A Graph-Based Approach for Containment Checking of Behavior Models of Software Systems

Huy Tran **Faiz UL Muram** **Uwe Zdun**

Research Group Software Architecture
University of Vienna, Austria.

huy.tran|faiz.ulmuram|uwe.zdun@univie.ac.at

Abstract. In the development of complex and large scale software systems, it is important to detect and fix the deviations of systems' behaviors at different abstraction levels in early phases. Our main focus here is the *containment checking*—a special type of consistency checking—that verifies whether the behavior (or functions) described by a low-level behavior model encompasses those specified in the high-level counterpart. As shown in our previous work, containment checking can be realized based on model checking, but not always the costly exhaustive searches employed by model checking are necessary for addressing the containment checking problem, leading to potentials for optimization. In addition, model checking and similar techniques often yield the checking results as *true* (satisfied) or *false* (unsatisfied) with error traces (e.g., counter-examples). Unfortunately, such feedback is rather not helpful for users with limited knowledge of the underlying formal methods to analyze and understand the causes of consistency violations. In this paper, we propose a lightweight graph-based approach for addressing the aforementioned problems of containment checking. The theoretical complexity of our approach is a cubic polynomial of the number of elements of the input behavior models. Additionally, we aim at generating feedbacks that are relevant and easy-to-understand for the stakeholders. Our approach is illustrated and evaluated on UML activity diagrams—that are widely used for modeling behaviors of software systems—using use cases derived from industrial scenarios.

Keywords: Containment checking, consistency checking, graph, behavior model, UML activity diagram.

1 Introduction

Model-based software development is maturing and potentials for solving problems with large and complex system development[3,12,32]. Thus, software engineers are increasingly using models for several development tasks such as describing and analyzing software systems or generating system implementations out of these models. A typical development scenario based on models is, especially in the domain of enterprise systems, that a business analyst or software architect uses a high-level model for outlining the system and discussing with the customers and developers. The high-level model will then be refined to one or more low-level models by the development team. In the course of software system modeling and implementation, as models are created and evolved independently by different stakeholders and teams, inconsistencies among models often occur. Hence, detecting model inconsistencies in early phases of the software development life cycle is crucial to eliminate as many anomalies as possible before the systems are actually implemented and deployed. This has led to a rich body of work for checking and managing model consistency in the literature [18]. Of these existing approaches, only a few are aiming at supporting consistency checking of behavioral models for software systems [18], for instance, checking behavioral models against non-behavioral models [15,29,36] or checking different types of behavior models [9,16,17,43,48]. Nonetheless, there are very few studies on checking the deviation of software behavioral models at different abstraction levels.

In this work, we focus on the containment relationship which is a special type of consistency relationship between software models at different levels of abstraction. The containment relationship is categorized as *vertical consistency* [39]. An unsatisfied containment relationship implies the deviation of the low-level descriptions from the corresponding high-level specifications and properties. To the best of our knowledge, very few existing studies have addressed the containment relationship so far.

Containment checking for software behavior models, from a broader point of view, is related to the notion of behavioral equivalence relations between state transition systems [19,27]. A notable challenge of behavioral equivalence checking (that can be deduced to containment checking) is that the estimated theoretical computational complexity is **NP**-hard, even for a class of simple finite communicating elements [28]. Thus, it is rather costly to apply behavioral equivalence checking to complex and large scale software systems. Moreover, behavioral equivalence checking is strict in requiring a bidirectional equivalence of two behavior models whilst the containment relationship mainly aims at unidirectional consistency.

There are some existing approaches trying to alleviate the complexity of equivalence checking by aiming at the *similarity* of behavior models in particular application domains, for instance, workflows and business processes [13,30], state-charts [22], state-based models [42], to name but a few. Nevertheless, the outcome of these techniques is not a precise answer whether two behavior models are equivalent or subsumed but rather an estimated degree of similarity of the input models. Hence, these approaches are useful for finding *similar* behavioral descriptions but not quite applicable for verifying the containment relationship.

Another challenge that has not been adequately addressed by the existing approaches for behavioral equivalence checking, and also consistency checking, is to assist the stakeholders in understanding the outcomes of the checking process. For instance, existing approaches for checking behavioral equivalence (aka *bisimulation*) often return a binary *true* (satisfied) or *false* (unsatisfied) answer without concrete information of the inequivalent cases [11,19,40]. Another example are verification techniques based on model checking [5,20] that produce counterexamples which are complex state based error traces [4,14]. Those are, however, rather cryptic for users who often have limited knowledge of the underlying formal methods [8].

We present in this paper a lightweight approach for addressing these challenges of containment checking. First, the input behavior models will be mapped onto intermediate representations, namely, *check models*. Based on a formal definition of the containment problem, our approach can be used to verify whether the resulting check models satisfy the containment relationship. Our proposed graph-based containment checking technique performs reasonably within the boundary of $O(n^3)$, where $n = max(n_1, n_2)$

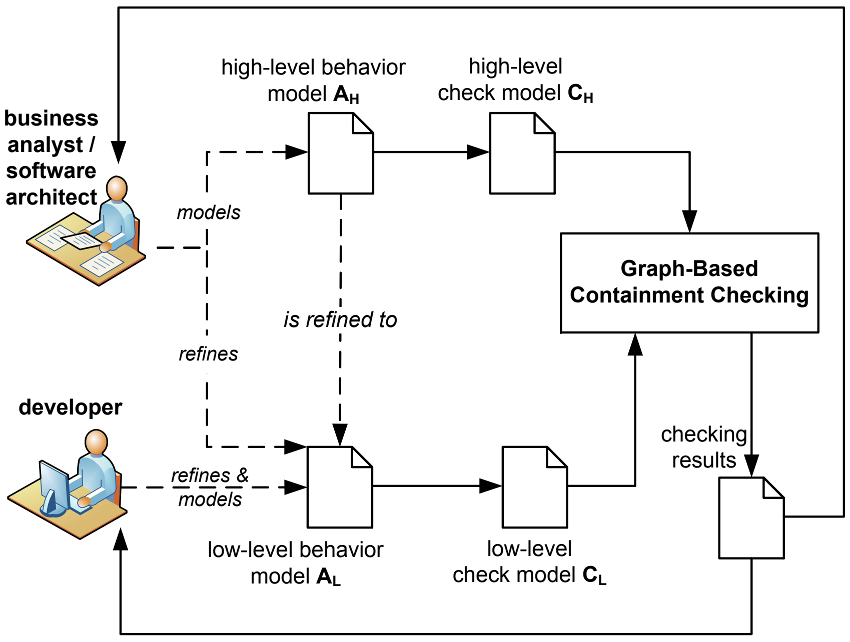
high-level behavior model $\mathbf{A_H}$
high-level check model $\mathbf{C_H}$
business analyst / software architect
models
is refined to
refines
Graph-Based Containment Checking
developer
refines & models
checking results
low-level behavior model $\mathbf{A_L}$
low-level check model $\mathbf{C_L}$

Figure 1. Overview of the graph-based containment checking approach

and n_1, n_2 are the numbers of elements of the two input models, respectively. Furthermore, our approach aims at producing concrete and helpful information about the inconsistencies, such as missing elements, missing execution paths, or missing loops, in case the containment relationship is not satisfied. We illustrate our approach using UML activity diagrams—a part of the Unified Modeling Language[26]—which are widely used in both academia and industry for modeling and analyzing behaviors of software systems. Nevertheless, our approach can also be applied to other types of behavior models such as BPMN[25] that share similar notions and structures with UML activity diagrams.

