Comparing model checkers for timed UML activity diagrams✩

Zamira Daw *, Rance Cleaveland

Department of Computer Science, University of Maryland, College Park, MD 20742, USA

A R T I C L E   I N F O

Article history:
Received 8 June 2014
Received in revised form 4 May 2015
Accepted 14 May 2015
Available online 27 May 2015

Keywords:
Model checking
UML activity diagrams
UPPAAL
SPIN
NuSMV

A B S T R A C T

This paper describes the results of an experimental study on the use of model checkers to verify properties of UML activity diagrams. The motivation for the study derives from the desirability of checking properties of systems early in the development process, and the fact that UML is a commonly used notation for system models. The study assesses the performance of different model checking tools, and strategies for converting activity diagrams into the tools input notation, for a class of real time activity diagrams used in medical device design. This paper compares different translations for four model checkers in particular: UPPAAL, PES, SPIN and NuSMV. The performance of these model checkers is then compared using a suite of UML activity diagrams of varying complexity developed by us for this purpose. The results of a case study involving the design of an infusion pump are also presented.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

The use of digital-control technologies in complex embedded applications has dramatically grown in the last decades because of ongoing improvements of microprocessors with regard to their price, processing performance, memory and power consumption. In industries such as automotive, aerospace and medical-devices, control functionalities have been increasingly realized as software components running on general-purpose microcontrollers and communications platforms rather than via special-purpose hardware. More sophisticated functionalities and more complex processing units (e.g. multicore) at the same time have led to an increase in the cost of the software construction, and also an increase in the risk for software errors, which depending on the application can cause significant device damage or even endanger the users.

Model-driven development (MDD) is increasingly used in order to overcome the new challenges in the development of embedded systems. In MDD, developers specify the system in terms of models, which may then be used as a basis for analyzing system design and generating source code. There is a general trend towards the use of behavioral models for specifying systems for these purposes, in particular synchronous data-flow models such as MATLAB/Simulink [1], block diagrams in SCADE [2] and activity diagrams in the Unified Modeling Language (UML) [3,4]. UML in particular has attracted significant attention in embedded software development in both academia and industry, because of the fact that it is a general purpose modeling language that can be easily adapted to a specific domain via syntactic and semantic extensions, and also because of its status as a non-proprietary and independently maintained standard. UML syntax is defined by a metamodel, and its semantics is described precisely, but informally, in natural language.

✩ Research supported by NSF Grant CCF-0926194. This paper is an extension of [11].
* Corresponding author.
E-mail addresses: zdaw@cs.umd.edu (Z. Daw), rance@cs.umd.edu (R. Cleaveland).

http://dx.doi.org/10.1016/j.scico.2015.05.008
0167-6423/© 2015 Elsevier B.V. All rights reserved.
In order to mitigate the risk of software errors, software should be formally verified against system requirements. In an MDD setting, this can be realized by verifying whether development models meet such requirements and establishing that the code derived from a model is semantically consistent to the model. In this paper, we focus on the first task: the verification of the development models. Since these models are developed in the design phase, any requirements violations detected in the models can be repaired before the full system is developed, thereby reducing cost and effort.

Model checking is an automated mathematical verification technique that systematically checks whether a system satisfies a given requirement. The most common approach to formally verifying UML models consists of a tool chain that translates UML models into the mathematically well-defined language supported by a model checker, followed by the verification of the translated models. With mathematically well-defined models, we refer to models with semantics defined using mathematical formalisms. Standard UML has a semi-formal semantics that leaves room for free interpretation. On one hand, this enables diverse uses of UML; on the other hand, this lack in the semantics impedes a direct formal verification of UML models into more mathematically precise notations. To enable the translation into the languages supported by model checkers, the UML semantics must be completely defined for the specific system domain. An optimal verification depends on the adequate choice of the model checker and the translation with respect to the development model type (e.g. UML activity diagrams, etc.) and to the system domain (e.g. Business Process Modeling BPL). In the embedded-systems domain, some specific aspects need special consideration: these include timing deadlines, limited memory and processing units, and overwriting of data, in particular in signal-processing applications.

This paper is focused on the model checking of UML activity diagrams in the embedded-computing domain. Such diagrams contain a graphical representation of processing flows of nodes, which represent functions that take inputs and convert them into outputs. These diagrams allow the modeling of choice, iteration and concurrency. Tool-chains for the verification of UML activity diagrams are presented in [5–7], which use NuSMV as the model checker, as well as in [8,9], where SPIN is used to model check the models. Although all these approaches analyze the same types of diagrams, there are some differences in the semantics due to the aforementioned aspects of UML, and there is a general lack of quantitative results regarding model-checker performance. This fact makes the decision about which model checker to use a difficult one for tool-chain builders.

The main contributions of this paper are: 1) an experimental study of the influence of the model checker and translation strategy on the performance of the formal verification of UML activity diagrams; 2) the development of eight translation approaches from UML activities into model-checker-supported notations; and 3) a UML activity-diagram-based verification approach for embedded systems. The model-based verification approach integrates a verification tool chain into the MDD framework DMOSES [10] (Section 3). Different translations for the model checkers NuSMV [15], SPIN [12], UPPAAL [13], and PES [14] are presented in Section 4. The first two tools are general-purpose finite-state model checkers, while the second two are specifically targeted at timed systems. The proposed tool chains are evaluated by means of test models used as benchmarks and by verifying system requirements of an infusion pump (Section 5). In Section 6, we discuss the results of the experimental study and present relevant information about the choice of the model checker and the translations. Such information aims to help the building of verification tool-chains for other development models and system domains. We conclude the paper and present future work in Section 7.

2. Background and related work

This section gives an overview of different model-checking approaches for the verification of UML activity diagrams and Petri Nets (PN). It should be noted that the activity diagram semantics has been changed from a state-machine-based framework in UML 1.x to a PN-based model in UML 2.x. This change has led to conceptually different approaches for the model checking of activity diagrams. Because of the close semantic correspondence between UML 2.x activity diagrams and PNs, this overview surveys PN-related model-checking approaches as well as those focused purely on UML.

NuSMV NuSMV [15] is a symbolic model checker based on binary decision diagram techniques. The NuSMV language specifies systems as finite-state transition systems and requirements as formulas in CTL (Computation tree logic) or LTL (Linear temporal logic). The NuSMV language allows the description of systems via re-usable modules. Furthermore, concurrent asynchronous processes are also supported. The state space of the system is determined by bounded variables. A transition system is defined in terms of a state space, a transition relation, and a set of initial states.

