Semantic Differencing of Activity Diagrams by a Translation into Finite Automata

Oliver Kautz and Bernhard Rumpe

RWTH Aachen University, Aachen, Germany
{surname}@se-rwth.de
http://www.se-rwth.de

Abstract. Activity diagrams are widely used in the business process modeling domain, for software documentation, and code generation. Revealing the semantic differences between activity diagrams is crucial for model evolution analysis to explore design alternatives and to detect errors. Existing approaches to semantic differencing of activity diagrams are based on complex and hard to understand translations to other problem spaces. The translations are implicit in the sense that they transform ADs to a model in the input format of a model checker, which again encodes another formalism. We present a simple and explicit translation from activity diagrams to finite automata and define the semantics of activity diagrams as the words recognized by these automata. This enables semantic differencing of activity diagrams via language inclusion checking between finite automata. We formally define the translation and present an implementation. The resulting simple and explicit translation increases comprehensibility of the semantic differencing operator and reduces implementation efforts.

1 Introduction

Activity diagrams (ADs) describe how an activity or a process needs to be performed. This paper considers an expressive variant of ADs with actions, branching fragments modeled with decision and merge nodes, and parallel fragments modeled with fork and join nodes. We use the standard notations of the UML [5]. An action describes a single task to be executed as part of an activity. Control flow nodes describe alternative (decision, merge) and parallel (fork, join) execution branches within an activity. An initial node marks where an activity starts and a final node marks where an activity ends. Transitions describe how the execution of an AD proceeds after visiting a node. We formally define the abstract syntax of ADs and an operational semantics for ADs via a translation to finite automata. The trace semantics of an AD is then defined as the language recognized by the automaton resulting from translating the AD. The semantic difference between two ADs ad_1 and ad_2 is defined as the set $\delta(ad_1, ad_2)$ of all traces in the semantics of the AD ad_1 that are no members in the semantics of ad_2. It therefore contains all execution traces possible in ad_1 that are not possible in ad_2. When interpreting ad_1 as a successor version of ad_2, the semantic

difference $\delta(ad_1, ad_2)$ contains the execution traces that have been added during the evolution from ad_2 to ad_1. Vice versa, $\delta(ad_2, ad_1)$ contains the execution traces that have been removed during the evolution to ad_1. The semantic difference can therefore effectively be used during AD evolution analysis to detect whether execution traces have been removed or new execution traces have been added between two successor versions. This analysis is in general not possible with syntactic approaches to AD differencing [11], which reveal the syntactic elements that have been changed during the evolution.

In contrast to previous work [3,11,12], this paper's translation explicitly maps ADs to finite automata and does not define an implicit mapping via a translation from an AD to a model in the input format of a model checker, which encodes a finite automaton with a semantics based on recognized words. Through the explicit mapping our translation is more simple and thus easier to understand. The translation is further less complex, which reduces implementation efforts. A main enabler for the complexity reduction is that this paper's approach uses a smaller subclass of UML ADs [5] than previous work [3].

We formally define the translation and present an implementation based on the language workbench MontiCore [7] and the finite automaton language inclusion checking tool RABIT [15]. Our contribution is a method for semantic differencing of ADs based on a simple as well as easy to understand and implement translation from ADs to finite automata.

Section 2 presents motivating examples, before Section 3 introduces preliminaries and notations as used in this paper. Then, Section 4 presents an abstract syntax for ADs, an operational semantics based on finite automata, a denotational semantics based on execution traces, and semantic differencing of ADs. Afterwards, Section 5 presents an implementation of the differencing operator as well as several example applications. Section 7 concludes.

2 Examples

This section motivates this paper's method with four example ADs previously presented in [10,11].

2.1 Example 1

Figure 1 depicts two ADs from [11]. The ADs describe workflows of a company to be executed when hiring new employees. The AD `hire.v1` describes the company's original workflow. The AD `hire.v2` describes a successor version.