The paper is structured as follows. We describe our graph-based approach for containment checking in detail in Section 2. In Section 3 we use a realistic example extracted from industrial case studies to illustrate our approach along with its performance evaluation. Section 4 is dedicated for discussing the related studies on supporting behavioral consistency checking and especially containment checking. We summarize the main contributions and discuss potential future work in Section 5.

2 Approach

In this section, we describe our graph-based approach for containment checking of UML activity diagrams. An overview of the approach is shown in Figure 1. The main focus of the approach is depicted by the solid lines whilst the dashed lines illustrate relevant modeling and developing activities of the involved stakeholders.

Our approach starts by mapping the input UML activity diagrams at different levels of abstraction into equivalent intermediate representations, namely, *check models*. Then, our graph-based containment checking algorithm is used for verifying whether the resulting check models satisfy the containment relationship. In case the containment relationship is not satisfied (i.e., the input UML activity diagrams are inconsistent), our approach will be able to produce relevant checking results with concrete information about the causes of inconsistencies such as missing elements, missing execution paths, or missing cycles as well as the involved model elements.

2.1 Activity Models and Check Models

As the definitions and semantics of UML activity diagrams are rather informal[26] Sec. 12, we derive a representative description, namely, activity model, to provide the basis for formally analyzing UML activity diagrams. The definition of an activity model is based on the definition of classical transition systems[19]. An activity model needs to adequately accommodate relevant concepts of a UML activity diagram such as different kinds of nodes, edges, and guards.

Definition 1 (Activity model)**.** *An activity model $\mathcal{A}$ is a tuple $(N, E, G, type, guard)$ where*

- *N is a finite set of nodes*
- *$E \subseteq N \times N$ is an ordered finite set of edges,*
- *G is a finite set of guard expressions,*
- *$type : N \rightarrow${*InitialNode, ActivityFinalNode, FlowFinalNode, Action, DecisionNode, MergeNode, ForkNode, JoinNode, LoopNode, ConditionalNode*} is a function that maps a node to its type. Node types are derived from the UML 2 specification[26] Sec. 12.*
- *$guard : E \rightarrow G$ is a function that maps an edge to its guard expressions*[1].

In Figure 2 we depict a simplified activity diagram $\mathcal{A} = (N, E, G, type, guard)$ of a travel agency system where N={Start, ReceiveItinerary, Fork, M1, M2, M3, BookCar, BookHotel, BookFlight, D1, D2, D3, Join, ChargeCreditCard, NotifyCustomer, Finish}, *guard*(e12) = "c_booked=no", *guard*(e15) = "c_booked=yes", *guard*(e13) = "h_booked=no", *guard*(e16) = "h_booked=yes", *guard*(e14) = "f_booked=no", *guard*(e17) = "f_booked=yes", *type*(Start) = *InitialNode*, *type*(M1) = *type*(M2) = *type*(M3) = *MergeNode*, *type*(ReceiveItinerary) = *type*(BookHotel) = *type*(BookFlight) = *type*(ChargeCreditCard) = *type*(NotifyCustomer) = *Action*, *type*(Fork) = *ForkNode*, *type*(Join) = *JoinNode*, *type*(D1) = *type*(D2) = *type*(D3) = *DecisionNode*, *type*(Finish) = *ActivityFinalNode*. For the sake of illustration, we explicitly name the control nodes such as "*Start*", "*Fork*", "*Join*", and "*Finish*". The edges are prefixed with "e". In reality, the developers can often accept the default identifiers generated automatically by UML modeling tools.

The main goal of our approach is to develop a lightweight graph-based algorithm for supporting the developers in checking the containment relationship between a high-level and low-level activity model. That is, our algorithm needs to verify whether the behavior (or functions) described by the low-level activity model encompasses those specified in the high-level counterpart. Thus, the containment checking algorithm will walk through the high-level activity model to check whether its elements and structures (e.g., actions, control nodes, and edges) have corresponding parts in the low-level model.

However, the current form of an activity model might render the graph-based searching inefficient because it might also contain—apart from the essential elements such as nodes (including both actions

[1]We note that guard expressions are sub-classes of *ValueSpecification* in UML. For instance, a guard expression could be an *OpaqueExpression* such as *"$x \leq 1$"* or a *LiteralString* such as *"credit card accepted"*. Here we omit the detailed formal definition of each possible kind of *ValueSpecification* as only their presence and comparability for equality is relevant for our containment checking approach, and hence, a more detailed definition is not needed.

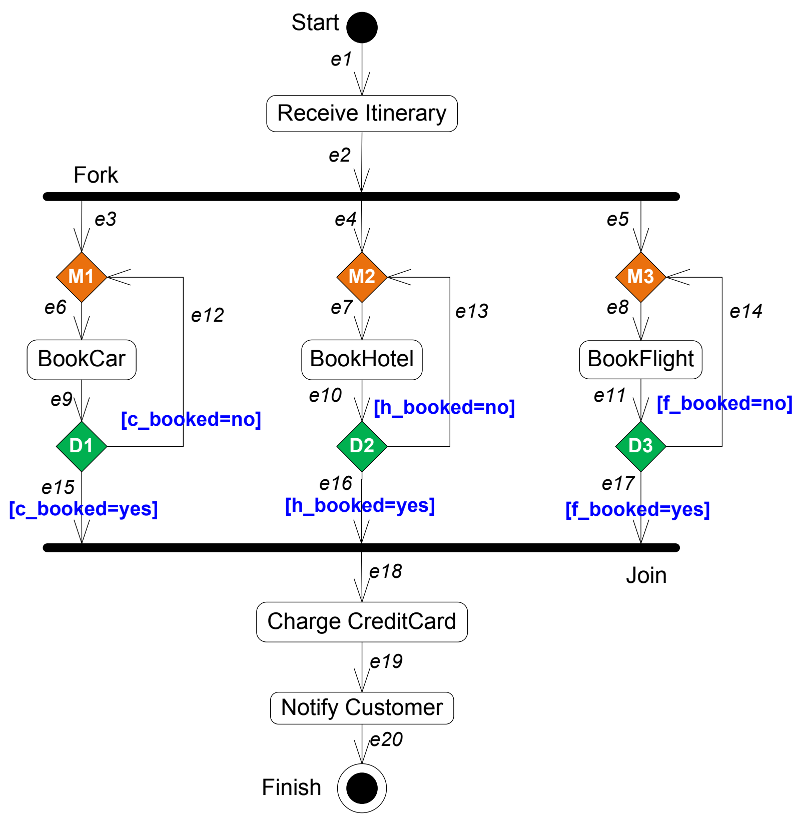

Figure 2. An illustrative simplified activity model of a travel agency

and control nodes) and control flows that are similar to the nodes and edges of a graph—some unparalleled kinds of elements. For instance, an activity model might have edges associated with guards. A guard will determine whether the corresponding edge can be activated (i.e., leading to the execution of the edge's target) following the execution of the edge's source. This implies that the algorithm must dedicate special treatments for them as for the other control nodes because the guards have significant impact on the behavior of the software systems described in the models.