Some approaches propose NuSMV for the verification of UML activity diagrams [5–7]. Eshuis [5,16] verifies workflows by translating UML activity diagrams into intermediate models called hypergraphs, followed by the generation of NuSMV-models based on these graphs. Since Eshuis worked with an extended version of UML 1.5, this translation cannot be directly used for verifying UML 2.x activity diagrams. However, some semantic approaches in that work have relevance for the newer UML standard. Nodes are modeled as Boolean variables, which indicate if the node is active or not. A node becomes active if its preceding nodes are also active. The activation/deactivation of all nodes is synchronized by a step-semantics. In contrast to Eshuis, Lam [7] and Grobelna [6] work with UML 2.x activity diagrams, whose semantics are based on PN. Lam translates UML nodes and edges directly into Boolean variables. The functionality of each type of activity node is implemented

---

1 DMOSES supports the automated generation of real-time control software from activity diagrams extended with information regarding execution time, parallelism and priority.
using case-structures. Grobelna transforms UML activities explicitly into PN and then translates these into the NuSMV input language. Additional optimizations are performed in the obtained PN in order to reduce the state space. The state of the system is defined by the places of the PN (parallel places are grouped into one state). All possible system transitions are defined by a case-statement. In contrast to these approaches, our formal semantics is based on token flow instead of node activation, covers a wider number of UML elements, and specifies the semantics directly using the NuSMV language. Other approaches [17–19] that verify PN using NuSMV define places as Boolean variables and transitions using case-statements. Depending on the application area, the translation of a place creates additional variables required for the verification. In addition to case-statements used in these approaches, our approach uses also a NuSMV TRANS-system in order to identify the performance influence of this structural difference. The verification of timed Petri Nets (TPN) using NuSMV is performed by Bobbio [19]. The timing semantics of the TPN are defined by a timing range in which enabled transitions can be fired. In order to reduce the state space, this approach proposes to discretize firing intervals using Kronecker algebra. In contrast to this approach, which uses a time variable for each transition, our translation reduces the state space by reusing the timers for activity elements that are executed in the same hardware platform. Furthermore, our study only considers the worst-case execution time in order to analyze the performance of non-real-time model checkers.

**SPIN** The SPIN model checker [12] verifies distributed systems specified in PROMELA (PRocess MEtaLanguage). Systems can be described as a set of asynchronous processes, which can interact via channels or shared variables. Requirements to be verified are defined using linear temporal logic (LTL) formulas. SPIN converts these formulas into Büchi automata and computes the system state space by as the synchronous interleaving product between the automata of the system and the automata of the requirements. SPIN generates a problem-specific on-the-fly verification program instead of performing the verification itself; compiling and running this program gives the verification result. In order to allow SPIN to verify timing requirements, Bošnački [20] presents a PROMELA extension that supports discrete real-time modeling based on a timeout-statement. Clocks are incremented when this statement becomes executable, which occurs when there is no other executable process in the system. Our proposed approach for modeling timing is also based on the timeout-statement. However, our clock is incremented by multiple ticks at once when this is semantically allowable, rather than on a tick-by-tick basis, thereby improving verification performance.

SPIN has been used to verify PN in [21–23]. Ribeiro [21] uses PN to model embedded systems; this approach maps places into global Boolean variables and the entire PN into a flat structure by using a do-statement. A similar approach for business processes is presented by Sai [22], with the difference that the places are byte-variables, which allows the analysis of data overwriting. Ganood [23] proposes a channel-based approach, which maps places into channels and transitions into processes. In contrast to the flattening approach of the previous two methods, this distributed structure allows the modeling of concurrent processing. With regard to UML activity diagrams (2.x), Cao [8] verifies Business Process Execution Language (BPEL) models via UML activities. The author uses channels to support communication between activity nodes, which are implemented as processes. In contrast, our channel-based translation reduces the usage of channels to one for each hardware platform by encoding the edges as messages. Guelfi [9] in contrast handles inter-node communication using global variables and allows verification of timing constraints by introducing timers in the PROMELA-model based on the work of Bošnacchi. While Guelfi assigns timers to nodes according to the structure of the activity, we assign timers according to the processing units given in the model. Furthermore, Guelfi’s semantics is based on the activation of nodes instead of token flows.

**Model checkers for timed automata** Although the previously described model checkers support the verification of UML activity diagrams, they do not intrinsically support a notion of time in the input system. Given the importance of time in embedded applications, this paper also considers model checkers that specialize on timed models given as timed automata. Timed automata (TA) [24] are finite automata extended with a finite set of real-valued clocks. There are different verification tools for TA; this study analyzes two TA model checkers: UPPAAL [13] and PES [14]. We choose an extended definition of TA presented by [25]. A timed automaton with variables is defined as a tuple $TA = (L, L_0, L_u, VR, VR_0, \Sigma, CX, I, E)$, where:

- $L$ is the finite set of locations.
- $L_0 \subseteq L$ is the nonempty set of initial locations.
- $L_u \subseteq L$ is the set of urgent locations.
- $VR$ is the set of data variables.
- $VR_0 \subseteq DV_d$ is the set of data valuations representing the possible initial values of the data variables (often the single assignment of 0 for all variables, provided $0 \in Z_f$), where $DV_d$ is the set of all data valuations and $Z_f \subseteq Z$ and $|Z_f| < \infty$ (data valuation is a mapping of data variables into a restricted set of integers).
- $CX$ is the nonempty finite set of clocks.
- $I : L \rightarrow \Phi(CX \cup VR)$ gives the invariant for each location $l$.
- $E \subseteq L \times \Sigma \times \Phi(CX \cup VR) \times 2^{CX} \times DA \times L$ is the set of edges. In an edge $e = (l, a, \phi, \lambda, dA[V], l')$ from $l$ to $l'$ with action $a, \phi \in \Phi(CX \cup VR)$ is referred to as the guard of $e$, which has constraints both on clocks and variables. $\Phi(CX \cup VR)$ is the set of all possible constraints. $\lambda$ is the set of clocks to reset and $dA[V]$ represents the data assignment function that gives new values to the data variables in $V \subseteq VR$. $DA$ represents the set of all data assignments.
UPPAAL is the most commonly used model checker in academia and in the industry for TA. UPPAAL provides a user-friendly environment for modeling, simulation and verification. Systems can be modeled as networks of timed automata (NTA), extended with data types (bounded integers, arrays etc.). Automata can be given via templates, thereby allowing multiple instantiations and parameterization. Automata can communicate via variables and channels, which can be defined as global elements or as parameters of the templates. A channel synchronizes multiple edges with the same channel (e.g. $a+b$). Urgent channels additionally ensure that no delay occurs. In UPPAAL, the properties to be checked are restricted to a subset of TCTL (timed computation tree logic). In order to verify NTA, UPPAAL computes the parallel composition between the TA during verification ("on-the-fly").

UPPAAL has not been used for the verification of UML activity diagrams, so we only present verification approaches for PNs that use this model checker. Byg [27] presents the TAPPAAL tool, which translates Time-Arc Petri Nets (TAPN) into TA. TAPNs are an extension of PNs in which a token is assigned a specific age and arcs restrict tokens of specific ages. Because of this particular extension, tokens are modeled as a single timed automaton with one local clock. This automaton takes the restrictions of the arcs into account. Similar to Byg, we specify systems as the parallel compositions of TA. However, our semantics is based on describing node behavior instead of token behavior. Cassez [28] translates TPNs into TA. A transition in the TPN is mapped into a timed automaton, which describes the behavior of the transition using the states: enable, disable, and firing. Transitions between these states are triggered depending on the mapping of the net defined using an array of integers. The TPNs are defined as the parallel composition of the TA of the transitions plus a supervisory TA that is responsible to synchronize all transitions of the TPN. This work has served as a basis for Cicirelli to verify Time Stream Petri Nets in [29] and Preemptive Time Petri Nets in [30]. Similar to Cassez and Cicirelli, our centralized-control translation has a supervisor TA. Since a node in an activity has more behaviors than a single transition in a PN does, we use three TA to handle nodes behavior. Gu [31] analyzes embedded real-time systems modeled by TPNs. Similar to Cassez, transitions are implemented as TA and places as integer variables. Furthermore, transitions have an execution time, which is similar to our modeling of the execution of a node. Guan’s group also compares NuSMV vs. UPPAAL for multiprocessor scheduling analysis in [32] and [33], and concludes that NuSMV achieves better performance.