In the original workflow, the employee is first registered. Then, the employee either directly gets assigned a project before her payment is authorized, or the employee gets a welcome package before her contact data is added to the company website, she is assigned a project, she is interviewed by the manager, and gets a manager report, before her payment is finally authorized. The actions of adding the employee to the company website and assigning the employee to a project can be executed independent of each other, *i.e.,* there is no strict order

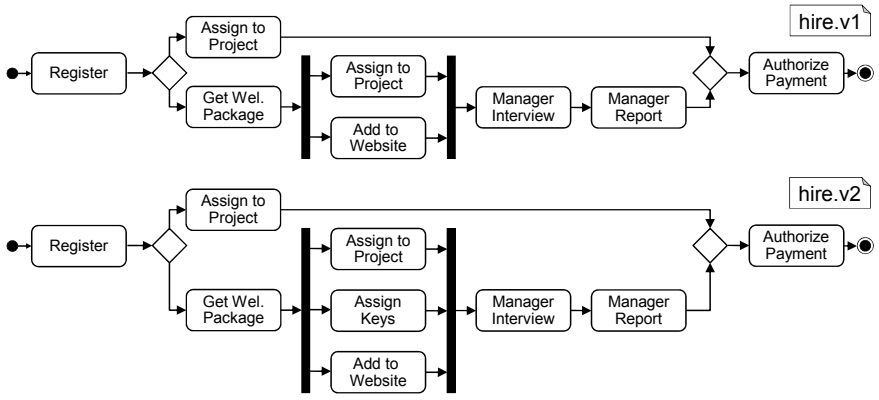

Fig. 1. Two activity diagrams adapted from [11].

in which the two actions have to be executed. The company decides to explicate that new employees must receive keys to enter the building. The company thus changes the workflow `hire.v1` by adding the action labeled `Assign Keys` as depicted in the bottom part of Figure 1.

A manager wants to understand how the syntactic changes impact the AD's possible execution traces. Using our method for semantic differencing reveals that there are possible execution traces of `hire.v1` that are not possible in `hire.v2` and vice versa. Thus, execution traces have been removed and new execution traces have been added. The manager gets presented that `Register`, `Get Wel. Package`, `Assign to Project`, `Add to Website`, `Manager Interview`, `Manager Report`, `Authorize Payment` is a possible execution trace of `hire.v1` that is no execution trace of `hire.v2`. This execution trace has been removed during the evolution of the workflow. Vice versa, the manager is presented an execution trace including the action `Assign Keys`, which is possible in `hire.v2` and not possible in `hire.v1`. This execution trace has been added during the evolution from `hire.v1` to `hire.v2`.

2.2 Example 2

Figure 2 depicts two ADs previously presented in [10]. The ADs model workflows to be performed in an insurance company in response to an incoming claim. The AD `wf1` models the company's original workflow. After some time, the company decides to modify the workflow to `wf2`. A manager is interested in the semantic difference between the new and the original workflow. She thus uses our framework and gets presented that `Record Claim`, `Check Claim`, `Reject Claim`, `Send Declinature` is a possible execution trace of the new workflow that is not possible in the original workflow. Vice versa, the manager is interested if execution traces have been removed during the evolution from `wf1` to `wf2`. She thus uses our framework again and gets presented that `Record Claim`, `Check`