Unfortunately, most of the existing approaches do not take guards into account adequately but rather assume that the control flows will be automatically activated. To alleviate this problem, we introduce an intermediate representation, namely, a check model, that can explicitly represent such kinds of elements. The essential idea is to transform the model elements associated with guards (e.g., edges) into pseudo nodes to yield an intermediate representation that poses the same semantics as the original model and is efficient for graph searching. We show here how the edges are transformed to pseudo nodes of a check model. Similar model elements can be treated in the same manner.

Definition 2 (Check model)**.** *We assume that $\mathcal{A} = (N_\mathcal{A}, E_\mathcal{A}, G_\mathcal{A}, type_\mathcal{A}, guard_\mathcal{A})$ is an activity model. A check model $\mathcal{C}$ derived from $\mathcal{A}$ is represented by a tuple ($N_\mathcal{C}$, $E_\mathcal{C}$, $G_\mathcal{C}$, $type_\mathcal{C}$, $guard_\mathcal{C}$) where $N_\mathcal{C} = N_\mathcal{A} \cup \{n \mid type_\mathcal{C}(n) = GuardNode\}$, $E_\mathcal{C} = E_\mathcal{A} \cup E^g$, $G_\mathcal{C} = G_\mathcal{A}$, $type_\mathcal{C} = type_\mathcal{A} \cup \{GuardNode\}$, and $guard_\mathcal{C} = N_\mathcal{C} \cup E_\mathcal{C} \to G_\mathcal{C}$.*

We use $e(s,t)$ to denote an edge $e \in E_\mathcal{A}$ that connects a source node s to a target node t where $s, t \in N_\mathcal{A}$. The construction of $N_\mathcal{C}$ and $E_\mathcal{C}$ is defined as follows. For every edge $e \in E_\mathcal{A}$ labeled with valid a guard constraint (i.e., $guard(e) \neq null$), we create a new node n_g, assign the guard value of e to the node, establish additional unlabeled links between n_g and the source and target nodes of e, and remove e from $\mathcal{C}$. Formally, the translation can be described as in Equation 1.

$$\begin{aligned}
&\forall e(s,t) \in E_{\mathcal{A}} \text{ such that } guard_{\mathcal{A}}(e) \neq null \bullet \\
&\quad \big(N_{\mathcal{C}} = N_{\mathcal{A}} \cup \{n_g\}\big) \wedge \\
&\quad \big(E_{\mathcal{C}} = (E_{\mathcal{A}} \cup \{(s,n_g),(n_g,t)\}\} \setminus \{(s,t)\}\big) \wedge \\
&\quad \big(guard_{\mathcal{C}}(n_g) \equiv guard_{\mathcal{A}}(e)\big)
\end{aligned} \tag{1}$$

We note that the containment checking will be performed on the check model. Thus, it is necessary to prove that an activity model and the check model derived from it are behaviorally equivalent. We present a simple sketch of a proof as following.

Proof Sketch. Let us assume that an activity model $\mathcal{A} = (N_{\mathcal{A}}, E_{\mathcal{A}}, G\mathcal{A}, type_{\mathcal{A}}, guard_{\mathcal{A}})$ is defined on a semantic domain $\mathbb{D}_A$. Based on $\mathbb{D}_A$ the semantics of a guarded edge $e \in E_{\mathcal{A}}$ is denoted as $[\![e]\!]$. A check model $\mathcal{C} = (N_{\mathcal{C}}, E_{\mathcal{C}}, G\mathcal{C}, type_{\mathcal{C}}, guard_{\mathcal{C}})$ is mapped from $\mathcal{A}$ using the procedure presented in Equation 1. We can derive a semantic domain $\mathbb{D}_C$ for the check model $\mathcal{C}$ based on $\mathbb{D}_A$ such that each newly added guarded node $n \in \mathcal{C}$ corresponding to an edge $e \in E_{\mathcal{A}}$ will have the same semantics of e. That is $[\![n]\!] \equiv [\![e]\!]$. Hence, it is rather straightforward that $\mathcal{C}$ and $\mathcal{A}$ behave in the same way with respect to these semantic domains. □

Here we give a simple example to illustrate the aforementioned proof. In the semi-formal token-based semantic domain $\mathbb{D}_A$ of the UML 2 specification[26] pg. 371, a token is allowed to pass along an edge e if and only if $guard_{\mathcal{A}}(e)$ evaluates to *true*. Hence, $\mathbb{D}_C$ can be defined based on $\mathbb{D}_A$ such that a guarded node n mapped from e will pass along the token to the next node(s) if and only if $guard_{\mathcal{C}}(n) \equiv guard_{\mathcal{A}}(e)$ holds.

In Figure 3 we depict the check model (with omitted edges' labels) derived from the activity model shown in Figure 2. The two models are rather alike. The major difference is that all guarded edges of the activity model are transformed into pseudo nodes (shown with dashed border lines) associated with corresponding guard expressions in the check model. In this way, a graph search algorithm can treat all edges in the same manner.

2.2 Graph-Based Containment Checking

Our approach takes as inputs a high-level activity model $\mathcal{A}_H$ and a low-level activity model $\mathcal{A}_L$ to verify whether the containment relationship between these models are satisfied (i.e., $\mathcal{A}_L$ "*contains*" $\mathcal{A}_H$). In case of inconsistencies, we need to inform the developers the particular causes of the violation. We use the relation symbol $\prec$ to denote the containment relationship between these behavior models. The containment relationship to be validated by our graph-based algorithm is defined in Equation 2.

$$\begin{aligned}
\mathcal{A}_H \prec \mathcal{A}_L \overset{\text{def}}{=\!=} \;& \text{noMissingNodes}(\mathcal{C}_H, \mathcal{C}_L) \\
& \wedge \text{noMissingTransitiveLinks}(\mathcal{C}_H, \mathcal{C}_L) \\
& \wedge \text{noMissingCycles}(\mathcal{C}_H, \mathcal{C}_L)
\end{aligned} \tag{2}$$

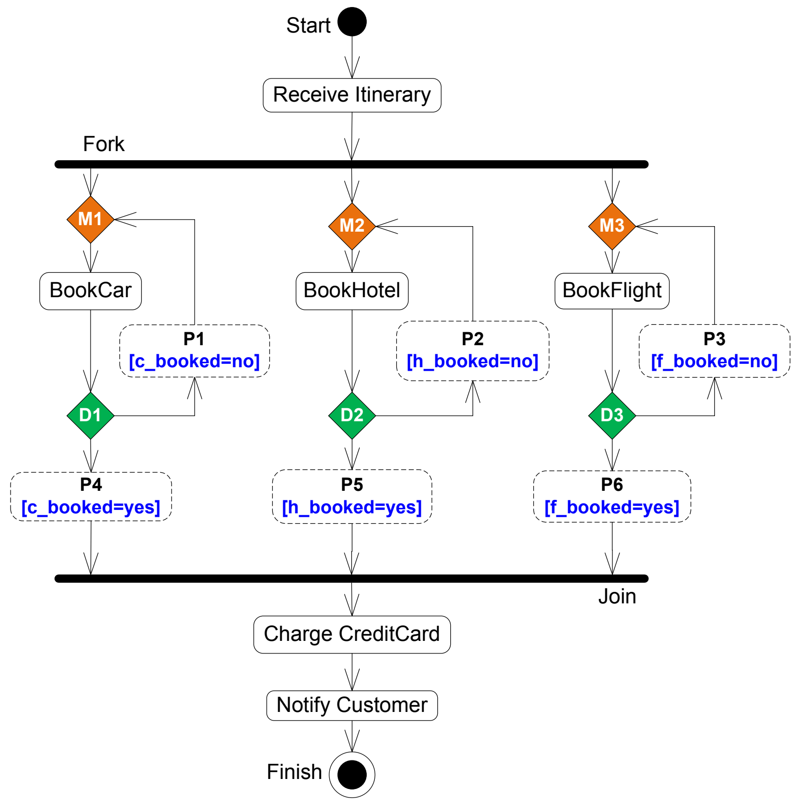

Figure 3. The corresponding check model of the travel agency activity model

where $\mathcal{C}_H$ (resp. $\mathcal{C}_L$) is mapped from $\mathcal{A}_H$ (resp. $\mathcal{A}_L$).