PES is a model checker based on predicate equation systems [14]. Parametric timed automata use variables as parameters to encode the state of the system. The requirements to verify are given as first-order mu-calculus formulas. PES performs the model checking by computing solutions of parametric equation systems. The proof search combines forward and backward techniques in order to speed up the termination in cases where errors are detected and at the same time to enable efficient computation when no errors are present. The verification of TA is a specific application of PES [26]. To date this tool has not been used for the verification either of UML activities or of PN.

3. Integrating model checking into DMOSES

This paper aims to study multiple model-checking approaches in order to identify which tool chain performs best for UML activity diagrams used for embedded systems. The specific version of activity diagrams used were defined for the DMOSES development process, which provides end-to-end modeling and code generation for real-time control applications. This section describes DMOSES and how model checking is incorporated into the framework.

3.1. The DMOSES development process

DMOSES [10] is a model-driven development method for hybrid embedded systems. In this method, systems are specified using UML behavioral models, which are automatically transformed into executable source code for different hardware platforms. The semantics of UML activity diagrams and state machine diagrams is extended in order to model functional and non-functional aspects relevant to the development and the analysis of embedded systems. The DMOSES method is supported by an Eclipse plug-in (see the DMOSES website\(^1\)).

The DMOSES development process is based on four main steps: Modeling, Code Generation, Implementation and Analysis (Fig. 1). In the Modeling step, the developer specifies the system using interconnected UML activities and state machines, which are extended by the DMOSES profile. The Code Generation step is divided into two transformations: Model-to-Model (M2M), which transforms UML models into DMOSES models, and Model-to-Text (M2T), which generates source code based on the DMOSES models and is defined using templates. The DMOSES-models are intermediate models that abstract information about the system behavior required for the development. The structure of these models, which is specified in the DMOSES metamodel, facilitates the generation of code in different programming languages and of analysis models (e.g. graphs). In the M2T-transformation, each programming language has its own templates, which are based on the implementation of the behavior defined by the extended UML models.

The DMOSES method allows adapting the granularity of models by defining atomic elements at the code level, which can represent anything from a simple mathematical operation to a complete algorithm. In the Implementation step, developers

---

\(^2\) We refer to hybrid embedded systems as systems that combine traditional CPUs with reconfigurable devices (FPGAs) or Application-Specific Integrated Circuits (ASICs).

\(^3\) http://www.emb.hs-mannheim.de/Query?node=102854&language=1.
can implement atomic elements into wrapper structures, which are automatically created by the code generator or can be chosen from existing libraries. Afterwards, the source code is ready to be compiled or synthesized. The DMOSES method performs several analyses through the entire development process. In this paper, we only focus on the timing analysis. The timing behavior of atomic elements is determined by worst-case execution time (WCET) tools, which analyze the source code. The UML models are enriched with this information, thus becoming timed models. That enables the analysis of time requirements, especially for real-time embedded systems.

3.2. Integrating formal verification in DMOSES

This section describes the integration of formal verification into DMOSES (it can be used for other MDD processes as well). The integration is implemented as a plugin in the DMOSES Eclipse environment, which involves a tool chain that translates the system described by UML models into the input notation of the chosen model checker, followed by formal verification via the model checker (Fig. 2). The translation consists of two steps: optimization and generation. The optimization step abstracts information from the timed DMOSES models relevant for the formal analysis. This step also reduces the state space of the system by using graph-based optimizations.

Timing values are optional. Thus, the models can be formally verified without timing information (e.g. in early phases of the development before code generation) or by considering real timing values of the system (e.g. in final phases of the development). Requirements are specified by developers using temporal logic. The generation step is a template-based transformation, which translates extended UML activities and the logical formulas into the model checker’s input notation. Templates are based on the description of the UML activity behavior using a specific model checker. Section 4 introduces different translations of the UML activity behavior for several model checkers.

DMOSES extended UML activity diagrams In order to explain the translations into the model checkers that are described later, the rest of this section describes activity diagrams in general, and the extended UML activity diagrams used in DMOSES.

Activity diagrams specify a behavior by using interconnected nodes. Nodes can be connected via edges or events. The UML standard specifies different kinds of nodes. In this paper, we are focused on the following types: action, call behavior action, join, fork, merge, decision, parameter set, pin, initial node, flow final, send signal action, and accept event action. The structure of an extended UML activity diagram is defined by the tuple \( a = (N, E, \Sigma, H) \) where:

- \( N \) is a set of nodes. A node is a tuple \( n = (\text{type}, t_{\text{exe}}, \sigma, c, h) \), where type is the type of the node, \( t_{\text{exe}} \) is the maximal execution time, \( \sigma \) is the event that can be sent or received, \( c \) is an activity associated to the node, and \( h \in H \) is the processing unit that executes the node.
\( E \) is a set of edges. An edge is a tuple \( e = (s, t, p, \text{async}) \), where \( s \) is the source node, \( t \) is the target node and \( s, t \in N \), \( p \in \mathbb{N}^+ \) is the priority of the edge and \( \text{async} \in B = \{ \text{true}, \text{false} \} \) defines if the flow is asynchronously executed.

- \( \Sigma \) is a set of events that can be sent or received within the activity.
- \( H \) is a set of processing units that execute the activity.

The UML metamodel defines the constraints of the activity structure in detail (e.g., send signal actions and accept event actions are the only node types that can send and receive an event, respectively). Actions are the atomic elements within the UML activity diagram and are notated as rounded rectangles. An action resembles a mathematical function; it takes a set of inputs and converts them into a set of outputs. The properties \( t_{\text{ref}}, h, p, \) and \( \text{async} \) are added to activity diagrams by the DMOSES profile. The \( p \) and \( \text{async} \) properties allow modeling the functionality and the execution of the system separately, thereby facilitating the management of concurrent processing and ensuring a unique specification of the implementation. In activity diagrams, a system can be described in different hierarchical levels by using the node type CallBehaviorAction. This type allows instantiating an action, whose behavior is defined by an activity. A CallBehaviorAction is marked by a rake in the UML diagram and its behavior is specified in the above-mentioned structure as the parameter \( c \) in the definition of node.

UML activity diagrams have a token-based semantics. Tokens flow through edges can contain data of a specific type or just a control signal. A node is executed when a required number of tokens is available in its input. These tokens are consumed in the execution and new ones are created at the end of the execution, which are offered in the output. According to the extended semantics, output tokens are fired according to the properties of the edge, such as priority and \( \text{async} \). Thus, the firing process waits until the previously fired flow-processing is finished before firing the next token if the edge is not \( \text{async} \). The required number of tokens depends on the type of the node and its connections. However, most node types can be divided in two groups in regard to the required tokens: OR-Type that requires one token in one input (merge, decision) and AND-Type that requires one token in all the inputs (action, fork, join, send, call behavior action, send signal action, accept event action). Thus, an action and a fork with same inputs, same outputs, and same execution time behave the same in relation to the token flow and the timing behavior. Therefore, the translation from UML activity diagrams into the mathematical well-defined languages supported by model checkers is based on the node types. In addition to the above-mentioned nodes, UML activity diagrams have other nodes (ParameterSet) and semantics (initialization/ finalization of an activity, and communication between send and accept) that have a more complex behavior specified as a constellation of OR-Type nodes and AND-Type nodes to facilitate the implementation and the optimization (see Appendix A).

