this total of six inputs, we invoke the semantic differencing based on the semantic match rules to obtain all diff witnesses (cf. Section 4).

6 Evaluation

We evaluated our approach regarding three aspects. First, we investigate whether our generic approach is powerful enough to specify semantic diff operators equivalent to

those specifically designed for particular modeling languages. Second, we examine the performance of applying semantic diff operators realized with our approach. Third, we evaluate the feasibility of realizing symbolic execution of models based on an operational semantics specification, in particular, using fUML.

Expressive power of generic semantic differencing. To assess whether our generic approach provides the necessary expressive power to define non-trivial semantic diff operators, we carried out two case studies, in which we implemented diff operators for UML activity diagrams and class diagrams according to ADDiff [17] and CDDiff [18] developed by Maoz *et al.* This allows us to evaluate whether our generic approach is powerful enough to compare to one of the most sophisticated language-specific approaches in the semantic differencing domain. Therefore, we implemented the same semantics of UML activity diagrams and class diagrams as defined by Maoz *et al.* using xMOF. While the focus of the evaluation lies on the expressive power of our approach regarding the definition of diff operators, the following figures shall indicate, that the semantics specifications developed for the case studies are of high complexity. The semantics specification of activity diagrams comprises 38 configuration classes consisting of 60 activities in total, which contain altogether 380 activity nodes. The semantics specification of class diagrams comprises 56 configuration classes, 119 activities, and 1003 nodes. For realizing the semantic diff operators we implemented semantic match rule corresponding to the semantic differencing algorithms defined by Maoz *et al.* They enabled us to detect the same diff witnesses as Maoz *et al.* among their case study example models published in [17,18]. The interested reader can find our case studies at our project website [21]. For brevity, we discuss only the results of the case studies in the following.

From the case studies, we conclude that the expressive power of our generic semantic differencing approach is sufficient for defining non-trivial semantic diff operators. Implementing the semantic match rules requires besides knowledge about model comparison languages, such as ECL, knowledge about the behavioral semantics specification of the considered modeling language. Hence, defining semantic diff operators is a task that has to be performed by someone experienced with language engineering.

Performance of applying the semantic matching. We measured the performance of the implemented diff operators in terms of time needed for evaluating whether the example models are semantically different for a given set of input values. This experiment was performed on a Intel Dual Core i5-2520M CPU, 2.5 GHz, with 8 GB RAM, running Windows 7. Table 1 shows the time needed for syntactic matching ($SynMatching$), model execution ($Execution$), and semantic matching ($SemMatching$), as well as the total time needed ($Total$). Please note that these figures do not include the time needed for generating all relevant inputs because an implementation of symbolic model execution for fUML is not integrated yet with our semantic differencing prototype. For activity diagrams, Table 1 also provides the number of activity nodes ($\#Nodes$), as well as the number of input values to consider ($\#Inputs$), as they have a significant influence on the execution time. For class diagrams, we provide the number of objects ($\#Objects$) as well as links ($\#Links$) of the input object diagram. The performance results indicate that the model execution is the most expensive step in the semantic model differencing as it takes around 95% of the overall time. Thus, the main reason

Table 1. Performance results for semantically differencing example models

UML activity diagram

Example	#Nodes	#Inputs	SynMatching	Execution	SemMatching	Total
Anon V1/V2	15/15	2	51 ms	7905 ms	341 ms	8297 ms
Anon V2/V3	15/19	1	72 ms	7374 ms	246 ms	7692 ms
hire V1/V2	14/15	1	47 ms	5259 ms	283 ms	5589 ms
hire V2/V3	15/15	1	47 ms	5745 ms	304 ms	6096 ms
hire V3/V4	15/15	1	48 ms	2011 ms	95 ms	2154 ms
IBM2557 V1/V2	18/16	3	85 ms	25889 ms	1159 ms	27133 ms

UML class diagram

Example	#Objects	#Links	SynMatching	Execution	SemMatching	Total
EMT V1/V2	2	1	17 ms	1203 ms	119 ms	1339 ms
EMT V1/V2	4	3	16 ms	6790 ms	275 ms	7081 ms
EMT V1/V2	6	5	15 ms	26438 ms	543 ms	26996 ms

for the weaker performance result compared to the approach of Maoz *et al.* [17,18] is the performance of the model execution carried out by the fUML virtual machine.