Next, we will explain in detail three functions *noMissingNodes()*, *noMissingTransitiveLinks()*, and *noMissingCycles()* presented in Equation 2. We note that these functions represent the (sub)tasks of containment checking. That is, our graph-based containment checking algorithm will verify whether there are any missing nodes (i.e., missing expected functions), missing transitive links (i.e., missing execution paths), and missing cycles (i.e., missing loop executions). In other words, the containment checking problem is divided into three sub-problems represented by the corresponding functions shown in Equaation 2.

The huge advantage of this *divide-and-conquer* strategy is that our approach is able to tell specifically about the violation of the containment relationship based on the results achieved from each function. For instance, we can inform the violation of the containment relationshp due to missing nodes, execution paths, or loops along with the involved elements by analyzing the relevant formulas. Moreover, we note that these functions can be performed independently, and therefore, can potentially be parallelized to gain better performance.

$$\begin{aligned}\text{noMissingNodes}(\mathcal{C}_H, \mathcal{C}_L) \overset{\text{def}}{=\!=} \forall n \in N_{\mathcal{C}_H} \bullet (type(n) = \text{MergeNode}) \vee \\ (type(n) \neq \text{MergeNode} \ \wedge \exists m \in N_{\mathcal{C}_L} \bullet \text{match}(n, m))\end{aligned} \tag{3}$$

The function *noMissingNodes()* aims to ensure that any nodes (e.g., actions, control nodes) that are present in the high-level model must also appear in the low-level counterpart. That implies the behavior described in the low-level model can embrace the expected functions defined in the high-level model. The expected output of *noMissingNodes()* will be a set of nodes that are described in the high-level model but missing in the low-level model. If *noMissingNodes()* yields an empty set, we can assume that the functions described in the low-level model at least embrace the behavior defined in the high-level model. As *noMissingNodes()* is rather straightforward, we present in Equation 3 its short formal description. Please note that *MergeNodes* can be safely ignored in this context because different combinations of merges do not lead to different ordering of the executed action nodes[26] pg. 398.

The function *match()*, which is used by *noMissingNodes()*, takes two model elements as inputs and returns *true* if two elements are matched and *false* otherwise. Intuitively, two matched elements must be of the same type and having the same identifier. Formally, *match()* can be described as shown in Equation 4.

$$\begin{aligned}
&\text{match}(p,q) \overset{\text{def}}{=\!=} type(p) = type(q) \wedge (p.name = q.name)\\
&\quad \wedge\ (type(p) = \text{DecisionNode} \implies \text{guardMatch}(p,q))\\
&\quad \wedge\ (type(p) = \text{JoinNode} \implies \text{specMatch}(p,q))\\
&\quad \wedge\ (type(p) = \text{GuardNode} \implies \text{specMatch}(p,q))
\end{aligned} \tag{4}$$

$$\begin{aligned}
&\text{guardMatch}(p,q) \overset{\text{def}}{=\!=} \forall e_i(p,t_i) \in E_{\mathcal{C}} \bullet\\
&\quad guard(e) \neq null \implies \exists e_o(q,t_o) \in E_{\mathcal{C}} \bullet \text{specMatch}(guard(p), guard(q))
\end{aligned} \tag{5}$$

The definition of *specMatch()* is not presented in detail here because it just defines the equality for all existing sub-classes of the class *ValueSpecification* in UML[26] pg. 139. As this is rather trivial and lengthy, we opt to omit its definition here.

In the definition of the function *noMissingTransitiveLinks()*, we use the conventional definitions of the *adjacency matrix* and *transitive closure* of a directed graph. Let $G = (V, E)$ be a directed graph where V is the set of nodes and E is the ordered set of arcs. The adjacency matrix A_G of G is an $n \times n$ boolean matrix whose elements $A_G[i,j]$ is *true* if $e(i,j) \in E$ and *false* otherwise.

Based on the adjacency matrix A_G, we derive a *reachability matrix* $R_G = A_G^*$ to represent the transitive closure of G. It is denoted as $R_G[i,j] = true$ if there is a directed *path* from node i to node j and $false$ otherwise. The function *noMissingTransitiveLinks()* is defined in Equation 6.

$$\begin{aligned}
&\text{noMissingTransitiveLinks}(\mathcal{C}_H,\mathcal{C}_L) \overset{\text{def}}{=\!=} \forall p,q \in N_{\mathcal{C}_H} \bullet (R_{\mathcal{C}_H}[p,q] = false) \vee\\
&\quad \big(R_{\mathcal{C}_H}[p,q] = true \implies (type(p) = \text{MergeNode} \vee type(q) = \text{MergeNode}) \vee\\
&\qquad (type(p) \neq \text{MergeNode} \wedge type(q) \neq \text{MergeNode}) \wedge\\
&\qquad (\exists p_{match}, q_{match} \in N_{\mathcal{C}_L} \bullet \text{match}(p,p_{match}) \wedge \text{match}(q,q_{match})\\
&\qquad\qquad \wedge R_{\mathcal{C}_L}[p_{match},q_{match}] = true)\big)
\end{aligned} \tag{6}$$

The main idea of *noMissingTransitiveLinks()* is to verify whether any possible execution paths defined in the high-level model are missing in the low-level counterpart. Let $\mathcal{C}_H$ (resp. $\mathcal{C}_L$) be input high-level and low-level check models and $R_{\mathcal{C}_H}$ (resp. $R_{\mathcal{C}_L}$) be the corresponding *reachability graph*. Let us consider an arbitrary pair of nodes (p,q), where $p,q \in N_{\mathcal{C}_H}$. In case there are no paths between p and q in $\mathcal{C}_H$, i.e., $R_{\mathcal{C}_H}[p,q] = R_{\mathcal{C}_H}[q,p] = false$, we do not need to consider the corresponding links in $\mathcal{C}_L$. If a path between p and q exists, i.e., either $R_{\mathcal{C}_H}[p,q] = true$ or $R_{\mathcal{C}_H}[q,p] = true$, there must be a corresponding path in the low-level check model.

The function *noMissingCycles()* is responsible for verifying whether any loops described in the high-level model are not realized by the low-level counterpart. In order to define *noMissingCycles()*, we use Tarjan's algorithm[33] to obtain a set of *strongly connected components* (SCC)[2] in the helper function *getStronglyConnectedComponents()*.