The DMOSES activity diagram shown in Fig. 3 requires a token in the input \( \text{(in1)} \) for its execution. As a result of the activity execution, two tokens are offered in the outputs \( \text{(out1, out2)} \). This activity contains two parallel flows, which start in the output of the Fork node. The first flow consists of the action A and the second flow contains of the nodes Decision, B, C, D. Both flows are sequentially executed since no edge has the async property. The order of the execution is defined by the priority level of the edges. The highest priority corresponds to the lowest number. The action A is invoked as consequence of the firing of the first outgoing token of the Fork node. After 60 cycles of the execution, the action A offers a token in its output, which sets the output \text{out1} of the activity. In this moment, the execution of the first flow is finished. The processing of a flow finishes when it reaches a FlowFinal, an output of an activity or the flow does not have nodes that can be executed due to the lack of tokens. Therefore, the processing returns to the Fork node that fires the second token through the outgoing edges with the following highest priority. This invokes the Decision node, which fires a token in one of its outputs by evaluating the input value, as a result, only one action of the actions B and C is invoked. After the execution of B or C, the outgoing token sets one input of the action D. However, this one token is not enough to execute the action D since it required at least two tokens. This means that no tokens are set in the output \text{out2}. This is a very common error in activity diagrams, which can be difficult to find in models with several activities and multiple hierarchical levels.
Table 1  
Timing modeling in PROMELA using an interval increase-based approach.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int clk = 0;</td>
</tr>
<tr>
<td>2</td>
<td>int interval = 0;</td>
</tr>
<tr>
<td>3</td>
<td>For every $h \in H$</td>
</tr>
<tr>
<td>4</td>
<td>int $t_h = 0$;</td>
</tr>
<tr>
<td>5</td>
<td>proctype incTime() (</td>
</tr>
<tr>
<td>6</td>
<td>idle: atomic{</td>
</tr>
<tr>
<td>7</td>
<td>timeout &amp;&amp; $\bigvee_{h \in H} (t_h != 0) \rightarrow$</td>
</tr>
<tr>
<td>8</td>
<td>d_step(</td>
</tr>
<tr>
<td>9</td>
<td>if</td>
</tr>
<tr>
<td>10</td>
<td>for every $h \in H$</td>
</tr>
<tr>
<td>11</td>
<td>$::(t_h != 0) &amp;&amp; (\bigwedge_{h' \in H \setminus {h}} (t_{h'} == 0)</td>
</tr>
<tr>
<td>12</td>
<td>$::else \rightarrow$ skip;</td>
</tr>
<tr>
<td>13</td>
<td>fi;</td>
</tr>
<tr>
<td>14</td>
<td>clock = clock + interval;</td>
</tr>
<tr>
<td>15</td>
<td>for every $h \in H$</td>
</tr>
<tr>
<td>16</td>
<td>$::(t_h != 0) \rightarrow t_h = t_h - interval;</td>
</tr>
<tr>
<td>17</td>
<td>$::else \rightarrow$ skip;</td>
</tr>
<tr>
<td>18</td>
<td>fi;</td>
</tr>
<tr>
<td>20</td>
<td>)</td>
</tr>
</tbody>
</table>
| 21   |     goto idle; |}

4. Verification tool chains

A verification tool chain verifies UML models by translating them into a model checker’s input notation and running the model checker on the input. The translation is based on the token-based semantics of the UML models explained in the previous section and the modeling language supported by the model checker. This section presents eight translations of UML activity diagrams into the input notations of the four studied model checkers: SPIN, NuSMV, UPPAAL and PES. Several of the tools have different translation schemes given in order to show how different approaches to translation can affect verification performance.

4.1. SPIN

For the SPIN model checker, we present three translations, which differ in their model the structure and communication between the processes. All the translations share a discretized model of time. The transformation rules of each translation are shown as a combination of PROMELA-code (e.g. $d_{\text{step}}$), elements of extended UML activities (e.g. $h$ for processing unit) and functions that facilitate the understanding of the rule (e.g. $\text{isOR}(n)$ returns true if $n$ is OR-Type).

4.1.1. Timing modeling

Timing modeling allows verification of timing deadlines and other real-time requirements based on the execution time of nodes. Since SPIN does not directly provide time verification, we develop a timing modeling based on the timeout-statement as shown in Table 1. This statement is executable when there is no other executable process. The timing behavior is modeled by using execution timers ($t_h$), a time-increase routine ($\text{incTime}$), and a real-time clock ($clk$).

Execution timers specify the number of cycles a processing unit must perform to execute a node to completion. After a node is invoked, the node sets the timer for its processing unit to its execution time ($t_{exec}$) and waits until the timer is zero to offer tokens in its output. Execution timers are decreased by a specific interval in the time-increase routine (Line 17). The interval is determined by the minimum time that current processing units have to work i.e. the minimum timer (Line 11). The usage of a time interval reduces the state space by treating several time values as one state. Note that the real-time clock is also increased in the time-increase routine (Line 14). This clock indicates how much time (in cycles) has passed since the beginning of the system execution. In contrast to the timers that specify the timing behavior of the system, the real-time clock only helps to verify timing requirements.

4.1.2. Variable-based translation

This translation distributes the activity behavior by modeling nodes as separate processes as is shown in Table 2. The processes communicate to each other via the variables $\text{in}_n$, which records the total number of the incoming tokens, and $\text{endF}_n$, which indicates the completion of the flow started by the node $n$.

At the beginning of the procedure, nodes are waiting for the required numbers of tokens to be executed (Line 7). The function $\text{reqInputs}(n)$ returns the required number of tokens for node $n$, which depends on the type of node and the number of incoming edges (e.g. OR-Type nodes such as Merge nodes only require one token for the execution). In this paper, translations take only the number of required tokens into account and not at which pin they arrive in order to simplify the translations for the comparison. If the required tokens are available, the incoming tokens are consumed (Line 8), followed
by the setting of the execution timer of the corresponding processing unit \( (t_{exe}) \). The set value corresponds to the WCET of the node \( (t_{exe}) \). The timer is decreased by the previously presented time-increase routine.

After the execution of a node is finished, which is denoted by zero in the timer, the firing process begins, which consists of adding tokens to the target nodes \( \text{in}_f \) \((\text{Lines 19, 20})\). The function \( \text{outgoingEdges}(n) \) returns outgoing edges of the node \( n \), which are sorted by the priority. The jumps to the label \( \text{firing}_e \) determine when a token is fired through the outgoing edge \( e \). For OR-Type nodes, only one token is fired and the outgoing edge is non-deterministically chosen via an \( \text{if} \)-statement \((\text{Lines 11–15})\), thereby allowing the model checker to analyze all possible paths of a token. For AND-Type nodes, the first outgoing edge is fired without any jump. The remainder of the edges are sequentially fired \((\text{firing\_next})\) depending on the priority \((\text{Line 20})\). The firing of an edge ends after firing a token if the target node cannot be immediately executed \((\text{Line 20})\). Otherwise, the firing process waits until the processing flow of the target node is finished as indicated by the variable \( \text{endF}_f \) \((\text{Line 22})\). Once the last outgoing edge is fired, the firing process of the node is finalized by indicating that the flow of the node is completely finished i.e. by setting the variable \( \text{endF}_n \) to one \((\text{Line 27})\). After that, the process returns to the \( \text{idle} \) state.

### 4.1.3. Channel-based translation

Similar to the previous translation, the channel-based translation implements nodes as processes \((\text{Table 3})\). However, these processes are connected via channels instead of variables. In order to avoid creating a channel for each UML edge, this translation creates one channel for each processing unit, which is implemented as a rendezvous synchronization channel with a single parameter \((\text{Line 2})\). Through these channels, messages corresponding to tokens and signals for flow-processing termination are transferred. A message \( \text{id}_n\text{ln} \) indicates that a token is sent to the node \( n \) and a message \( \text{id}_n\text{end} \) indicates the completion of the flow started by the node \( n \). Note that messages are constant values determined by the function \( \text{uniqueID} \) \((\text{Lines 4–5})\). Channels are given as parameter to the node processes in the \( \text{init} \)-process.