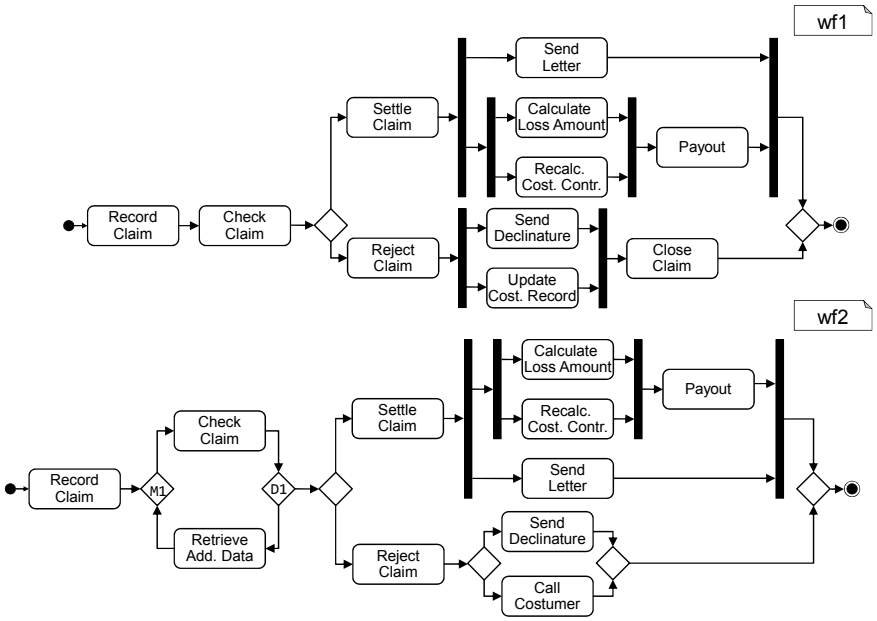

Fig. 2. Two activity diagrams adapted from [10].

`Claim`, `Reject Claim`, `Send Declinature`, `Update Cost. Record`, `Close Claim` is a possible execution trace of `wf1` that is not possible in `wf2`.

3 Preliminaries

This section introduces the notations for ε-NFAs (e.g. [8]) as used in this paper. The empty word is denoted by ε. An alphabet is a non-empty finite set A that does not contain the empty word $\varepsilon \notin A$. We denote by A^* the set of all finite sequences (words) over an alphabet A, which contains the empty word $\varepsilon \in A^*$.

Definition 1. *A non-deterministic finite automaton with epsilon moves (ε-NFA) is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where*

- *Q is a finite set of states,*
- *Σ is an alphabet,*
- *$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is a transition relation,*
- *$q_0 \in Q$ is an initial state, and*
- *$F \subseteq Q$ is a set of final states.*

Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. For all $q \in Q$, let $E_A(q) \subseteq Q$ denote the epsilon closure of q in A, *i.e.,* the set of states reachable from q in A by following transitions from δ labeled with ε. It is defined as the smallest set satisfying $q \in E_A(q)$ and $\forall p \in E_A(q) : \forall (s, a, t) \in \delta : (s = p \wedge a = \varepsilon) \Rightarrow t \in E_A(q)$.

Let $n \in \mathbb{N}$. The automaton A accepts a word $w = w_1, w_2, ...w_n \in \Sigma^*$ iff there exists a finite sequence of states $r = r_0, r_1, ..., r_n \in Q^*$ satisfying

1. $r_0 \in E_A(q_0)$,
2. $r_{i+1} \in E_A(r')$ where $r' \in Q$ such that $\delta(r_i, a_{i+1}, r') \in \delta$ for all $0 \leq i < n$,
3. $r_n \in F$.

The language recognized by A is defined as $\mathcal{L}(A) = \{w \in \Sigma^* \mid A \text{ accepts } w\}$. There exist well-known algorithms for checking if the language recognized by an ε-NFA is included in the language recognized by another ε-NFA [8]. We reduce semantic differencing of ADs to language inclusion checking between ε-NFAs.

4 Activity Diagrams

This section defines the syntax and semantics of ADs as used in this paper. To this effect, we define the abstract syntax of ADs and a translation from ADs to ε-NFAs. The semantics of an AD is then defined as the language recognized by the ε-NFA obtained from translating the AD.

4.1 Activity Diagram Syntax

The abstract syntax of ADs is defined as follows.

Definition 2. *An AD is a tuple* $(L, N, A, i, F, XOR, AND, T, l)$ *where*

- L *is an alphabet containing action labels,*
- N *is a finite set of nodes,*
- $A \subseteq N$ *is a set of action nodes,*
- $i \in N$ *is the initial node,*
- $F \subseteq N$ *is a non-empty set of final nodes,*
- $XOR \subseteq N$ *is a set of decision and merge nodes,*
- $AND \subseteq N$ *is a set of fork and join nodes,*
- $T \subseteq N \times N$ *is the transition relation,*
- $l : N \to L \cup \{\varepsilon\}$ *is the node labeling function that is required to map each control-flow node to the empty word, i.e.,* $\forall n \in N \setminus A : l(n) = \varepsilon$,
- $\{A, \{i\}, F, XOR, AND\}$ *is a partition of* N.