Assuming that $\mathcal{S}_H$ (resp. $\mathcal{S}_L$) is the set of the strongly connected components of $\mathcal{C}_H$ (resp. $\mathcal{C}_L$), we define *noMissingCycles()* in Equation 7. Given an SCC s, N_s and E_s are used to denote the sets of nodes and edges of s, respectively.

$$
\begin{aligned}
&\text{noMissingCycles}(\mathcal{C}_H, \mathcal{C}_L) \overset{\text{def}}{=\!=} \forall s_H \in \mathcal{S}_H \bullet \\
&\quad \big(|N_{s_H}| = 1 \wedge \exists p \in N_{s_H} \wedge \text{hasSelfLink}(p, E_{s_H}) \implies \exists s_L \in \mathcal{S}_L \bullet \\
&\qquad (|N_{s_L}| = 1 \wedge \exists q \in N_{s_L} \bullet \text{match}(p,q) \wedge \text{hasSelfLink}(q, Es_L)) \vee \\
&\qquad (|N_{s_L}| > 1 \wedge \text{allNodesPresent}(s_H, s_L))\big) \\
&\quad \bigvee (|N_{s_H}| > 1 \wedge \exists s_L \in \mathcal{S}_L \bullet \text{allNodesPresent}(s_H, s_L)) \qquad (7) \\
&\text{where} \\
&\mathcal{S}_x = \text{getStronglyConnectedComponents}(\mathcal{C}_x) \\
&\text{hasSelfLink}(n, E) \overset{\text{def}}{=\!=} \exists e = (s,t) \in E \bullet s = n \wedge t = n \\
&\text{allNodesPresent}(G_1, G_2) \overset{\text{def}}{=\!=} \forall n_1 \in N_{G_1}, \exists n_2 \in N_{G_2} \bullet \text{match}(n_1, n_2)
\end{aligned}
$$

The function *noMissingCycles()* will examine two cases: either there is only one element in the nodes of an SCC or more. An SCC with only one node must have a link from that node to itself to form a cycle. If the only node p in a single node SCC has a link to itself, a containing SCC in the low-level model must exist respectively and contain the same cycle. This can be the case, either if the containing SCC is the same single node graph with a self link, or if it is a bigger SCC that contains the node p. If an SCC has more than one element, a bigger cycle is present in s_H, and an SCC $s_L \in \mathcal{S}_L$ must exist that contains all the nodes from s_H. Please note that it is sufficient to show here that the same nodes exist in cycles, as we have already established above (in *noMissingTransitiveLinks()*) that none of the transitive links between nodes is missing.

So far we have described the main ideas along with formal definitions of the containment relationship between two software behavior models. These definitions are the basis of our lightweight graph-based approach for containment checking. In the subsequent part of the paper, we will analyze the theoretical complexity of our approach based on the individual (sub)-functions and conduct a quantitative evaluation of the scalability and applicability of our approach using realistic industrial scenarios derived from our previous projects.

[2]A graph is strongly connected if every vertex is reachable from every other vertex. The strongly connected components of a directed graph form a partition into subgraphs that are themselves strongly connected.

Table 1. Estimation of Theoretical Complexity of Graph-based Containment Checking

Main function	**Sub-function(s)**	**Worst-Case Complexity**
translate$\mathcal{A}$to$\mathcal{C}$	*addPseudoNodes*(G)	$O(\|E_G\|)$
noMissingNodes	*checkMissingModelElements*(G_1, G_2)	$O(\|V_{G_1}\| \times \|V_{G_2}\|)$
	match$(n1, n2)$	$O(\|out(n_1)\| \times \|out(n_2)\|)$
noMissingCycles	*hasSelfLink*(n)	$O(\|out(n)\|)$
	getStronglyConnectedComponents(G)	$O(\|V_G\| + \|E_G\|)$
	allNodesPresent(S_1, S_2)	$O(\|V_{S_1}\| \times \|V_{S_2}\|)$
noMissingTransitiveLinks	*getTransitiveClosure(G)*	$O(\|V_G\|^3)$
	checkMissingTransitiveLinks(G_1,G_2)	$O(\|\mathcal{S}_{G_1}\| \times \|\mathcal{S}_{G_2}\|)$

2.3 Theoretical Complexity Analysis

In this section, we present an analysis of the theoretical worst-case complexity of our approach through the constituent functions. The mapping of an activity model into a check model can be achieved by traversing the set of the activity's edges and converting guarded edges to pseudo nodes as described in Equation 1. This process is represented by the function *addPseudoNodes()* shown in Table 1.

The complexity of *noMissingNodes()* (see Equation 3) can be estimated as two loops over the nodes of two input check models. *noMissingNodes()* uses the function *match()* (see Equation 4) to check whether two arbitrary model elements are matched. We note that *match()* essentially compares the nodes' names, types, and/or guard conditions or *valueSpec*. Thus, the worst-case of *match()* occurs when we analyze the guards associated with the outgoing edges of a *DecisionNode* (we use $out(n)$ to denote the set of outgoing edges of n).

In the function *noMissingTransitiveLinks()*, establishing the *transitive closure* of an input digraph G is obtained by using the traditional Warshall's algorithm[45] that has the time complexity of $O(|V_G^3|)$ where V_G is the number of nodes of G. Then, comparing for transitive links (aka execution paths) of two check models can be quickly performed using the reachability matrix.

The function *noMissingCycles()* comprises three sub-functions. The first function *hasSelfLink()* can perform in constant time. The second function *getStronglyConnectedComponents()* is based on Tarjan's algorithm[33] that has the worst-case complexity of $O(|V|+|E|)$. The third function *allNodePresent()* compares the nodes of two corresponding *strongly connected components* (SCC), and therefore, is bounded by $O(|V_{S_1}| \times |V_{S_2}|)$ where S_1 and S_2 are the aforementioned SCCs. The worst-case complexity of *allNode-Present()* occurs when a strongly connected component under consideration becomes the whole input graph (which rarely, or even never, happens because this implies all nodes of the graph must be connected).

3 Quantitative Evaluation and Discussion

3.1 Quantitative Evaluation

We implement our graph-based approach for containment checking and conduct a preliminary evaluation of its performance. The main idea is to validate whether the proposed approach performs reasonably for typical models used in industry on typical workstations used by developers. We also aim to compare with existing techniques for containment checking. Unfortunately, apart from our early work based on model checking[20], we cannot find any working tools for comparison purpose. Nevertheless, we note that the containment checking approach presented in[20] is based on model checking techniques that can support exhaustively exploring the state space to find out any behavioral inconsistencies. Therefore, this technique, when enabling the exhaustive state space search options, can be considered as an estimated upper boundary of the complexity of the containment checking problem. As a result, we will perform the comparison of our graph-based approach and the model checking based approach[20] to see how well our approach scales with respect to that upper boundary.

The workstation used for the performance evaluation is running Linux on a CPU Quad-Core 2.66 GHz with 2048 megabytes of memory. The two approaches under consideration are implemented in Java and executed with the Java VM 1.7. We note that the model checking based approach uses the NuSMV model checker[4] version 2.5.4 for verifying the containment relationship. Hence, the NuSMV's source code has been instrumented for measuring the corresponding model checking time. The evaluation is conducted through four behavior models extracted from industrial scenarios in network service and banking sectors. These behavior models are, namely, *Order Processing* (OP), *Travel Booking* (TB), *Customer Fulfillment* (CF), *Billing Renewal* (BP), and *Loan Approval* (LA) with different sizes and complexity.