The number of received tokens is saved in the local variable \( \text{inT} \), which is increased when a message with the ID \( \text{id}_n\text{ln} \) is received \((\text{Lines 10–11})\). Once the required number of tokens has been received \((\text{Line 13})\), the execution begins in the same way as the variable-based translation \((\text{Lines 17–19})\). In case that there are not enough tokens, the node sends a signal for flow-processing termination \((\text{Lines 14, 33})\) and continues waiting for more tokens. The firing process \((\text{Lines 20–28})\) is performed by sending messages \((\text{tokens}\text{id}_n\text{ln}) \) that correspond to the IDs of the target nodes \((\text{Lines 22, 27})\). After firing a token, the node waits \((\text{Lines 23, 27})\) until receiving a message that corresponds to the flow-processing termination \((\text{tokens}\text{id}_n\text{EndF})\). For AND-Type nodes, the firing order is based on the priorities of the edges \((\text{Lines 20–23})\). For OR-Type nodes, one token is fired through an outgoing edge, which is non-deterministically chosen \((\text{Lines 24–28})\). After finishing the firing process of the node, the message \( \text{id}_n\text{EndF} \) is sent through the channel \( \text{tokens} \) \((\text{Line 33})\), thereby indicating that the processing flow is finished.
4.1.4. Flat translation

The flat translation specifies the behavior of the entire system in a single process as shown in Table 4. Note that three blocks can be identified in the translation: the do-statement (Lines 7–15), the execution_n (Lines 17–24), and the firing_e (Lines 26–34).

The first block determines which node to execute or which edge has to fire a token according to the priority and the source node type of the edge. This decision is based on the variables in_n, endF_n and out_n. The second block and the third block specify the node execution and the token firing, respectively.

The execution of the node is invoked (execution_n), if there is the required number of input tokens (Line 9). The execution procedure begins by consuming the required tokens and starting the timer (Line 18). After the execution is finished, the firing process begins. For OR-Type nodes, an if-statement determines which outgoing edge has to be fired by jumping to a firing_e-label (Lines 20–24). For AND-Type nodes, the first outgoing edge is directly fired after the execution ends (Line 28) and the firing of remaining edges (Line 30) is invoked from the do-statement (Line 12). Note that the flow of the current firing outgoing edge has to be completely processed (endF_e) in order to fire the next edge. The current firing outgoing edge is determined by the variable out_s. A token firing consists of adding a token to the target node (in_t) and resuming the flow-processing termination of the previous fired edge (endF_e preTarget(e)) as is shown in line 30. The resetting is omitted for edges that do not have a previous fired edge (Line 28). In case that a target node does not have the required tokens for executing, the firing process of the edge terminates after the token transfer; therefore, the flow-processing of the outgoing edge is finished (Line 32). Note that after a token is fired, the process jumps to the idle-label and then to the do-statement (Line 34). The firing process of the node ends when the flow-processing of the last outgoing edge for AND-Type nodes or an outgoing edge for OR-Type nodes is finished (Lines 14).

4.2. NuSMV

We develop two translations for NuSMV: a modular translation and a flat translation. Since NuSMV does not intrinsically support a notion of time, we use a discrete-time model that is shared by both translations. The transformation rules are shown as a combination of NuSMV-code and functions of extended UML activities.
4.2.2. The finish therefore, state variable real-time

In the

The behavior of this system is based on the following (Lines 2–8): the number of received tokens (inT_n), the last fired outgoing edge (outT_n), the current controlling node (system), an execution timer (timer), and a time-increase module (proc).

4.2.1. Timing modeling

The modeling of time in NuSMV is based on an execution timer (timer), a time-increase module (incTime), and a real-time clock (clock) as is shown in Table 5.

The variable clock indicates how much time (in cycles) has passed since the beginning of the system execution. The variable state determines if the processing unit is free to process a node (idle) or if the processing unit is already done with the last node (finish). NuSMV does not provide functionality similar to the timeout functionality of SPIN; therefore, the UML activity controls when the time increase has to occur using the input variable timer. When the timer is set to −1, no time passes. Otherwise, the clock is incremented by the value of the timer and the state changes to finish. This state indicates to the UML activity that the processing of the node is finished. Note that this module assumes a single processing unit.

4.2.2. Modular translation

In this translation each variable of the system is specified by a case-statement. Table 6 shows the translation rules. The behavior of the system is based on the following (Lines 2–8): the number of received tokens (inT_n), the last fired outgoing edge (outT_n), the current controlling node (system), an execution timer (timer), and a time-increase module (proc).
The variable system specifies which node is allowed to be executed or to fire tokens. The update of this variable is determined by activity edges (Lines 20–23). This variable changes from the source node to the target node of an edge after firing a token (Line 22), and from the target node to the source node after flow-processing termination (Line 23). Note that the first executed node is specified by the function rootNode. The case-statement of timer is responsible for node execution (Lines 26–30). A node is executed if the required number of tokens is reached and the processor is idle (Line 28). The execution starts by setting the timer to $t_{exec}$. The end of the execution is indicated by the variable proc.state, which is set to finish.

Once the execution finishes, the firing process begins. This process is divided in two steps: choosing an outgoing edge and adding a token, which are implemented in the case-statements of outT_n (Lines 38–40) and inT_n (Lines 32–37), respectively. If the node is an AND-Type the firing process chooses an outgoing edge according to the priority (Lines 39, 41); otherwise, the output is chosen non-deterministically (Lines 39). Note that the variable outT_n has two additional values to the output number (Line 5). The value 0 indicates that a node is not in the firing process, while the maximal value indicates the firing process of the node is finished. Therefore, the flow-processing termination flag (endF_n) is based on this variable (Line 11). This flag is used to choose the next outgoing edge to be fire (Line 41). The case-statement of inT_n adds a token to a target node if the source node is in the corresponding output (Line 34).

### 4.2.3. Flat translation

The flat translation uses propositional formulas to specify the system behavior instead of case-statements (see Table 7). This translation is based on the same variables as the modular-translation. Lines 11 and 12 summarize information about

<table>
<thead>
<tr>
<th>Table 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformation algorithm from extended UML activities into NuSMV using the modular translation.</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>33</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>36</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>39</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>41</td>
</tr>
<tr>
<td>42</td>
</tr>
<tr>
<td>43</td>
</tr>
<tr>
<td>44</td>
</tr>
</tbody>
</table>
the time processing: \texttt{start} indicates that the processing unit is free to process the next node, and \texttt{ready} signals that the processing of the last node has finished and stops the passing of time by setting the timer to \texttt{-1}. The behavior of the system is specified using six types of formulas (Lines 13–23). Note that all formulas use the function \texttt{only}(), which returns NuSVM-code that forces all variables to maintain the current value except for the given input parameter (e.g. if \texttt{only(timer)} is written, only the variable \texttt{timer} is allowed to change). The first formula refers to the start of the node execution when the node has the required number of tokens (Line 15). This formula sets the timer of the corresponding node to \texttt{texec}. After the execution is finished, tokens are consumed (\texttt{inTn/s}) and the first outgoing edge to be fired is chosen (\texttt{outTn/s}). That is specified in Line 17 for \texttt{AND-Type} nodes and in Line 20 for \texttt{OR-Type} nodes. Note that for \texttt{OR-Type}, the formula is generated for each outgoing edge. Since all these multiple formulas are valid after the execution of the node, the model checker non-deterministically chooses one of them. The following two mutually exclusive formulas fire a token (\texttt{inTf}) through an outgoing edge (\texttt{outTf}). The first is valid if the target node can be immediately executed (Line 21), thereby implying that the firing process must wait until the flow of the target node is executed. Otherwise, the second is valid (Line 22), and as a consequence, the firing process continues with the next output for \texttt{AND-Type} nodes or is finished for \texttt{OR-Type} nodes by setting the variable \texttt{outTs}. After the target node ends its firing process, the last formula resets \texttt{outTf} and chooses the next outgoing edge to be fired (Line 23).