For an AD $ad = (L, N, A, i, F, XOR, AND, T, l)$ and all nodes $n \in N$, we denote by $\delta^+_{ad}(n) \stackrel{\text{def}}{=} \{(n, x) \subseteq T \mid x \in N\}$ the set of outgoing transitions starting in n. Similarly, $\delta^-_{ad}(n)$ denotes the set of incoming transitions ending in n.

The AD `hire.v1` (cf. Figure 1), for instance, can be formally defined by $\texttt{hire.v1} = (L, N, A, i, F, XOR, AND, T, l)$ with

- labels $L = \{$`Register`, `Assign to Project`, `Get Wel. Package`, `Add to Website`, `Manager Interview`, `Manager Report`, `Authorize Payment`$\}$,
- nodes $N = \{R, ATP1, GWP, ATP2, ATW, MI, MR, AP, D1, M1, F1, J1, i, f\}$,
- action nodes $A = \{R, ATP1, GWP, ATP2, ATW, MI, MR, AP\}$,

- initial node i,
- final nodes $F = \{f\}$,
- decision and merge nodes $XOR = \{D1, M1\}$,
- fork and join nodes $AND = \{F1, J1\}$,
- the transition relation $T = \{(i, R),\ (R, D1),\ (D1, ATP1),\ (ATP1, M1),\ (D1, GWP), (GWP, F1), (F1, ATP2), (F1, ATW), (ATP2, J1), (ATW, J1),\ (J1, MI),\ (MI, MR),\ (MR, M1),\ (M1, AP),\ (AP, f)\}$, and
- labeling function l with $l(R) =$ `Register`, $l(ATP1) =$ `Assign to Project`, $l(GWP) =$ `Get Wel. Package`, $l(ATP2) =$ `Assign to Project`, $l(ATW) =$ `Add to Website`, $l(MI) =$ `Manager Interview`, $l(MR) =$ `Manager Report`, $l(AP) =$ `Authorize Payment`, and $l(D1) = l(M1) = l(F1) = l(J1) = l(i) = l(f) = \varepsilon$.

Usually, the following (or even stronger) well-formedness rules apply (e.g. [1]):

Definition 3. *An AD is called well-formed if, and only if, the following conditions are satisfied:*

- every action node has exactly one incoming and one outgoing transition,
- every XOR-node (decision or merge) and every AND-node (fork or join) has at most one incoming transition or at most one outgoing transition,
- the initial node has no incoming and exactly one outgoing transition,
- every final node has no outgoing and exactly one incoming transition.

4.2 Activity Diagram Semantics

This section explicitly defines a mapping from ADs to ε-NFAs. The ε-NFA resulting from translating an AD encodes exactly the traces modeled by the AD under the assumption that exactly one action can be performed at a point in time. Stated differently, no two action can be performed simultaneously. The semantics of an AD is then defined as the language recognized by the ε-NFA that it is mapped to. ADs contain action labels whereas ε-NFAs contain transition labels. The mapping therefore inverts nodes and transitions of an AD during the translation, *i.e.,* the nodes of the AD correspond to transitions in the ε-NFA and the transitions in the AD correspond to the ε-NFA's states. The state set of the automaton is given by the powerset of the set of transitions of the AD. Using the powerset is necessary as an AD can reside in several nodes simultaneously (after visiting a fork node). The set of transition labels in the ε-NFA is equal to the set of node labels in the AD. The initial state of the automaton is the singleton set containing the transition that has the initial AD node as source node. The final states are exactly the singleton sets containing a transition that has a final AD node as target node. The transition relation is defined as the smallest set of transitions satisfying two conditions. The first condition encodes the AD's behavior in case it executes a node that is neither a join nor a fork node. If an action node is executed, then the AD moves out of the executed action via the action's outgoing transition to the next node and outputs the action node's label. The AD's execution state regarding other nodes remains unchanged. Similarly,