For measuring and comparing the performance of our approach and the exhaustive exploring techniques, we focus on the worst-case scenarios. In theory, the worst-case execution of containment checking taking two input behavioral models is, as shown in the theoretical complexity analysis, when these input models have approximate or equal sizes (the *size* of a behavioral model refers to the number of elements of different types such as nodes, paths, and so on). Thus, in our experiments we take each behavior model and perform the containment checking using different techniques to verify the model against itself. This way, we can estimate the worst-case performance of both approaches. We report in Table 2 our measurement and analysis results based on the aforementioned cases. We also summarize the total execution time of the two techniques and visualize them in Figure 4 to analyze how fast the execution times grow.

In the first part of Table 2, we present the complexity of the input UML activity diagrams in terms of their elements including actions—representing computational or data handling tasks, control nodes—representing the nodes that can change the flow of execution described in the activity diagrams, and edges—representing the links between nodes. In measuring the execution of the graph-based containment checking approach, we observe individual tasks such as model loading, translating activity models to check models (AtoC), checking for missing nodes (cNMN), missing transitive links (cNML), and missing cycles (cNMC), respectively. The execution time of the model checking based approach can be broken down to model loading, translating activity models to NuSMV descriptions (UMLtoSMV), and

Table 2. Performance Evaluation and Comparison

	OP	TB	CF	BR	LA
Input Size					
Action Nodes	11	7	16	22	29
Control Nodes	8	12	8	17	22
Edges	22	24	30	54	63
Guarded Edges	3	8	4	10	19
Total Elements	41	43	54	93	114
Graph-Based Approach					
Model Loading (ms)	3159 ± 112	3377 ± 77	3890 ± 161	5666 ± 137	5142 ± 417
AtoC (ms)	311 ± 38	821 ± 91	447 ± 74	1048 ± 105	1776 ± 239
cNMN (ms)	34 ± 2	24 ± 7	49 ± 3	116 ± 10	216 ± 57
cNMC (ms)	23 ± 2	37 ± 6	30 ± 5	56 ± 3	85 ± 17
cNML (ms)	575 ± 19	723 ± 11	1145 ± 92	6124 ± 97	$15\,045 \pm 1349$
Total Time (ms)	4102	4982	5560	13 009	22 264
Model Checking Based Approach					
Model Loading (ms)	3167 ± 882	3357 ± 80	3862 ± 122	5481 ± 190	5364 ± 135
UMLtoSMV (ms)	535 ± 38	564 ± 30	665 ± 31	1190 ± 29	1590 ± 48
ModelChecking (ms)	$22\,635 \pm 1859$	$32\,978 \pm 1394$	$24\,632 \pm 715$	$56\,606 \pm 2985$	$438\,174 \pm 35\,814$
Total Time (ms)	26 336	36 899	29 159	63 277	445 128

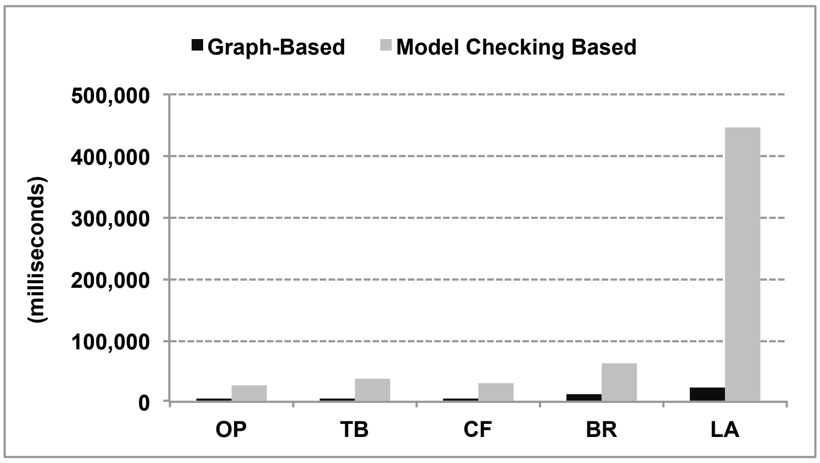

Figure 4. Comparison of the scalability of two approaches

performing model checking. The corresponding time consumed by each task is the average time out of 1000 rounds of execution. Before measuring each task, sufficient warming-up executions are performed to reduce potential confounding factors of class loading and instantiation in Java. The execution time is measured in nanoseconds but rounded and shown in milliseconds for the sake of readability. We also include in Table 2 the corresponding unbiased standard deviation of the execution time of each case.

In accordance with the theoretical complexity analyzed in Section 2.3, the costly aspect of our approach is *noMissingTransitiveLinks()* that consumes reasonable time for building the transitive closure matrix. Nevertheless, the total execution time of the graph-based technique, to the best of our knowledge, is still reasonable for a typical working environment. In the model checking based technique, performing model checking is often a time-consuming task and it grows rather exponentially (see Figure 4).

3.2 Discussions

The containment relationship between two behavior models at different abstraction levels is based on the assumption that a high-level model element and its low-level corresponding have the same name and type. That reflects in the function *match()* where two input elements are compared with respect to their names, types, and/or other associated properties such as guards. The aforementioned assumption is rather realistic because a low-level model is mainly achieved through a refinement of a high-level model where existing high-level elements are often enriched with more details and elements[35]. Nevertheless, in cases of mismatches of their names and types, one possibility to alleviate this problem, like in the approaches on checking behavior similarity[2], is to employ supporting text matching techniques[21].

Breaking down the problem of containment checking into smaller tasks (or functions) as shown in Equation 2 brings a number of advantages. First, these tasks are independent from each other, and therefore, can be performed in any order. For instance, we can perform *noMissingNodes()* to quickly validate whether the low-level model covers all functions specified in the high-level model or *noMissingTransitiveLinks()* to assess that no execution link in the high-level model is missing. Moreover, these tasks can also be executed in parallel to gain better performance. Last but not least, each of these tasks can inform the developers precisely about the causes of the violation of the containment relationship, for instance, due to missing nodes, cycles, or execution paths, along with the involved model elements. To the best of our knowledge, none of existing related approaches has addressed this aspect yet.

The most challenging issues in comparing behavior models are to deal with loops (e.g., a combination of decisions nodes and backward edges or a structured loop node[26] pg. 396) and parallel execution branches (e.g., a combination of fork and join nodes). A loop, especially a conditional loop, cannot be efficiently described by existing property specification logics such as temporal logics[47] that are the formal basis of several model checking based techniques. Parallel execution branches lead to the *state explosion* problem for existing techniques that are based on exhaustive searches through state spaces of the behavior models[5]. We aim to address these issues in our graph-based approach by using the *noMissingTransitiveLinks()* to verify the presence of all possible execution branches and *noMissingCycles()* to verify the presence of corresponding loops in both behavior models.

There is a considerable overhead (growing towards $O(V^3)$) of building the transitive closure (TC) used by our graph-based containment checking technique in *noMissingTransitiveLinks()*. Our implementation of *noMissingTransitiveLinks()* is based on Warshall's algorithm[45]. Nevertheless, Nuutila has presented heuristics for improving Tarjan's algorithm[33] in detecting strongly connected components (SCC) and uses the improved SCC detection techniques (plus a special representation of successor sets) to achieve better TC finding[23]. Our approach is well in line with Nuutila's techniques with respect to the use of Tarjan's algorithm for *noMissingCycles()* and Warshall's algorithm for *noMissingTransitiveLinks()*. However, we opted not to integrate Nuutila's techniques in order to better analyze individual performance. Moreover, tight integration of Nuutila's techniques implies the dependency between *noMissingTransitiveLinks()* and *noMissingCycles()*, and hence, may nullify the potential parallelizability of our approach.

