Semantic Differencing of Statecharts for Object-Oriented Systems

Imke Drave, Robert Eikermann, Oliver Kautz, Bernhard Rumpe
Software Engineering, RWTH Aachen University, Aachen, Germany
{*surname*}*@se-rwth.de*

Keywords: semantics, semantic differencing, model reasoning, statecharts, UML.

Abstract: Statecharts are well-suited to describe the behavior of objects in object-oriented systems. Effective statechart evolution analysis for detecting errors and exploring design alternatives requires to reveal the semantic differences from one statechart to another. Previous work has not produced approaches to semantic differencing of statecharts for object-oriented systems. This paper defines a syntax and a semantics for reduced UML/P statecharts and presents an explicit semantics-preserving translation from statecharts to finite automata. This enables to reduce semantic statechart differencing to language inclusion checking between finite automata. The translation is formally defined and proven to be correct. This paper further introduces operators for hiding and matching events to compare statecharts semantically on different levels of abstraction. The result is a sound and complete method for semantic differencing of statecharts and abstraction methods, which ultimately facilitates semantic statechart evolution analysis.

1 INTRODUCTION

Statecharts model the behavior of reactive systems and have originally been introduced in (Harel, 1987). The UML/P (Rumpe, 2016) is a variant of the UML targeting implementability and code generation. The UML/P focuses on modeling object-oriented systems and is specifically suited for Java as target language.

The behavior of an UML/P statechart models a high-level view on the behavior of objects of a class. With this, it abstracts from low-level implementation details. The states of the statechart correspond to control states of objects. The behaviors modeled by a statechart abstract from data states. Nevertheless, data states can be manipulated within reactions of a statechart but they do not influence the statechart's behavior with respect to its control flow.

We formally define a reduced abstract syntax for UML/P statecharts and a semantics mapping based on possible execution traces, which represent the behaviors of the statechart. We assume that the set of a statechart's control states is representable by a finite set and that each statechart only contains finitely many transitions. The semantic difference from one statechart SC_1 to another statechart SC_2 is defined as the set $\delta(SC_1, SC_2)$ of all behaviors in the semantics of SC_1 that are no members in the semantics of SC_2. Each behavior $b \in \delta(SC_1, SC_2)$ is thus a behavior modeled in statechart SC_1 that is not modeled in statechart SC_2. The behavior b is interpretable as a possible behavior of an object that behaves as specified with SC_1, which is no behavior of any object that behaves as specified with SC_2. It represents a possible observation of reactions to environmental inputs (method calls) of an object. This paper further defines a translation from statecharts to finite automata. The translation enables to reduce semantic differencing of statecharts to language inclusion checking between the automata, which is a well-known decidable problem (Hopcroft et al., 2006). Two statecharts may be inequivalent on one level of abstraction but equivalent on another. We present two statechart abstraction methods that enable to compare statecharts semantically on different levels of abstraction.

The next section recaps details about UML/P statecharts in the context of modeling object oriented systems. Section 3 presents a motivating example for semantic differencing of statecharts. Subsequently, Section 4 presents formal preliminaries grounding our approach. Then, Section 5 presents a reduced abstract syntax and a semantics for statecharts. Based on this, Section 6 presents a method for semantic differencing of statecharts based on a translation to finite automata. Then, Section 7 presents abstraction methods. Finally, Section 8 discusses related work, before Section 9 concludes.

[DEKR19] I. Drave, R. Eikermann, O. Kautz, B. Rumpe:
Semantic Differencing of Statecharts for Object-oriented Systems.
In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19), 2019.
www.se-rwth.de/publications

2 UML/P STATECHARTS

UML/P statecharts (Rumpe, 2016) are intended to describe the behavior of objects in object-oriented systems. States in UML/P statecharts are interpreted as control states (Rumpe, 2016). With this, a statechart abstracts from the actual data states of objects during runtime. An object's state is divided into its control state and its data state, determined by the values assigned to its attributes. UML/P statecharts abstract from implementation details by abstracting from the possibly infinite set of actual data states with finitely many control states. Each control state represents a class of possibly infinitely many data states. In statecharts, initial states correspond to the object's initial control state after the constructor has been called. As modern garbage collectors delete detected unreachable objects, there are no explicit final states in UML/P statecharts (Rumpe, 2016). Transitions between states correspond to the execution of methods. A transition is taken upon the occurrence of its *stimulus* event, provided that the object's control state is equal to the transition's source state. In UML/P statecharts, the occurrence of a stimulus event corresponds to calling a method of an object (including its arguments). The reaction of the object to a method call is modeled by the transition's *reaction*. A reaction is a sequence of statements, *e.g.,* method calls or attribute assignments, which are executed in the body of the method corresponding to the stimulus (Rumpe, 2016). UML/P statecharts abstract from object recursion, *i.e.,* transitions including reactions that call other public methods (stimuli) of the same object. It may nevertheless appear in private methods as long as the statechart's control state is not manipulated. The above interpretation of statecharts and the abstraction from recursion enable a simplifying transformation to a statechart normal form as described in (Rumpe, 2016). This transformation eliminates, for example, hierarchically nested states. We will define an abstract statechart syntax based on the simple normal form to facilitate a semantics mapping definition.