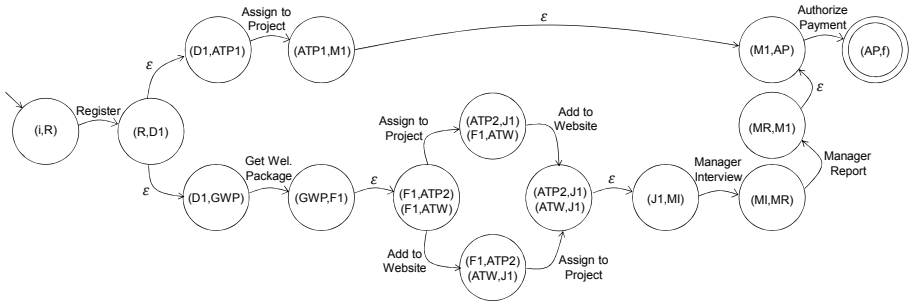

Fig. 3. Reachable part of the ε-NFA obtained from translating `hire.v1` (cf. Figure 1).

if the AD proceeds through a decision or merge node, its state regarding the other nodes remains unchanged and it outputs nothing (the empty word ε). The situation is different for fork and join nodes: An AD can only proceed through a join or fork node if its state contains all nodes that have a transition to the node. Further, if the AD proceeds through a fork or join node, it leaves all states that precede the node and enters all states that the node leads to.

Definition 4. *Let $ad = (L, N, A, i, F, XOR, AND, T, l)$ be a well-formed AD. The ε-NFA associated to ad is defined as $nfa(ad) = (Q, \Sigma, \delta, q_0, F')$ where*

- $Q = \mathbb{P}(T)$,
- $\Sigma = L$,
- $q_0 = \{(s,t) \in T \mid s = i\}$,
- $F' = \{\{(s,t)\} \subseteq T \mid t \in F\}$, *and*
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ *is the smallest set satisfying the following conditions:*
  - *Move other than fork or join:*
    $\forall X \subseteq T : \forall n_1, n_2, n_3 \in N : ((n_1, n_2) \in X \wedge (n_2, n_3) \in T \wedge n_2 \notin AND) \Rightarrow (X, l(n_2), (X \setminus \{(n_1, n_2)\}) \cup \{(n_2, n_3)\}) \in \delta$
  - *Move fork or join:*
    $\forall X \subseteq T : \forall j \in AND : (\delta^-_{ad}(j) \subseteq X) \Rightarrow (X, \varepsilon, (X \setminus \delta^-_{ad}(j)) \cup \delta^+_{ad}(j)) \in \delta$

The trace semantics $sem(ad)$ of an AD ad is defined as the language recognized by the ε-NFA associated to the AD ad, *i.e.,* $sem(ad) \stackrel{\text{def}}{=} \mathcal{L}(nfa(ad))$.

As an example, Figure 3 depicts the ε-NFA obtained from translating the AD `hire.v1` (graphically illustrated in Figure 1 and formally defined in subsection 4.1) after removing the states and transitions that are not reachable from the initial state. A possible execution trace of the AD is, for example, given by `Register`, `Assign to Project`, `Authorize Payment` $\in sem($`hire.v1`$)$.