### 4.3. UPPAAL

We define two translations for the model checker UPPAAL: a centralized-control translation and a distributed-control translation. Both are composed of multiple parallel timed automata (TA). However, these translations differ in the control of the token flow. While the centralized-control translation has a TA that specifies the token flow, the distributed-control translation divides the control of the token flow into several TA corresponding to every node.

#### 4.3.1. Centralized control translation

The centralized-control translation specifies the behavior of an activity as a network of timed automata (NTA) (Algorithm 1). This NTA is based on the composition of a token-flow control TA (\texttt{TS}) and a set of TA that describes basic common behaviors of nodes, such as collecting tokens (\texttt{Tl}), execution (\texttt{TC}), and firing tokens (\texttt{TO}) as shown in Fig. 4. In order to describe the behavior of a specific node, these three behaviors are implemented using templates (Lines 13–15). All TA in the NTA communicate via urgent channels, which are specified in the set actions \texttt{\textbackslash{A}'} (Line 4). A channel synchronizes edges with same channel (e.g. \texttt{a!} with \texttt{a?}).

\texttt{TS} sends tokens to a node \texttt{n} via the channel \texttt{inToken}_{\texttt{n}} (Line 12); these are received by \texttt{Tl}_{\texttt{n}} (Line 13). \texttt{TO}_{\texttt{n}} defines through which output a token is fired via the channel \texttt{outToken}_{\texttt{n}}. However, the \texttt{TS} determines the target node (Line 8). Also, the \texttt{TS} indicates to the \texttt{TO}_{\texttt{n}} when the previous fired outgoing edge is completely processed via the channel \texttt{endFlow}_{\texttt{n}} (Line 9).
Algorithm 1: Translation from UML activities into timed automata based on the centralized control translation.

Data: \( A = (N, E, \Sigma, H) \)

Result: \( NTA = (TS)\) if \( (TI_0)_{i=1}^{\mid E\mid} (TC_0)_{i=1}^{\mid E\mid} (TO_0)_{i=1}^{\mid E\mid} \) \( \) where \( TS = (L, \lambda_0, L_0, V, V_0, A', C, I, E') \)

1. \( L := \{id_n, idln_n \mid n \in N\} \)
2. \( \lambda := \text{rootNode}(A) \)
3. \( \lambda_0 := \{idln_n \mid n \in N\} \)
4. \( V := \{busyn, outTn \mid n \in N\} \)
5. \( V_0 := \{busyn = false, outTn = 0 \mid n \in N\} \)
6. for all the \( e \in E, e = (s, t, p, \text{async}) \) do
7. \( p_e = \begin{cases} p & \text{if isAND}(s) \\ 1 & \text{if isOR}(s) \end{cases} \)
8. \( fTrans_e = (id_e, outToken_e, ?, outT_e := p_e, \emptyset, \emptyset, idln_e) \)
9. \( bTrans_e = (id_e, endFlow_e, !, outT_e := p_e \wedge \text{busyn}, \emptyset, \emptyset, id_e) \)
10. end
11. for all the \( n \in N \) do
12. \( in_n = (idln_n, \emptyset, \text{inToken}_n, \emptyset, \emptyset, id_n) \)
13. \( Ti_n := \text{inputTemplate}(\text{reqInputs}(n), \text{busyn}, \text{inToken}_n, \text{execute}_n, \text{finish}_n) \)
14. \( TC_n := \text{calculateTemplate}(c_e, t_{ex}, \text{execute}_e, \text{ready}_e) \)
15. \( TO_n := \text{outputTemplate}(\text{reqOutputs}(n), \text{outT}_n, \text{ready}_n, \text{outToken}_n, \text{endFlow}_n, \text{finish}_n) \)
16. end

![Diagram](image)

Fig. 4. Basic common behaviors of UML activity nodes defined by TA templates.

An activity edge \( e \) is translated into two transitions: a forward transition \( (fTrans_n) \), which transports the fired tokens (Line 8), and a backward transition \( (bTrans_n) \), which indicates the flow-processing termination (Line 9). These transitions have guards related to the variable \( outT_n \), which determines the firing order based on the priority of the edge. Variable \( \text{busyn} \) indicates the status of the processing of the flow started by the target note. For OR-Type nodes, \( fTrans \) has the same guard for all edges; thus the transitions are enabled at the same time, allowing the model checker to verify all the possible paths of the token (Line 7). Note that \( fTrans \) has \( idln_n \) as target location instead of \( id_e \). This is because in UPPAAL one transition cannot have multiple actions. Therefore, two locations are created in \( TS \) for each node (Line 1). While the transition to an \( idln_n \) indicates that a token has been fired to the node \( n \) (Line 8), the transition from an \( idln_n \) triggers the token reception in the \( TI_n \) via the channel \( \text{inToken}_n \) (Line 12).

In order to describe the behavior of a specific node, \( TI, TC \) and \( TO \) are parameterized to the number of required inputs (\( \text{reqInputs} \)), required outputs (\( \text{reqOutputs} \)), the execution time of the activity (\( t_{ex} \)), and clock of the processing unit (\( c_e \)). \( TI \) collects the incoming tokens and starts the execution via the channel \( \text{execute}_n \) when the required number of inputs is reached (Fig. 4(a)). \( TC \) models the execution of the node by using the corresponding clock (Fig. 4(b)). Thus, the finalization of the executions (\( \text{ready}_n \)) occurs when a specific amount of time has passed (\( t_{ex} \)). \( \text{ready}_n \) initializes the firing process defined by \( TO \) (Fig. 4(c)). For each outgoing token, the \( TO \) synchronizes with the \( TS \) using the channel \( \text{outToken}_n \). Afterwards, the \( TS \) synchronizes with the \( TI \) corresponding to the target node of the outgoing edge. The \( TO \) remains in the state until the \( TS \) indicates that the flow-processing is finished by using \( \text{endFlow}_n \). The firing process will continue until all outgoing tokens have been fired and their flows have been processed. The finalization of the firing process is signaled on channel
Algorithm 2: Translation from UML activities into timed automata based on the distributed control translation.