3 EXAMPLE

Webshops usually provide virtual carts. Customers add items to a cart by selecting items offered on product sites. Figure 1 depicts a statecharts describing a webshop system's behavior when processing orders. On the left, statechart SC models the behavior of objects of the Order class specified by the class diagram in the middle. Our analysis performs a transformation into a nondeterministic finite automaton as shown on the right of Figure 1. After construction, an order object resides in the Created-state, where it is not yet paid. A place-order button executes the place() method, which triggers the execution of requestPay(price) setting the paid attribute of the order, before it takes the Placed state. Once placed, the customer may either cancel or re-place() and pay the order. Shipment is initiated by tryShip(), which will only trigger the transition to Completed if paid==true, since the webshop explicitly requires that only paid orders are shipped. If so, the system executes ship(), which sends the ordered items to the customer. In case a customer cancels her order, a cancellation request is sent to the system. As modeled by SC, canceling an order is also possible after the order has already been shipped. A webshop developer now notices that canceling orders after shipping lead to negative ratings for the webshop. The developer decides to inhibit canceling shipped orders. To this effect, she modifies the statechart SC by replacing the requestCancel() reaction in state Completed by a new method illegalCancel(), which notifies the customer that a cancellation of the order is not possible. The statechart is updated to suit the new requirement. The developer wants to understand how the syntactic change influenced the statechart's semantics. She thus uses our method for semantic statechart differencing. It turns out that the updated version of SC has incomparable semantics to the original, *i.e.,* there exists a behavior modeled in SC that is not possible after the update and vice versa. The developer is presented with two witnesses that are concrete examples of semantic differences: A witness for the semantic difference from the original statechart to the statechart after replacing requestCancel() by illegalCancel() is (place(), paid=requestPay(price)), (tryShip(), ship()), (cancel(), requestCancel()). Vice versa, a witness for the semantic difference from the changed SC to the original SC is the behavior (place(), paid=requestPay(price)), (tryShip(), ship()), (cancel(), illegalCancel()).

4 PRELIMINARIES

The empty word is denoted by ε. An alphabet is a non-empty finite set A satisfying $\varepsilon \notin A$. A^* denotes the set of all finite sequences (words) over an alphabet A. For every alphabet A, $\varepsilon \in A^*$ holds. Let $u, v \in A^*$ be words. $u \cdot v$ (or uv) denotes the result of appending the word v to the word u. $|u|$ denotes the length of u where $|\varepsilon| = 0$. $u.i$ denotes the $(i+1)$-th symbol in u with $|u| > i$. For a symbol $s \in A$, we denote by $s : u$ the result of prepending s to u. $d(u)$ denotes the result from deleting the first symbol from $u \neq \varepsilon$. For a set of symbols $B \subseteq A$, we denote by $u \downarrow B$ the result of removing all occurrences of the symbols contained in

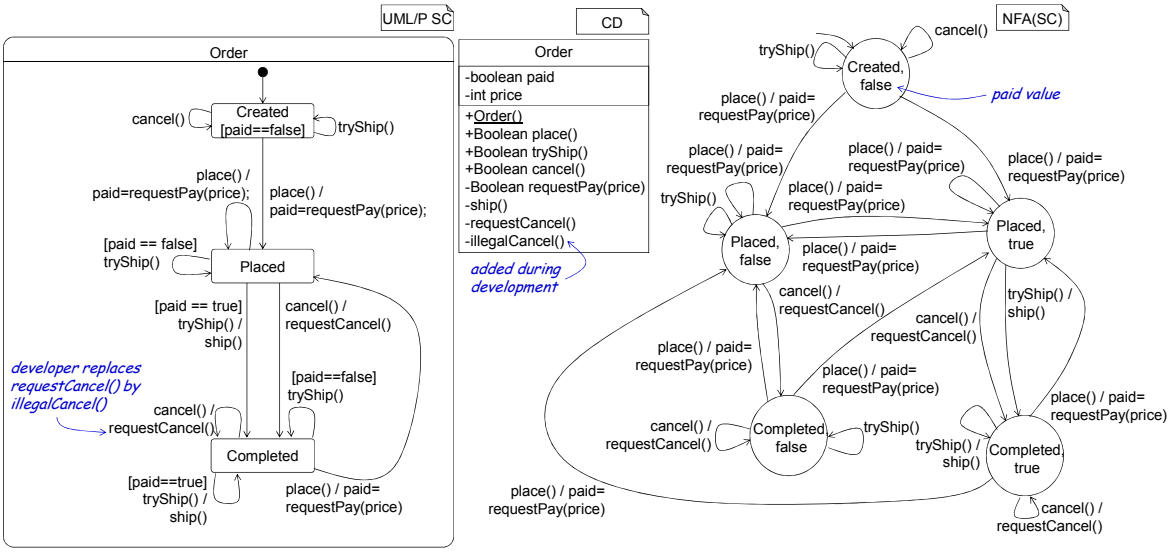

Figure 1: Statechart modeling the behavior of a webshop system, the associated class, and the corresponding nondeterministic finite automaton (NFA).