Symbolic execution of fUML. An important prerequisite for realizing symbolic execution is the identification of conditional executions, as well as the extraction of the condition in terms of a quantifier free path constraint. Therefore, we analyzed the feasibility of identifying and mapping conditional language concepts of fUML's action language to corresponding OCL path constraint templates. In this analysis, we faced several challenges, since the execution flow is driven by offering and accepting object tokens through input and output pins of actions; if an action, e.g., a ReadStructuralFeatureValueAction, reads a non-existing value, no token is placed on its output pin, which in turn prevents the execution of the action that waits for the token. Thus, actions that take inputs from actions reading symbolic values, have to be considered as conditional and, therefore, a dedicated path constraint has to be generated. Besides, we have to map the different ways of using DecisionNodes to path constraints, which was, however, mostly straightforward. Nevertheless, we were able to map the fUML concepts used in the case studies to corresponding path conditions. Moreover, in the semantics specifications of our two case studies, we only used the standardized primitive operators and behaviors of fUML, such as greater than, logical AND, list size, etc. As all of these operators and behaviors are supported in OCL, it is straightforward to represent the symbolic state of symbolic variables entirely in OCL, which enabled us to avoid suffering from symbolic imprecision. For realizing generalized concolic execution [29,11] of fUML, it is moreover crucial to distinguish among concrete and symbolic values and, therefore, to extract a value dependency graph for determining whether a value depends directly or indirectly on a symbolic value. Therefore, we performed an experimental implementation for analyzing the dependencies of values, which is, in comparison to usual programming languages, easily possible thanks to the explicit representation of the object flow in fUML. The support for explicating data dependencies has been integrated in our execution trace for fUML [19].

Constraint solving with symbolic execution paths. To enable finding concrete objects and links that satisfy a sequence of path conditions, we implemented an integration of xMOF and fUML with the model validator plug-in for USE [13] by Kuhlmann *et al.*

Therefore, we translate the (configuration) metamodel of the modeling language into a UML class diagram in USE, transform the models to be compared into object diagrams, and represent path constraints as OCL invariants. Now, we may start the model validator plug-in, which internally translates those diagrams and the constraints into the relational logics of Kodkod [30] to apply an efficient SAT-based search for an object diagram that satisfies all constraints. Note that constraint solving is an inherently computation-intensive task. Although our experiments showed that this plug-in of USE is comparably efficient, the constraint solving took up to several seconds in our case studies, which may impair the runtime of symbolic execution drastically. Nevertheless, the runtime of the constraint solving significantly depends on the bounds (e.g., number of objects per class, range of Integer values); thus, there is a large potential for optimizing the constraint solving by extracting heuristics from the operational semantics specification to limit the bounds automatically. Moreover, a feature of this plug-in that significantly improved its application for our purpose is its support for *extending* specified object diagrams *incrementally* for validating additional constraints.

7 Conclusion

We proposed a generic semantic model differencing approach that—in contrast to existing approaches—makes use of the behavioral semantics specifications of modeling languages for supporting the semantic comparison of models. Thus, non-trivial transformations into a semantic domain specifically for enabling semantic differencing can be avoided. Instead, the behavioral semantics specifications of modeling languages, which may also be employed, e.g., for simulation, are reused to enable semantic differencing.

We showed how our approach can be realized for the operational semantics specification approach xMOF to enable the implementation of semantic diff operators. Furthermore, we presented a solution for generating relevant inputs required for the underlying model execution based on symbolic execution. The evaluation of our approach with two case studies revealed that our approach is expressive enough to define different semantic equivalence criteria for specific modeling languages. However, it also turned out that we face serious performance issues caused by the slow model execution, but also by the inherently time-consuming constraint solving task in the symbolic execution. To address this issue we plan to improve the virtual machine for executing models, but also envision an adaptation of *directed and differential symbolic execution* [15,27] for generating relevant inputs more efficiently. One idea behind those symbolic execution strategies is to consider syntactic differences in the models to be compared and direct the symbolic execution towards those differences, whereas unchanged parts are pruned, if possible. Moreover, we will investigate whether it is possible to avoid the generation of concrete inputs at all and, instead, analyze the symbolic representations of the execution trees of both models directly for reasoning about semantic differences.