Data: $A = (N, E, \Sigma, H)$
Result: $NTA = (\{TIn\}[1]...|\{TIn\}[n]|\{TEo\}[1]...|\{TEo\}[n])$ where $TIn = (L_{in}, I_{in}, L_{out}, V_{in}, V_{out}, A_{in}, C_{in}, I_{in}, E_{in})$ and $TEo = (L_{in}, I_{in}, L_{out}, V_{in}, V_{out}, A_{in}, C_{in}, I_{in}, E_{in})$

1. forall the $n \in N$ do
   2. $L_{in} := \{\text{idle, received, executing}\}$; $L_{out} := \{\text{firing}\}$; $C := \{c_h \mid h \in H\}$
   3. $V_{in} := \{\text{outTn} = 0, \text{busyn} = \text{false} \mid n \in N\} \cup \{\text{busyn}_{\text{target}} = \text{false} \mid \alpha \in \text{outgoingEdges}(n)\}$
   4. $V_{out}^{\alpha} := \{\text{inTn} = 0, \text{busyn} = \text{false} \mid \text{busy}_{\alpha} = \text{false}\}$
   5. $T_{start} := \{\text{received, execue}\!, \inTn \geq \text{reqInputs}(n) \land \text{busyn}_{\alpha, \text{false}, \vartheta, \vartheta}, \text{executing}\}$
   6. $T_{end} := \{\text{executing}, \text{finish}_n, \vartheta, \vartheta, \text{busyn} = \text{true} \land \text{reqInputs}(n)\}$
   7. forall the $i \in \text{incomingEdges}(n)$ do
      8. $A_{in} := \{\text{inTn}, \text{endFlow}\!\}$
      9. $E_{in} := \{(\text{idle, inTn}, \vartheta, \vartheta, \{\text{inTn} = +, \text{inExe} = i\}, \text{received}\}$
      10. $E_{out} := \{\text{received, endFlow\!\!\!, inTn < reqInputs(n) \lor \text{busyn}_{\alpha} \land \text{inExe} = i, \vartheta, \vartheta, \text{idle}\}$
   11. end
   12. end

13. forall the $n \in N$ do
   14. $L_{in} := \{\text{idle, calculating, firing, waiting}\}$; $L_{out} := \{\text{firing}\}$; $C := \{c_h \mid h \in H\}$
   15. $V_{in} := \{\text{inTn}, \text{busyn} = \text{false} \mid \text{busy}_{\alpha} = \text{false} \mid n \in N\}$
   16. $V_{out}^{\alpha} := \{\text{inTn} = 0, \text{busyn} = \text{false} \mid \text{busyn}_{\alpha} = \text{false} \mid \alpha \in \text{outgoingEdges}(n)\}$
   17. $T_{start} := \{\text{calculating}, c_h \leftarrow \text{tstart}\}$
   18. $E_{in} := \{\text{tstart}, \text{tend}\!, \inTn \cup \text{Exe} = \{\text{execute}, \text{finish}_n, \text{false}, \vartheta, \vartheta\}$
   19. $T_{start} := \{\text{idle, calculate}\!, \vartheta, c_h, \text{busyn} = \text{false}, \text{calculate}\}$
   20. $T_{end} := \{\text{tend}, \vartheta, c_h \leftarrow \text{tstart}, \vartheta, \text{outTn} = 0, \text{firing}\}$
   21. $T_{end} := \{\text{firing}, \text{finish}_n, \text{outTn} = \text{reqOutputs}(n), \vartheta, \text{busyn} = \text{false}, \text{false}\}$
   22. forall the $\alpha \in \text{outgoingEdges}(n)$ do
   23. $p_0 := \begin{cases} p & \text{if isAND}(s) \\ 0 & \text{if isOR}(s) \end{cases}$
   24. $A_{in} := \{\text{inTn}, \text{endFlow}\!\}$
   25. $E_{in} := \{(\text{firing, inTn}, \text{outTn} = p_0 \land \text{busyn}_{\alpha} = \text{false}, \text{busyn}_{\alpha} = \text{false}, \text{firing}\}$
   26. $E_{in} := \{(\text{firing, inTn}, \text{outTn} = p_0 \land \text{busyn}_{\alpha} = \text{false}, \text{busyn}_{\alpha} = \text{false}, \text{firing}\}$
   27. $E_{in} := \{(\text{waiting, endFlow\!\!\!, outTn} = p_0, \text{busyn}_{\alpha} = \text{false}, \text{busyn}_{\alpha} = \text{false}, \text{firing}\}$
   28. end
   29. end

finish_{\alpha}, which allows the node (busyn_{\alpha} = false) to start a new execution. This variable also enables forward transitions of the node defined in the TS.

4.3.2. Distributed control translation

This translation specifies the behavior of a UML activity diagram as a network of timed automata (NTA) consisting of TA that describe the behavior of nodes (Algorithm 2). In contrast to the previous translation, a part of token flow is defined within every TA that corresponds to the node. This means that every node has information about the target nodes of its outgoing edges. Due to the additional information about the token flow, the behavior of the nodes cannot be abstracted into one basic behavior as we did for the centralized control translation.

The behavior of a node $n$ is specified via two TA: $TIn_n$, which models the input behavior (Lines 1–12), and $TEo_n$, which models the execution and the firing process (Lines 13–29). Both TA contain information about the token flow. $TIn_n$ has three locations: idle, which is the initial location; received, which indicates that a token has been received; and executing, which indicates that a node is executing. The transition $t_{start}$ is triggered when the required number of tokens has been received; as consequence, $TIn$ changes to the location executing (Line 5). The functions reqInputs and reqOutputs ensure the correct number of tokens for all node types. After the execution is finished, as denoted by channel finish_{\alpha}, $TIn$ consumes the required tokens and returns to the location received in order to determine if a new execution is possible (Line 6). $E_{in}$ contains transitions between idle and received; these are triggered by reception of a token (Line 9) or an indication of the termination of an execution flow (Line 10).

$TEo$ contains four locations: idle; calculating, which indicates that the node is being executed; firing, which determines that the node is in the firing process; and waiting, which implies that the node is waiting for the processing termination of a previous fired flow. The execution of the node begins by triggering $t_{start}$ (Line 19), which is synchronized with $TIn$ via the channel execute_{\alpha}. Because of the invariant in the calculate location, $TEo$ is forced to change to the location from calculate to firing at the moment that the time $t_{exe}$ has elapsed (Line 20). The firing process of a node is defined by the outgoing edges and the node type (Lines 22–28). An outgoing edge is translated into three transitions. The first transition is triggered if the target node cannot be executed by the fired token (Line 25). This indicates that the flow of the target node is finished, therefore, the firing process can continue with the next edge. Otherwise, the second transition is triggered and the firing process has to wait until the target node finishes its flow processing (Line 26). The variable outTn indicates which outgoing edge has to be fired according to the priorities. For OR-Type nodes, all transitions have the same outTn-value allowing a
Algorithm 3: Translation from UML activities into timed automata based on the distributed control translation for PES.

Data: $A = (N, E, \Sigma, H)$

Result: $\mathcal{I} = (I, I_0, I_d, V, V_0, A', I, E')$

1. $L := \{\text{root} \}; \quad L_0 := \text{root} \}; \quad L_d := \emptyset \}; \quad A' := \emptyset$

2. $V := [n, \text{in}_n, \text{out}_n \mid n \in N]; \quad V_0 := [n = \text{idle}, \text{in}_n = 0, \text{out}_n = 0 \mid n \in N]

3. $C := \{x, c_1 \mid h \in H\}; \quad J := \{(n = \text{firing}, x \in 0, (n = \text{calculating}, c_0 \leq t_{\text{exe}}^n) \mid n \in N\}

4. $E' := \bigcup_{n \in N} E'_n$

5. forall the $n \in N$

6. $E'_n = \{t_{\text{start}}, t_{\text{end}}\} \cup T_{\text{fire}}$

7. if $0 \in \text{inputs}(n)$ then

8. for $i = 0$ to $\text{inputs}(n) - 1$ do

9. $t_{\text{start}} = (\theta, \emptyset, (n = \text{calculating} \land \text{in}_n = n + 1), x, (n = \text{firing}, \text{out}_n = 0, \text{in}_n = 1), \emptyset)