B from u. For all $u \in A^*$ and $B \subseteq A$, the operator is recursively defined by the following equation:

$$u \downarrow B = \begin{cases} \varepsilon & \text{if } u = \varepsilon \\ u.0 : (d(u) \downarrow B) & \text{if } |u| > 0 \wedge u.0 \notin B \\ d(u) \downarrow B & \text{if } |u| > 0 \wedge u.0 \in B \end{cases}$$

We lift functions $m : A \to A$ mapping elements of a set A to elements of the same set to words $u \in A^*$ over A in a point-wise manner, *i.e.,* $m(\varepsilon) = \varepsilon$ and $m(u) = m(u.0) : m(d(u))$ for all $\varepsilon \neq u \in A^*$.

An nondeterministic finite automaton (NFA) (*e.g.,* (Hopcroft et al., 2006)) is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, Q_0 is a set of initial states, and F is a set of final states. Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. A run of A on a word $w = w_1, w_2, ..., w_n \in \Sigma^*$ is a finite sequence of states $r = r_0, r_1, ..., r_n \in Q^*$ satisfying $r_0 \in Q_0$, $r_n \in F$ and $(r_i, w_{i+1}, r_{i+1}) \in \delta$ for all $0 \leq i < n$. The NFA A accepts a word w iff a run of A on w exists. The language recognized by A is defined as $\mathcal{L}(A) = \{w \in \Sigma^* \mid A \text{ accepts } w\}$, *i.e.,* all words accepted by A. There are well-known algorithms to check whether the language recognized by one NFA is included in the language recognized by another NFA (Hopcroft et al., 2006).

5 STATECHART SYNTAX AND SEMANTICS

A complete definition of a modeling language consists of the following parts (Cengarle et al., 2009; Grönniger and Rumpe, 2011): Concrete Syntax, abstract syntax, reduced abstract syntax, semantic domain, and semantic mapping.

The *concrete syntax* is a model's representation that the user interacts with. A definition of the graphical and a textual concrete syntax for statecharts as used in this paper is available in (Rumpe, 2016). This paper's approach abstracts from a concrete syntactic representation. The *abstract syntax* is a reduced and compact representation of the concepts that can be modeled using the concrete syntax. For every model in the concrete syntax, there must exist a model in the abstract syntax that the model is mapped to. Different models in the concrete syntax may be mapped to the same model in the abstract syntax. The *reduced abstract syntax* is a subset of the abstract syntax of a modeling language. It only includes a subset of the abstract syntax without losing expressibility. Normal forms of propositional formulas are examples for reduced abstract syntaxes. A reduced abstract syntax eases the definition of the semantics of a language (Cengarle et al., 2009). Section 5.1 presents a reduced abstract syntax for UML/P statecharts. A translation from the abstract syntax (including hierarchy, OR-states, epsilon-transitions, entry and exit actions) to the reduced abstract syntax is described in (Rumpe, 2016). The *semantic domain* contains the elements of the semantics of a model. In this paper, the semantic domain for statecharts is a set of all finite sequences of stimuli/reaction tuples. Each sequence represents a behavior as a trace of method calls. The stimuli represent methods called on an object of a statechart's class and the reactions contain method calls performed by the object. The *semantic*

mapping relates the elements of the reduced abstract syntax to elements of the semantic domain. In this paper, the semantic mapping maps each statechart to the set of possible behaviors of its class's objects.

5.1 Reduced Abstract Syntax

State hierarchies in UML/P statecharts can be eliminated by a transformation (Rumpe, 2016). Therefore, in the following we assume statecharts to be flat and their set of control states to be finitely representable. Thus, the types of variables used on guards must also be finitely representable. Examples for finite types are the primitive types short, int, and boolean. Let $\mathcal{E}$ denote a countable infinite set of events. For example, the set $\mathcal{E}$ may contain all expressions for calling methods or attribute assignments. The following defines the reduced abstract syntax for flat statecharts:

Definition 1 (Statechart)**.** *A statechart is a tuple* (I, O, S, T, S_0) *where*

- $I, O \subset \mathcal{E}$ *with* $I \cap O = \emptyset$ *are finite sets of input and output events,*
- S *is a finite set of states,*
- $T \subseteq S \times S \times I \times O^*$ *is a finite set of transitions,*
- $S_0 \subseteq S$ *is a set of initial states.*

Notation. Let $t = (src, tgt, st, ev) \in T$ be a transition. Then, $src(t) = src$ denotes the transition's source state, $trg(t) = tgt$ denotes the transition's target state, $st(t) = st$ denotes the transition's stimulus, and $r(t) = ev$ denotes the transition's reaction.