4 Related Work

The consistency checking problem has been extensively studied in the literature[18]. However, very few of these studies focus on the consistency of behavior models[18]. To the best of our knowledge, none of them considers the containment checking problem for behavior models, which verifies whether an expected software behavior specified at a higher level of abstraction is satisfied by a corresponding behavior description at a lower level of abstraction.

Containment checking for behavior models, to a broader extent, is related to the notion of behavioral equivalence[19]. However, behavioral equivalence based techniques are rather not applicable for checking the containment relationship. On the one hand, these techniques produce a binary "*true*" (satisfied)" or "*false*" (unsatisfied) answer but do not provide further concrete information of the inequivalent cases[11,19,40,41]. Several studies have been conducted to alleviate the aforementioned complexity by measuring the degree of behavioral equivalence. On the other hand, Rabinovich showed that the complexity of the behavioral equivalence checking on directed graphs is **NP**-hard, even for a class of simple finite communicating elements[28]. It leads to a considerable number of approaches for alleviating the aforementioned complexity by relaxing the equivalence relationship and, instead, computing the similarity of behavior models in different application domains. For instance, Nejati et al. propose an approach to matching hierarchical state-charts models[22]. Walkinshaw and Bogdanov propose two techniques to compare state machines in terms of language perspective—the externally observable sequences of events (transition labels) and in terms of structure perspective—to compute the precise difference of their actual states and transitions[42].

In the field of business process management, there also is a research trend focusing on computing the similarity of process models[2]. Some approaches concentrate on searching for process models in a large repository that match a given process model or a process fragment thereof[6,7]. Also in this field, there are a considerable number of approaches using trace theory to validate the conformance of two process models or process models against their execution traces recorded in the event logs. An approach presented in[37] checks the conformance between Petri net based process models and their execution traces and returns a degree of behavioral similarity ranging from "*completely different*" to "*identical*". Another approach aims at verifying whether two process models are similar using their corresponding event traces mined from process execution logs[38]. Wang et al. measure the similarity of behavior of process models, based on the *coverability* tree of labeled Petri nets[44]. Bae et al. propose to use distance measures metric for measuring mining process similarity and difference[1]. In this method, dependency graphs are extracted from process models and converted into normalized matrices. Afterwards, metric space distances are calculated based on the difference between the normalized matrices. Weidlich, Dijkman, and Weske consider the compatibility between referenced process models and the corresponding implementation based on the notion of behavior inheritance[46]. We note that, unlike our approach, the aforementioned techniques do not aim at providing precise answers whether two behavior models are equivalent or subsumed nor concrete information about any inconsistencies. These approach rather produce an estimated *degree of similarity* of these models. Therefore, they are very useful for finding similar or alternative behavioral descriptions but not applicable for verifying the containment relationship.

A closely related work is proposed by Eshuis and Grefen for structurally matching BPEL[24] business process models[10]. The main idea of this approach is to transform a BPEL process model into a tree structure and exhaustively compare two trees to find out whether the underlying process models fall into one of four matching categories, namely, "*exact matching*", "*plug-in matching*", "*inexact matching*", and "*mismatched*". This approach leverages the tree structures that are only possible to derive from block-structured behavior models like BPEL processes[3]. Thus, it is not applicable to a wide range of behavior models that allows flexible connections between model elements (e.g. UML activity diagrams[26] Sec. 12, BPMN[25]). Moreover, the tree structure in this approach does not allow cyclic paths, and therefore, cannot be used to verify loop matching as our approach[4].

Our graph-based containment checking approach presented in this paper is illustrated via UML activity diagrams. Like many other behavior models, an activity diagram embraces fundamental constructs to express sequential and concurrent executions, choices, merges, and iterations. Unfortunately, the semantics described in the UML 2 specification[26] is rather informal. As a result, our earlier work presented in[20]—among the others based on model checking techniques—must derive a transformation of the input activity diagrams to equivalent formal descriptions[31,34]. One challenging aspect is that the non-determinism of decision nodes and loop nodes make the translation of the input models to formal properties, e.g., in temporal logics, very difficult and inefficient. Another challenge is that parallel structures, for instance, formed due to the combination of "*ForkNodes*" and "*JoinNodes*" in UML activity diagrams, often cause the *state explosion* problem[5] as we discussed above. In contrast to our approach, most of the model checking based approaches have not considered to provide adequate and concrete information about the causes of the non-equivalence between behavior models. In case inconsistencies exist, the outcomes, for instance, counterexamples[5], are very cryptic for the stakeholders who often have limited knowledge about the underlying formal methods[8]. Last but not least, most of the approaches based on model checking techniques are very time-consuming as shown in our evaluation, and therefore, rather not realistic to apply in complex and large software development settings.

5 Final Remarks and Future Work

In this paper, we presented a graph-based approach for addressing the problem of containment checking of software behavior models at different levels of abstraction. In this approach, the input behavior models are mapped to a formal intermediate representation that can be handled efficiently by graph search algorithms. The containment relationship is formally defined and divided into smaller problems that are resolved by three tasks: finding missing nodes, missing execution paths, and missing loops, respectively. The advantage of this *divide-and-conquer* strategy is twofold. On the one hand, these tasks can be performed independently, and therefore, can be parallelized to gain better performance. On the other hand, each task produces concrete and precise information about the violation of the containment relationship accordingly. The prototypical implementation of our approach performs within the boundary

[3]Please note that the authors consider links in BPEL but restrict the boundary of links inside the containing block in order to create the underlying tree structures.

[4]Please note that the authors introduce loop sensitive matching but only for loops nodes, which is fundamentally different from loop structures formed by cyclic paths.

of $O(n^3)$ where n is the size of the inputs. The quantitative evaluation on industrial scenarios shows that the proposed approach performs reasonably on a typical working environment and scales well within the complexity upper bound of an exhaustive model checking based approach.

Nevertheless, there is still considerable room for improvement in the future. Our research agenda includes support for other constructs of UML activity diagrams, such as exception handling and data flows, as well as to investigate software behavior models radically different from UML activity diagrams, such as state-charts and sequence diagrams. Although the prototypical implementation shows reasonable performance, further integration of improved graph algorithms, for instance for transitive closure finding and strongly connected component detection, is promising in order to gain more improvement in performance.

Acknowledgment The research leading to these results has received funding from the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001. The authors would like to thank the anonymous referees for many helpful comments that greatly improved the presentation of the paper.

References

[1] J. Bae, L. Liu, J. Caverlee, L.-j. Zhang, and H. Bae. Development of distance measures for process mining, discovery and integration. *International Journal of Web Services Research*, 4(4):1–17, Jan. 2007.

[2] M. Becker and R. Laue. A comparative survey of business process similarity measures. *Computers in Industry*, 63(2):148–167, Feb. 2012.

[3] M. Brambilla and P. Fraternali. Large-scale Model-Driven Engineering of Web user interaction: The WebML and WebRatio experience. *Science of Computer Programming*, 89, Part B:71 – 87, 2014.