4.3 Semantic Differencing of Activity Diagrams

The (asymmetric) semantic difference $\delta(ad_1, ad_2)$ between two ADs ad_1, ad_2 is defined as the set of traces possible in the AD ad_1 that are not possible in

AD	#reach. states	#reach. transitions
`hire.v1`	15	26
`hire.v2`	19	36
`wf1`	26	57
`wf2`	27	76

	time
$\delta(\texttt{hire.v1}, \texttt{hire.v2})$	89ms
$\delta(\texttt{hire.v2}, \texttt{hire.v1})$	138ms
$\delta(\texttt{wf1}, \texttt{wf2})$	116ms
$\delta(\texttt{wf2}, \texttt{wf1})$	173ms

Fig. 4. Number of reachable states and transitions in the NFAs resulting from translating the ADs as well as computation times for semantic differencing of the ADs.

the AD ad_2, *i.e.,* $\delta(ad_1, ad_2) = sem(ad_1) \setminus sem(ad_2)$. With the explicit mapping from ADs to ε-NFAs, reuse of well-known constructions from automata theory [8] is possible. It holds that $\delta(ad_1, ad_2) = \emptyset$ iff $sem(ad_1) \subseteq sem(ad_2)$, which is again equivalent to $\mathcal{L}(nfa(ad_1)) \subseteq \mathcal{L}(nfa(ad_2))$. We can thus reuse well-known techniques for language inclusion checking and counterexample generation for ε-NFAs, which are two well-studied decidable problems. For example, semantic differencing of `hire.v1` and `hire.v2` yields that `Register`, `Get Wel.`, `Package`, `Assign to Project`, `Add to Website`, `Manager Interview`, `Manager Report`, `Authorize Payment` $\in \delta(\texttt{hire.v1}, \texttt{hire.v2})$ is an execution trace of the AD `hire.v1` that is no execution trace of the AD `hire.v2`.

5 Implementation and Usage

We have implemented a prototype for the semantic differencing operator using the language workbench MontiCore [7] and the finite automata language inclusion checking tool RABBIT [15]. We developed a MontiCore modeling language for ADs, and a translation to finite automata in the BA format, which is the input format of RABIT. The tool as well as this paper's examples are available online [16]. The left table in Figure 4 summarizes the sizes of the NFAs resulting from translating the ADs presented in Section 2 after eliminating epsilon-transitions as well as removing unreachable states and transitions. The right table summarizes the computation times reported by RABIT after applying the semantic differencing operator. All checks were executed using RABIT 2.4 on a computer with a 3.0 GHz Intel Core i7 CPU, 16 GB Ram, and Windows 10. The NFA obtained from translating the AD `wf2`, for example, contains 27 states and 76 transitions that are reachable from the initial state (cf. left table in Figure 4). Checking whether $\delta(\texttt{wf2}, \texttt{wf1}) = \emptyset$ and computing a witness $w \in \delta(\texttt{wf2}, \texttt{wf1})$ took 173ms (cf. right table in Figure 4). We conclude that this paper's translation provides promising results. However, the tool was only applied to a small set of ADs. Therefore, the results are not generalizable: The tool's execution time may strongly differ when increasing the size of the input ADs.

6 Related Work

Semantic differencing of ADs is introduced in [11]. In this approach, the input ADs are translated to models in the input language of the SMV model

checker [14,17] using the translation described in [12]. These models then encode finite automata that correspond to the ADs. The translation from ADs to SVM models is rather complicated. Further, the overhead produced by the translation steps from ADs to SVM models and from SMV models to finite automata causes additional overhead in contrast to directly translating ADs to finite automata. The overhead adds unnecessary complexity to the translation and thus makes the translation hard to understand. In contrast, our translation from ADs to finite automata is direct and simple and therefore easy to understand. The framework presented in [11] has been extended in [13] for summarizing elements in the semantic difference between two ADs based on equivalence classes defined on the set of possible traces. The idea is to present only one representative of an equivalence class to a user. The summarization technique is easily integrable into this paper's framework. The framework presented in [10] can be used to detect which syntactic changes between two different versions of an AD induce a concrete witness. The application to ADs as presented in [10] is based on the semantic differencing operator of [11]. Using the techniques of [10] with the semantic differencing method presented in this paper is directly possible. The translation from ADs to SMV models of [11] is inspired by a similar translation defined in [3]. The translation defined in [3] is also more complicated than our direct translation to finite automata. Further, the scope of [3] is symbolic model checking of ADs, whereas this paper focuses on semantic differencing.