We assume that each statechart models the behavior of instances of a class in an object-oriented system. Each state of the statechart is a control state that represents a possibly infinite set of actual data states. For instance, a statechart's class may contain several attributes. The statechart abstracts from the data states of objects of the class (*i.e.,* assignments to the attributes) as it only contains control states. Although the data state may be manipulated via executing transitions, it never influences the behavior of the statechart. If the assignment to an attribute is meant to influence a statechart's behavior, it must be explicitly encoded as control state. The set of initial states models the possible initial control states of objects of the statechart's class after their creation. The set of input events of a statechart contains expressions representing the public methods of the statechart's class. Output events represent statements in the bodies of these methods. These are, for example, statements for calling private methods of the same class, statements for calling methods of attributes of the class, or attribute assignments. Recursion in statecharts induces several semantic ambiguities (Rumpe, 2016). Therefore, the set of input events is required to be disjoint to the set of output events, which prevents recursion on statechart model level. However, recursive method calls may appear, for example, inside private methods.

5.2 Behavior Semantics

UML/P statecharts can be interpreted as NFAs with input and output alphabets. With this, the execution of a transition is an atomic process, which cannot be interrupted: When a statechart receives an event, it takes a transition with the corresponding stimulus and must execute the transition's reaction before receiving the next event is possible. Immediately after executing the reaction, the statechart switches its state to the transition's target state. Further, in UML/P statecharts, events in reactions of transitions cannot trigger further transitions, *i.e.,* no recursion is allowed on statechart level. The following definition formalizes the intuition behind the notion of execution:

Definition 2 (Execution)**.** *An execution of a statechart* $SC = (I, O, S, T, S_0)$ *is a finite alternating sequence* $s_0, t_1, s_1, t_2, ..., s_n$ $(n \geq 0)$ *of states and transitions ending in a state such that*

1. $s_0 \in S_0$*,*
2. $s_i \in src(t_{i+1})$ *for all* $0 \leq i < n$*, and*
3. $s_i \in trg(t_i)$ *for all* $0 < i \leq n$*.*

$Ex(SC)$ *denotes the set of all executions of SC.*

With the above definition, a statechart execution starts in one of the statechart's initial states. Then, the statechart iteratively executes a transition starting at its current state and then enters the transition's target state. A transition may end in any of the statechart's states, *i.e.,* there are no explicit final states. This reflects that unreferenced objects in an object-oriented system may be garbage collected, which is a process that does not depend on an object's state.

Executions comprise the internal state changes of a statechart that are invisible to its environment. The behavior of an execution only captures the visible events while the statechart performs the execution:

Definition 3 (Behavior)**.** *Let* $SC = (I, O, S, T, S_0)$ *be a statechart. The behavior* $beh(ex)$ *of an execution* $ex = s_0, t_1, s_1, t_2, ..., s_n \in Ex(SC)$ *of the statechart SC is defined as the sequence* $beh(ex) = (st(t_1), r(t_1)), ..., (st(t_n), r(t_n)) \in (I \times O^*)^*$.

With the above, a behavior of an execution is the sequence of tuples of stimuli and reactions of the transitions occurring in the execution. The semantics of a statechart is the set of all behaviors of the statechart.

Definition 4. *The semantics* $[\![SC]\!]$ *of a statechart SC is the set of all behaviors of all executions of SC, i.e.,* $[\![SC]\!] = \{beh(ex) \mid ex \in Ex(SC)\}$.

6 SEMANTIC DIFFERENCING OF STATECHARTS

This section formally introduces semantic differencing of statecharts and presents a method that can be performed on push-button basis.

Definition 5 (Semantic Difference)**.** *The semantic difference of a statechart SC_1 to a statechart SC_2 is defined as $\delta(SC_1, SC_2) = [\![SC_1]\!] \setminus [\![SC_2]\!]$.*

With this, the semantic difference contains all behaviors modeled in one statechart that are not modeled in another statechart. If $\delta(SC_1, SC_2) = \emptyset$, then the Statechart SC_1 is a refinement of the statechart SC_2, *i.e.,* SC_1 has no behavior that SC_2 does not have. Therefore, the former statechart contains less underspecification as the latter. In model-driven development, statecharts are naturally refined in reaction to receiving additional requirements.