Acknowledgments. This work is partly funded by the European Commission under the ICT Policy Support Programme grant no. 317859 and by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the FFG BRIDGE program grant no. 832160.

References

1. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
2. Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional 9(2), 29–34 (2008)
3. Cadar, C., Sen, K.: Symbolic Execution for Software Testing: Three Decades Later. Communications of the ACM 56(2), 82–90 (2013)
4. Clarke, L.A.: A Program Testing System. In: Proceedings of the 1976 Annual Conference (ACM 1976), pp. 488–491. ACM (1976)
5. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
6. Esparza, J., Nielsen, M.: Decidability Issues for Petri Nets. Technical Report, BRICS RS948, BRICS Report Series, Department of Computer Science, University of Aarhus (1994), `http://www.brics.dk`
7. Fahrenberg, U., Acher, M., Legay, A., Wasowski, A.: Sound Merging and Differencing for Class Diagrams. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 63–78. Springer, Heidelberg (2014)
8. Fahrenberg, U., Legay, A., Wasowski, A.: Vision Paper: Make a Difference (Semantically). In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 490–500. Springer, Heidelberg (2011)
9. Gerth, C., Küster, J.M., Luckey, M., Engels, G.: Precise Detection of Conflicting Change Operations Using Process Model Terms. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 93–107. Springer, Heidelberg (2010)
10. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? Computer 37(10), 64–72 (2004)
11. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
12. Kolovos, D., Rose, L., García-Domínguez, A., Paige, R.: The Epsilon Book (March 2014), `https://www.eclipse.org/epsilon/doc/book/`
13. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011)
14. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-Specific Models. European Journal of Information Systems 16(4), 349–361 (2007)
15. Ma, K.-K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed Symbolic Execution. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 95–111. Springer, Heidelberg (2011)
16. Maoz, S., Ringert, J.O., Rumpe, B.: A Manifesto for Semantic Model Differencing. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 194–203. Springer, Heidelberg (2011)
17. Maoz, S., Ringert, J.O., Rumpe, B.: ADDiff: Semantic Differencing for Activity Diagrams. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011), pp. 179–189. ACM (2011)
18. Maoz, S., Ringert, J.O., Rumpe, B.: CDDiff: Semantic Differencing for Class Diagrams. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 230–254. Springer, Heidelberg (2011)

19. Mayerhofer, T., Langer, P., Kappel, G.: A Runtime Model for fUML. In: Proceedings of the 7th Workshop on Models@run.time (MRT) co-located with the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS 2012), pp. 53–58. ACM (2012)
20. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: Executable DSMLs Based on fUML. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 56–75. Springer, Heidelberg (2013)
21. Moliz project, http://www.modelexecution.org
22. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving Executability into Object-Oriented Metalanguages. In: Briand, L.C., Williams, C. (eds.) MODELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
23. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1 (August 2011), http://www.omg.org/spec/UML/2.4.1
24. Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0 (February 2011), http://www.omg.org/spec/FUML/1.0
25. Ohst, D., Welle, M., Kelter, U.: Differences Between Versions of UML Diagrams. SIGSOFT Software Engineering Notes 28(5), 227–236 (2003)
26. Oliveira, H., Murta, L., Werner, C.: Odyssey-VCS: A Flexible Version Control System for UML Model Elements. In: Proceedings of the 12th International Workshop on Software Configuration Management (SCM 2005), pp. 1–16. ACM (2005)
27. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential Symbolic Execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2008), pp. 226–237. ACM (2008)
28. Reiter, T., Altmanninger, K., Bergmayr, A., Schwinger, W., Kotsis, G.: Models in Conflict – Detection of Semantic Conflicts in Model-based Development. In: Proceedings of the 3rd International Workshop on Model-Driven Enterprise Information Systems (MDEIS) co-located with the 9th International Conference on Enterprise Information Systems (ICEIS 2007), pp. 29–40 (2007)
29. Sen, K.: Concolic Testing. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pp. 571–572. ACM (2007)
30. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
31. Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-oriented Design Differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), pp. 54–65. ACM (2005)