Other direct translations from ADs and business process models to finite automata focus on deadlock detection [19,20]. However, the translations do not result in automata that represent the set of execution traces of the input ADs. Further, the translations are more complex than our translation. With minor adjustments, this paper's translation can also be used for deadlock detection by changing the accepting states of an automaton resulting from translating an AD.

Other semantics definitions for ADs are based on Petri nets [18], on the system model [4] for characterizing object oriented systems as defined in [2], or on the notion of *step* [9] inspired by the popular STATEMATE semantics for statecharts [6]. In contrast, this paper defines an operational semantics based on a mapping to ε-NFAs and a denotational semantics based on the language recognized by the resulting ε-NFAs.

7 Conclusion

We formally defined an abstract syntax of an AD variant. Based on this, we presented an operational semantics for ADs via a translation from ADs to ε-NFAs and a denotational semantics given by the language recognized by these ε-NFAs. This enabled to reuse well-known algorithms for language inclusion checking between ε-NFAs for semantic differencing of ADs. We presented a tool prototype and applied the semantic differencing operator to four example ADs, which showed that the differencing operator yields promising results. The translation enables a comprehensive and easy to implement method for semantic differencing of ADs, which ultimately facilitates semantic AD evolution analysis.

References

1. van der Aalst, W.M.P.: Formalization and Verification of Event-driven Process Chains. Information & Software Technology **41**(10) (1999)
2. Broy, M., Cengarle, M.V., Grönniger, H., Rumpe, B.: Definition of the UML System Model. In: UML 2 Semantics and Applications. John Wiley & Sons (2009)
3. Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. ACM Trans. Softw. Eng. Methodol. **15**(1) (2006)
4. Grönniger, H., Reiß, D., Rumpe, B.: Towards a Semantics of Activity Diagrams with Semantic Variation Points. In: Conference on Model Driven Engineering Languages and Systems (MODELS'10) (2010)
5. Object Management Group: OMG Unified Modeling Language (OMG UML) (May 2017), `https://www.omg.org/spec/UML/About-UML/` [accessed 2018-06-19]
6. Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts. ACM Trans. Softw. Eng. Methodol. **5**(4) (1996)
7. Hölldobler, K., Rumpe, B.: MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Shaker Verlag (December 2017)
8. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
9. Knieke, C., Goltz, U.: An Executable Semantics for UML 2 Activity Diagrams. In: Proceedings of the International Workshop on Formalization of Modeling Languages (FML'10) (2010)
10. Maoz, S., Ringert, J.O.: A framework for relating syntactic and semantic model differences. Software & Systems Modeling **17** (July 2018)
11. Maoz, S., Ringert, J.O., Rumpe, B.: ADDiff: Semantic Differencing for Activity Diagrams. In: Conference on Foundations of Software Engineering (2011)
12. Maoz, S., Ringert, J.O., Rumpe, B.: An Operational Semantics for Activity Diagrams using SMV. Tech. Rep. AIB-2011-07, RWTH Aachen University (2011)
13. Maoz, S., Ringert, J.O., Rumpe, B.: Summarizing Semantic Model Differences. In: Models and Evolution Workshop (ME'17) at MODELS (2011)
14. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
15. RABIT Tool Homepage (2018), `http://http://www.languageinclusion.org/` [accessed 2018-06-20]
16. Activity Diagram Semantic Differencing Tool (2018), `http://www.se-rwth.de/materials/semdiff/` [accessed 2018-06-26]
17. SMV Model Checker Homepage (2018), `https://www.cs.cmu.edu/~modelcheck/smv.html` [accessed 2018-06-20]
18. Störrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. Electronic Notes in Theoretical Computer Science **127**(4) (2005)
19. Sugunnasil, P.: Detecting deadlock in activity diagram using process automata. In: International Computer Science and Engineering Conference (ICSEC'16) (2016)
20. Tantitharanukul, N., Sugunnasil, P., Jumpamule, W.: Detecting Deadlock and Multiple Termination in BPMN Model Using Process Automata. In: International Confernce on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI'10) (2010)