As the semantics of a statechart is typically an infinite set, checking whether $\delta(SC_1, SC_2) = \emptyset$ is, in general, not trivial. In the following, we present a sound and complete method to check whether one statechart refines another statechart. In case of non-refinement, the method yields an element $w \in \delta(SC_1, SC_2)$, which is a concrete witness for a behavior possible in the one statechart that is not possible in the other statechart. To this effect, we present a translation from statecharts to NFAs and reduce statechart refinement checking to language inclusion checking between NFAs. As language inclusion checking between NFAs is fully automatable, we obtain an automated method for semantic statechart differencing.

6.1 Mapping Statecharts to Automata

For every statechart, there exists an NFA such that the language accepted by the NFA is equal to the set of behaviors of the statechart. The following definition explicates the construction of such an NFA:

Definition 6. *Let $SC = (I, O, S, T, S_0)$ be a statechart. The NFA $NFA(SC)$ associated to SC is defined as $NFA(SC) = (S, \Sigma, \delta, S_0, S)$ where*

- $\Sigma = I \times (\bigcup_{t \in T} \{r(t)\})$,
- $\delta = \{(u, (st, r), v) \mid (u, v, st, r) \in T\} \subseteq S \times \Sigma \times S$.

As T is finite, $\bigcup_{t \in T} \{r(t)\}$ is also finite. Thus, as additionally I is finite, Σ is finite. Further S and $S_0 \subseteq S$ are finite. Therefore, $NFA(SC)$ is a well-defined NFA.

With Definition 6, every NFA input symbol corresponds to a stimulus/reaction tuple of the statechart. The recognized language of an NFA associated to a statechart equals the statechart's semantics.

Theorem 1. *For every statechart SC, it holds that $[\![SC]\!] = \mathcal{L}(NFA(SC))$.*

Proof. Let $SC = (I, O, S, T, S_0)$ be a statechart and let $NFA(SC) = (S, \Sigma, \delta, S_0, S)$.

$\subseteq$: Let $e = s_0, t_1, s_1, t_2, ..., s_n \in Ex(SC)$ be an execution of SC. We need to show that $beh(e) \in \mathcal{L}(NFA(SC))$. By definition of execution, we have:

1. $s_i \in src(t_{i+1})$ for all $0 \leq i < n$, and
2. $s_i \in trg(t_i)$ for all $0 < i \leq n$.

From this and the definition of δ, we can conclude $(s_i, (st(t_{i+1}), r(t_{i+1})), s_{i+1}) \in \delta$ for all $0 \leq i < n$. By definition of execution, it holds that $s_0 \in S_0$. As every state in $NFA(SC)$ is accepting, we can conclude that $s_0, s_1, ..., s_n$ is a run of $NFA(SC)$ on the word $beh(e) = (st(t_1), r(t_1)), ..., (st(t_n), r(t_n)) \in \mathcal{L}(NFA(SC))$.

$\supseteq$: Let $r = r_0, r_1, ..., r_n \in S^*$ be a run of A on a word $w = (a_1, b_1), (a_2, b_2), ..., (a_n, b_n) \in \Sigma^*$. We need to show that $w \in [\![SC]\!]$. As r is a run on w, it holds that $r_0 \in S_0$ and $(r_i, (a_{i+1}, b_{i+1}), r_{i+1}) \in \delta$ for all $0 \leq i < n$. By definition of δ, this implies $(r_i, r_{i+1}, a_{i+1}, b_{i+1}) \in T$ for all $0 \leq i < n$. Now let $t_{i+1} = (r_i, r_{i+1}, a_{i+1}, b_{i+1})$ for all $0 \leq i < n$. This above implies that $r_i \in src(t_{i+1})$ for all $0 \leq i < n$. Further, the above implies $r_i \in trg(t_i)$ for all $0 < i \leq n$. By definition of execution, we can conclude that $e = r_0, t_1, r_1, t_2, ..., r_n$ is an execution of SC. As e is an execution of SC, it holds that $beh(e) = (st(t_1), r(t_1)), ..., (st(t_n), r(t_n)) \in [\![SC]\!]$ is a behavior of SC. Observing $st(t_i) = a_i$ and $r(t_i) = b_i$ for all $0 < i \leq n$, we obtain $(a_1, b_1), (a_2, b_2), ..., (a_n, b_n) \in [\![SC]\!]$. □

Figure 1 shows the reachable part of the NFA associated with the webshop-order statechart introduced Figure 1. The attribute paid used in guards is expanded into the state space. For each of the transitions with the stimulus place() and the reaction paid=requestPay(price) in the statechart, there exist multiple transitions with the respective label from each respective state in the NFA. For each of these states, there exists one transition with the respective label to the NFA state where the value of paid is assigned to false and one transition with the respective label to the NFA state where the value of paid is assigned to true. This is the case, because the value assigned to the paid attribute in the transitions' reactions cannot be predetermined.