[4] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *11th International Conference on Computer Aided Verification (CAV)*, pages 495–499. Springer, 1999.

[5] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[6] R. Dijkman, M. Dumas, and L. García-Bañuelos. Graph matching algorithms for business process model similarity search. In *7th International Conference on Business Process Management (BPM)*, pages 48–63, Ulm, Germany, Sept. 2009. Springer.

[7] R. Dijkman, M. Dumas, B. van Dongen, R. Käärik, and J. Mendling. Similarity of business process models: Metrics and evaluation. *Information Systems*, 36(2):498–516, Apr. 2011.

[8] Y. Dong, C. R. Ramakrishnan, and S. Smolka. Model checking and evidence exploration. In *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 214–223, April 2003.

[9] D. D'Souza and M. Mukund. Checking Consistency of SDL+MSC Specifications. In *10th International SPIN Workshop on Model Checking Software*, pages 151–166, Portland, OR, USA, May 2003. Springer.

[10] R. Eshuis and P. Grefen. Structural Matching of BPEL Processes. In *Fifth European Conference on Web Services (ECOWS'07)*, pages 171–180, Halle, Germany, Nov. 2007. IEEE.

[11] R. J. Glabbeek. The linear time - branching time spectrum ii. In *4th International Conference on Concurrency Theory (CONCUR)*, pages 66–81. Springer, 1993.

[12] B. Hailpern and P. Tarr. Model-driven Development: The Good, the Bad, and the Ugly. *IBM Syst. J.*, 45(3):451–461, 2006.

[13] J. Hidders, M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, and J. Verelst. When are two workflows the same? In *Australasian Symposium on Theory of Computing - Volume 41 (CAT)*, pages 3–11, Newcastle,

Australia, 2005. Australian Computer Society, Inc.

[14] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[15] S.-K. Kim and D. A. Carrington. A Formal Model of the UML Metamodel: The UML State Machine and Its Integrity Constraints. In *2nd International Conference of B and Z Users on Formal Specification and Development in Z and B (ZB)*, pages 497–516. Springer, 2002.

[16] R. Laleau and F. Polack. Using formal metamodels to check consistency of functional views in information systems specification. *Information and Software Technology*, 50(7-8):797–814, June 2008.

[17] V. Lam and J. Padget. Consistency checking of sequence diagrams and statechart diagrams using the π-calculus. In *5th International Conference on Integrated Formal Methods (IFM)*, pages 347–365. Springer, 2005.

[18] F. J. Lucas, F. Molina, and A. Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631–1645, Dec. 2009.

[19] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[20] F. U. Muram, H. Tran, and U. Zdun. Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking. In *11th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA)*, pages 93–107, Grenoble, France, Apr. 2014.

[21] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, Mar. 2001.

[22] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *29th International Conference on Software Engineering (ICSE)*, pages 54–64, Minneapolis, MN, May 2007. IEEE.

[23] E. Nuutila. Efficient transitive closure computation in large digraphs. *Acta Polytechnica Scandinavia: Math. Comput. Eng.*, 74:1–124, July 1995.

[24] OASIS. Web Services Business Process Execution Language (WSBPEL). `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`. (Last accessed: August 17, 2015).

[25] Object Management Group. Business Process Model and Notation (BPMN) version 2.0.2. `http://www.omg.org/spec/BPMN/2.0.2/PDF`. (Last accessed: August 17, 2015).

[26] Object Management Group. OMG Unified Modeling Language™ OMG UML, Superstructure, Version 2.4.1. `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`. (Last accessed: August 17, 2015).

[27] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer, 1981.

[28] A. Rabinovich. Complexity of Equivalence Problems for Concurrent Systems of Finite Agents. *Information and Computation*, 139(2):111–129, Dec. 1997.

[29] H. Rasch and H. Wehrheim. Checking Consistency in UML Diagrams: Classes and State Machines. In *Formal Methods for Open Object-Based Distributed Systems (FMOOD)*, pages 229–243, Paris, France, Nov. 2003. Springer.

[30] A. Rozinat and W. M. P. van der Aalst. Conformance testing: Measuring the fit and appropriateness of event logs and process models. In *3rd International Conference on Business Process Management (BPM)*, pages 163–176. Springer, 2006.

[31] Y. Shinkawa. Inter-model consistency in uml based on cpn formalism. In *8th Asia Pacific Software Engineering Conference (APSEC)*, pages 411–418. IEEE Computer Society, 2006.

[32] H. Tai, K. Mitsui, T. Nerome, M. Abe, K. Ono, and M. Hori. Model-driven Development of Large-scale Web Applications. *IBM J. Res. Dev.*, 48(5/6):797–809, Sept. 2004.

[33] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[34] Y. Thierry-Mieg and L.-M. Hillah. UML behavioral consistency checking using instantiable Petri nets. *Innovations in Systems and Software Engineering*, 4(3):293–300, Aug. 2008.

[35] H. Tran, U. Zdun, and S. Dustdar. Name-based view integration for enhancing the reusability in process-driven SOAs. *International Journal of Business Process Integration and Management*, 5(3):229—-239, 2011.

[36] A. Tsiolakis and H. Ehrig. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In *Workshop on Graph Transformation Systems (GRATRA)*, pages 77–86, 2000.

[37] W. M. P. van der Aalst, A. K. A. de Medeiros, and A. J. M. M. Weijters. Process equivalence: Comparing two process models based on observed behavior. In *4th International Conference on Business Process Management (BPM)*, pages 129–144. Springer, Sept. 2006.

[38] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance checking of service behavior. *ACM Transactions on Internet Technology*, 8(3):1–30, May 2008.

[39] R. van der Straeten. *Inconsistency Management in Model-Driven Engineering An Approach using Description Logics*. Doctoral dissertation, Vrije Universiteit Brussel, 2005.

[40] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *Theories of Concurrency: Unification and Extension (CONCUR)*, pages 278–297. Springer, 1990.

[41] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G. X. Ritter, editor, *Information Processing 89: IFIP 11th. World Computer Congress*, pages 613–618, San Fransisco, CA, USA, 1989. Elsevier Science Publishers B.V., North-Holland.

[42] N. Walkinshaw and K. Bogdanov. Automated Comparison of State-Based Software Models in Terms of Their Language and Structure. *ACM Transactions on Software Engineering and Methodology*, 22(2):1–37, Mar. 2013.

[43] H. Wang, T. Feng, J. Zhang, and K. Zhang. Consistency check between behaviour models. In *IEEE International Symposium on Communications and Information Technology (ISCIT)*, volume 1, pages 470–473. IEEE, Oct. 2005.

[44] J. Wang, T. He, L. Wen, and N. Wu. A behavioral similarity measure between labeled Petri nets based on principal transition sequences. In *Confederated International Conferences: CoopIS, IS, DOA and ODBASE*, pages 394–401, Hersonissos, Crete, Greece, Oct. 2010. Springer.

[45] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, Jan. 1962.

[46] M. Weidlich, R. Dijkman, and M. Weske. Behaviour Equivalence and Compatibility of Business Process Models with Complex Correspondences. *The Computer Journal*, 55(11):1398–1418, Feb. 2012.

[47] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.

[48] S. Yao and S. Shatz. Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. In *2006 15th International Conference on Computing*, pages 289–297. IEEE, Nov. 2006.