With Theorem 1, semantic differencing of statecharts can be reduced to language inclusion checking between NFAs. Thus, reuse of well known algorithms from automata theory is possible (*e.g.,* (Hopcroft et al., 2006)). Given two statecharts SC_1 and SC_2, it holds that $\delta(SC_1, SC_2) = \emptyset$ iff $\mathcal{L}(NFA(SC_1)) \subseteq \mathcal{L}(NFA(SC_2))$. For checking whether there exist elements in the semantic difference of a statechart to

another statechart, we can thus construct the NFAs associated to the statechart before performing language inclusion checking on the NFAs. In case language inclusion checking yields a counterexample in form of a word accepted by the one automaton that is not accepted by the other automaton, the word can be used as a witness for a behavior of the one statechart that is no behavior of the other statechart. The witness can be presented to a user to facilitate finding the syntactic statechart elements causing the presence of witnesses.

7 ABSTRACTION

Two statecharts may be equivalent on one level of abstraction but inequivalent on another abstraction level. A statechart developer may be aware that two statecharts differ on one level of abstraction. At the same time, the developer could be interested in comparing the statecharts on another abstraction level where they might be equivalent. This is especially useful if the developer wants to focus on comparing some aspects of the statecharts but wants to abstract from others.

For example, a developer might be interested in applying semantic differencing to two statecharts while abstracting from calls of private methods in reactions. This corresponds to event hiding as introduced in Section 7.1.

Similarly, at an early development stage, the arguments passed to a method, which is called in a reaction of a transition, might be unknown. In later development stages, the arguments might be known. As a consequence, the developer changes all events that correspond to the method calls with unknown parameters to events where the arguments are all explicated. This implies that the statechart where the arguments are explicated is not equivalent to the statechart where the arguments are not explicated. The developer might want to abstract from the arguments to check whether the two statecharts are equivalent provided that the method is called with the same arguments. This corresponds to event matching as introduced in Section 7.2. Combinations of event hiding and matching methods are also possible.

7.1 Event Hiding

Two statecharts describing the behavior of the same class may behave equally when observed from the outside but may still have incomparable semantics. For example, the statecharts may call different private methods of their class but the same methods of other classes. Sometimes, it is required to abstract from internal method calls to check whether statecharts have the same externally visible behaviors. This can be achieved by hiding:

Definition 7 (Hiding)**.** *Let $SC = (I, O, S, T, S_0)$ be a statechart and let $H \subseteq O$ be a set of output events. Hiding the events of H in SC yields the statechart $SC \Downarrow H = (I, O \setminus H, S, T', S_0)$ where $T' = \{(src, trg, st, re \downarrow H) \mid (src, trg, st, re) \in T\}$.*

Thus, hiding events from a statechart effectively removes them from the reactions of the statechart's transitions. An important property of the hiding operator is that it strictly induces an abstraction with respect to refinement: If one statechart is a refinement of another statechart before hiding events, then the statechart is still a refinement of the other statechart after hiding the same events in both statecharts.

Theorem 2. *Let $SC = (I, O, S, T, S_0)$ and $SC' = (I', O', S', T', S'_0)$ be two statecharts and let $H \subseteq O$ be a set of output events. If $[\![SC]\!] \subseteq [\![SC']\!]$, then $[\![SC \Downarrow H]\!] \subseteq [\![SC' \Downarrow H]\!]$.*

Proof. Let SC, SC', and H be given as above. Assume $[\![SC]\!] \subseteq [\![SC']\!]$ holds.

Let $b \in [\![SC \Downarrow H]\!]$ be a behavior of the statechart $SC \Downarrow H$. Then, there exists an execution $e = s_0, t_1, s_1, ..., s_n \in Ex(SC \Downarrow H)$ such that $beh(e) = b$. By definition of $\Downarrow$, we have that for all $0 < i \leq n$ there exists a transition $t'_i = (src_i, trg_i, st_i, re_i) \in T$ such that $t_i = (src_i, trg_i, st_i, re_i \downarrow H)$. Since e is an execution of $SC \Downarrow H$ and by definition of $\Downarrow$, it follows that $e' = s_0, t'_1, s_1, ..., s_n \in Ex(SC)$ is an execution of SC. Thus, $beh(e') \in [\![SC]\!]$ is a behavior of SC. By assumption $[\![SC]\!] \subseteq [\![SC']\!]$, it holds that $beh(e') \in [\![SC']\!]$. Therefore, there exists an execution $\alpha = s'_0, u_1, ..., s'_n \in Ex(SC')$ such that $beh(\alpha) = beh(e')$. By definition of $\Downarrow$, we obtain that $\alpha' = s'_0, u'_1, ..., s'_n \in Ex(SC' \Downarrow H)$ where $r(u'_i) = r(u_i) \downarrow H$ and $st(u'_i) = st(u_i)$ for all $0 < i \leq n$. Thus, $beh(\alpha') \in [\![SC' \Downarrow H]\!]$. Observing that $r(u'_i) = r(u_i) \downarrow H = re_i \downarrow H$ and $st(u'_i) = st(u_i) = st_i$ for all $0 < i \leq n$, we obtain that $b = (st_1, re_1 \downarrow H), ..., (st_n, re_n \downarrow H) = beh(\alpha') \in [\![SC' \Downarrow H]\!]$. □

7.2 Event Matching

The events used in the transitions of one statechart may contain more details than the events used in the transitions of another statechart. For example, a statechart developed in an early development stage may not contain the exact arguments that are passed to each method. In this case, reactions in the statechart's transitions may include events that represent the name

of the method without providing details about its arguments. These arguments may be specified in a later development stage. As a consequence, the events in the reactions are changed by explicating the details about which arguments are passed to the method. A developer might be interested in whether the changes imply refinement when abstracting from the arguments passed to the method. Then, we must abstract from the method arguments by matching the events including arguments to the events without arguments:

Definition 8 (Matching)**.** *Let $SC = (I, O, S, T, S_0)$ be a statechart and let $m : \mathcal{E} \to \mathcal{E}$ be a function that maps events to other events. Matching the events in SC according to m yields the statechart $SC \oplus m = (I', O', S, T', S_0)$ where*

- $I' = \{m(i) \mid i \in I\}$,
- $O' = \{m(o) \mid o \in O\}$, *and*
- $T' = \{(s, t, m(st), m(re)) \mid (s, t, st, re) \in T\}$.

Thus, matching replaces all occurrences of an event in a statechart by the event that it is mapped to by the matching function. Event matching induces an abstraction in context of statechart refinement: If a statechart refines another statechart before applying a matching, then the statechart obtained from applying the matching to the former statechart refines the statechart obtained from applying the same matching to the latter statechart:

Theorem 3. *Let $SC = (I, O, S, T, S_0)$ and $SC' = (I', O', S', T', S_0')$ be two statecharts and let $m : \mathcal{E} \to \mathcal{E}$ be a function that maps events to other events. If $[\![SC]\!] \subseteq [\![SC']\!]$, then $[\![SC \oplus m]\!] \subseteq [\![SC' \oplus m]\!]$.*

Proof. Let SC, SC' and m be given as above. Assume $[\![SC]\!] \subseteq [\![SC']\!]$ holds. Let $b \in [\![SC \oplus m]\!]$ be a behavior of the statechart $SC \oplus m$. Then, there exists an execution $e = s_0, t_1, s_1, ..., s_n \in Ex(SC \oplus m)$ such that $beh(e) = b$. By definition of $\oplus$, we have that for all $0 < i \leq n$ there exists a transition $t_i' = (src_i, trg_i, st_i, re_i) \in T$ such that $t_i = (src_i, trg_i, m(st_i), m(re_i))$. Since e is an execution of $SC \oplus m$ and by definition of $\oplus$, it follows that $e' = s_0, t_1', s_1 ..., s_n \in Ex(SC)$ is an execution of SC. Thus, $beh(e') \in [\![SC]\!]$ is a behavior of SC. By assumption $[\![SC]\!] \subseteq [\![SC']\!]$, it holds that $beh(e') \in [\![SC']\!]$. Therefore, there exist an execution $\alpha = s_0', u_1, ..., s_n' \in Ex(SC')$ such that $beh(\alpha) = beh(e')$. By definition of $\oplus$, we obtain that $\alpha' = s_0', u_1', ...s_n' \in Ex(SC' \oplus m)$ where $r(u_i') = m(r(u_i))$ and $st(u_i') = m(st(u_i))$ for all $0 < i \leq n$. Thus, $beh(\alpha') \in [\![SC' \oplus m]\!]$. Observing that $r(u_i') = m(r(u_i)) = m(r(t_i')) = m(re_i) = r(t_i)$ and $st(u_i') = m(st(u_i)) = m(st(t_i')) = m(st_i) = st(t_i)$ for all $0 < i \leq n$, we obtain that $b = (st(t_1), r(t_1)), ..., (st(t_n), r(t_n)) = beh(\alpha') \in [\![SC' \oplus m]\!]$. □

8 RELATED WORK AND DISCUSSION

Related work divides into different statechart semantics and behavior specifications as well as model checking of statecharts.

Statecharts have originally been introduced in (Harel, 1987). Since then, many statechart semantics have been proposed. A survey (Eshuis, 2009) compares three established and commonly used semantics: STATEMATE (Harel et al., 1988; Harel and Naamad, 1996; Harel and Politi, 1998), a fixpoint semantics based on (Pnueli and Shalev, 1991), and an UML semantics, which resembles the behavior description for Rhapsody statecharts (Harel and Gery, 1997; Harel and Kugler, 2004).

In general, the three semantics induce different statechart behavior interpretations (Eshuis, 2009). All the three semantics rely on the notion of step. A statechart performs a step in reaction to an occurring input event. In all the three semantics, when an event occurs, a statechart takes a transition, produces several events and may again react to the produced events (either immediately or delayed). This induces several semantical problems (Rumpe, 2016). In contrast, a UML/P statechart always executes exactly one transition in response to an occurring event. One transition cannot trigger another transition by definition. This further eases comprehension and code generation (Rumpe, 2016). Another difference of the three step semantics to the UML/P semantics is that the former permit feedback loops between communicating statecharts, *i.e.,* allow to model recursion. With this, the three step semantics are more tailored towards modeling implementation details, whereas UML/P statecharts abstract from data states and are tailored towards modeling high-level concepts.

In contrast to the statecharts supported by the UML semantics (Eshuis, 2009), the UML/P semantics also support underspecification: With UML/P statecharts, it is possible to model multiple default (initial) states and to model multiple transitions with the same stimulus originating from the same state. In contrast, statecharts supported by the UML semantics are by definition not underspecified. The default states of composite states are always unique. Further, ambiguities in the transition relation are eliminated by using priorities (Eshuis, 2009). Modeling underspecified statecharts is especially useful in early development stages where most of the system requirements are unknown. When additional requirements become known in later development stages, underspecification can be removed by changing the respective statechart syntactically to restrict the possible

system behaviors accordingly. Then, semantic differencing facilitates developers in detecting whether the new statechart version is indeed a refinement of its predecessor version.

A method to verify statecharts against computation tree logic formulas is introduced in (Zhao and Krogh, 2006). The method is generic and thus applicable to a general statechart class. The approach is based on step-semantics and assumes that statecharts exhibit bounded behaviors. Extending the approach could yield semantic differencing methods for statecharts with bounded behaviors under step semantics.

In (Maggiolo-Schettini et al., 1996), statecharts are translated to labeled transition systems, which represent their semantics. Based on the translation, several equivalence relations for statecharts are defined. The semantics used in the approach is a variant of the semantics described in (Pnueli and Shalev, 1991). Transfering the equivalence relations to our approach is interesting for the development of further abstraction methods.

Another approach (Meng et al., 2004) that considers refinement of statecharts uses a coalgebraic semantics inspired by the operational semantics introduced in (Latella et al., 1999). The approach uses another semantics than the one in this paper and does not describe an automated method for checking refinement. However, the paper introduces eight refinement rules that are correct by construction. Developing such rules for statecharts using this paper's semantics is interesting future work.

9 CONCLUSION

This paper formally defined a reduced abstract syntax for UML/P statecharts modeling the behavior of objects in object-oriented systems. We further defined a semantics for statecharts based on the behaviors induced by taking transitions during statechart executions. Subsequently, this paper presented a semantics-preserving translation from statecharts to NFAs. The translation enables to reduce semantic differencing of statecharts to language inclusion checking between NFAs, which is fully automatable. Afterwards, this paper defined event hiding and event matching as two abstraction methods that enable to apply semantic differencing to statecharts on different levels of abstraction. The result is a fully automated semantic differencing method for a statechart variant that supports underspecification. For future work, we plan to investigate the method's performance. The method ultimately facilitates semantic evolution analysis for developers of statecharts specifying the behavior of objects in object-oriented systems.

REFERENCES

Cengarle, M. V., Grönniger, H., and Rumpe, B. (2009). Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*.

Eshuis, R. (2009). Reconciling statechart semantics. *Science of Computer Programming*, 74(3).

Grönniger, H. and Rumpe, B. (2011). Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8.

Harel, D. and Gery, E. (1997). Executable Object Modeling with Statecharts. In *International Conference on Software Engineering*.

Harel, D. and Kugler, H. (2004). The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Applications in Engineering*.

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., and Shtul-Trauring, a. (1988). Statemate: A Working Environment for the Development of Complex Reactive Systems. In *Proceedings of the 10th International Conference on Software Engineering*.

Harel, D. and Naamad, A. (1996). The STATEMATE Semantics of Statecharts. *ACM Transactions on Softwware Engineering Methodology*, 5(4).

Harel, D. and Politi, M. (1998). *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Latella, D., Majzik, I., and Massink, M. (1999). Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Formal Methods for Open Object-Based Distributed Systems*.

Maggiolo-Schettini, A., Peron, A., and Tini, S. (1996). Equivalences of Statecharts. In *International Conference on Concurrency Theory*.

Meng, S., Naixiao, Z., and Barbosa, L. S. (2004). On Semantics and Refinement of UML Statecharts: A Coalgebraic View. In *International Conference on Software Engineering and Formal Methods*.

Pnueli, A. and Shalev, M. (1991). What is in a Step: On the Semantics of statecharts. In *Theoretical Aspects of Computer Software*.

Rumpe, B. (2016). *Modeling with UML: Language, Concepts, Methods*. Springer International.

Zhao, Q. and Krogh, B. H. (2006). Formal Verification of Statecharts Using Finite-State Model Checkers. *IEEE Transactions on Control System Technology*, 14(5).