10. end

11. else

12. $t_{\text{start}} = (\theta, \emptyset, (n = \text{calculating} \land \text{in}_n \leq \text{reqInput}(n)), x, (n = \text{firing}, \text{out}_n = 0, \text{in}_n = 0), \emptyset)

13. end

14. $T_{\text{fire}} = \bigcup_{\text{outgoingEdges}(n)} T_0$

15. forall the $o \in \text{outgoingEdges}(n)$ do

16. $T_o = \{t_{\text{fire}}(o)\} \cup \{t_{\text{fire}}(o), (0 \leq i \leq \text{inputs}(t) - 2) \cup \{t_{\text{fire}}(o), (0 \leq j \leq \text{inputs}(t) - 1)\}$

17. $t_{\text{fire}}(o) = (\emptyset, \emptyset, (s = \text{firing} \land \text{out}_i = p \land t = \text{idle} \land \text{in}_i = \text{reqInput}(t) - 1), c_s)$

18. $t_{\text{end}} = (\emptyset, \emptyset, (s = \text{waiting} \land \text{out}_i \leq p \land t = \text{firing} \land \text{out}_i \leq \text{reqOutput}(t) \land \text{in}_i \leq \text{reqInput}(t)), x, (s = \text{firing}, \text{out}_i = p + 1, t = \text{idle}), \emptyset)

19. if $\text{AND}(T_{\text{fire}})$ then

20. for $i = 0$ to $\text{inputs}(t) - 2$ do

21. $t_{\text{fire}}(o) = (\emptyset, \emptyset, (s = \text{firing} \land \text{out}_i = p \land t = \text{idle} \land \text{in}_i = i), x, (s = \text{firing}, \text{out}_i = p + 1, \text{in}_i = i + 1), \emptyset)

22. end

23. end

24. for $j = 0$ to $\text{inputs}(t) - 1$ do

25. $t_{\text{fire}}(o) = (\emptyset, \emptyset, (s = \text{firing} \land \text{out}_i = p \land t = \text{idle} \land \text{in}_i = j), x, (s = \text{firing}, \text{out}_i = p + 1, \text{in}_i = j + 1), \emptyset)

26. end

27. end

28. end

nondeterministic choice. The last transition is fired when the flow-processing corresponding to the target node is finished, as triggered by $\text{endFlow}_n$ (Line 28). After all outgoing tokens have been fired, the transition $t_{\text{end}}$ is triggered, which returns the $T_{\text{en}}$ to $\text{idle}$ and notifies the $T_{\text{in}}$ that the processing of the node is finished via the channel $\text{finish}_n$ (Line 21).

4.4. PES

Any behavior described using PES is determined by variables; there is no notion of channel or location native to the tool. The state of the system is defined by the combination between the variables, and the transitions of the system are triggered by specific values of the variables or by timing events. Therefore, the description of UML activity diagrams has no location or actions, as is shown in Algorithm 3. Since PES does not support parallel TA, the translation encodes the entire behavior in one TA. Three variables are defined for each node — $n, \text{in}_n, \text{out}_n$ — and one clock $c_i$ for each of the processing units. In order to model urgent states, an additional clock $x$ is added.

The variable $n$ specifies the state of the node: $\text{idle}, \text{calculating}, \text{firing}, \text{and waiting}$. The invariant defines that all $\text{firing}$ states are urgent as well as all state $\text{calculating}$ have to wait for $t_{\text{exe}}$ (Line 3). The set of transitions of the TA is specified via five types of transitions: $t_{\text{start}}, t_{\text{end}}, t_{\text{fireFree}}, t_{\text{fireBusy}}, t_{\text{start}}$ is triggered when the execution has finished, as given by the clock $c_i$ (Lines 9, 12). This transition consumes the required tokens. While $t_{\text{start}}$ depends only on the node, the remainder of the transitions types depend on the edges. These transitions synchronize the states of the node and target node. Transition $t_{\text{fireExe}}$ is triggered when a target node $(t)$ can be executed as a result of the firing of a token from the corresponding source node $(s)$, as shown in Line 17. Note that the target node can be executed only if $t$ is in $\text{idle}$ and only one token is required for the execution. This transition changes the source node to $\text{waiting}$ and the target node to $\text{calculating}$. The source node remains in this state until the processing flow of the target node is finished, which is triggered by transition $t_{\text{end}}$ (Line 18). $t_{\text{end}}$ returns the state of the source node to $\text{firing}$ and indicates the next outgoing edge by incrementing the variable $\text{out}_T$. Furthermore, the state of the target node changes to $\text{idle}$, where it waits for tokens. In case that the outgoing token does not immediately trigger an execution of the target node, either $t_{\text{fireFree}}$ or $t_{\text{fireBusy}}$ is triggered. Transition $t_{\text{fireFree}}$ is activated when the target node does not have enough tokens to be executed, while $t_{\text{fireBusy}}$ is activated when the target node is busy in another execution. Both transitions are self-transitions for the source node from/to the state $\text{firing}$ and increase the variable $\text{out}_T$ by one.
5. Experimental study

This section describes the experimental comparison of the different model checkers and translation schemes. The experimental set-up consists of benchmarks, translations, model checkers and a desktop computer. The benchmarks and translations can be downloaded. The benchmarks are composed of a set of 67 UML activity diagrams, which are designed to cover a wide spectrum of UML activity node types. The models also vary in terms of numbers of nodes and edges and use different abstraction levels, in order to determine the performance tendencies of the tool chains for activities with different characteristics. Some of the models also contain common modeling errors in activity diagrams to ascertain that the tool chains can detect these errors.

The translations have been implemented as an Eclipse plugin derived from the metamodel-based code generator ooW [34]. The input format of the translations is XMI (XML Metadata Interchange), which is supported by Eclipse and most UML tools. The model checker inputs are generated as in Section 3. Unnecessary state space is reduced by an optimization on the graph-level based on UML model structure. The optimization is carried out before the translations to the model checkers. For evaluation purposes, one reachability formula is generated for each node in the UML activity; these formulas are used to evaluate the tool chain performance by measuring the verification time needed for each formula. All model checkers have been configured to perform full verification; tool-specific approximations, such as SPIN’s bit-state feature, were not used. In SPIN, an LTL formula was used. Once the UML activities are translated, the model checker verifies if the model satisfies the reachability formulas for that model. In this study, every experiment was run 10 times in order to obtain an average of the performance. The experiments were run on a Windows 8 PC with an Intel Core i5-2500 CPU and 8 GB of memory.

5.1. Translation evaluation

The tool chains are grouped in relation to their respective model checker. The performance of every tool chain is shown as a box-and-whisker diagram. For visualization purposes, only the 10 most representative activities of the benchmark are shown. The activities are ordered in relation to the size from left to right. The box-diagram shows the maximum, minimum, first quartile, third quartile, mean and the median of the verification time of an activity. Note that by the verification of an activity we mean the sum of the model-checking times of all of the reachability properties for the activity.

Translations are compared using a scatter-graph. Each axis corresponds to the verification time of a tool chain, and every point in this graph represents the total verification time of an activity. A dotted line (x = y) is added to the scatter-graph in order to facilitate the analyses of the results. Points above of the reference line indicate that the tool chain corresponding to the horizontal axis has a better performance for the corresponding activity.

5.1.1. UPPAAL

The performance of the centralized and distributed control translations are shown in Figs. 5(a) and 5(b). Note that the verification time increases proportional to the size of the activity. Furthermore, there is an offset value represented by the minimum value, which also varies with the size of activity. The dividing line within the box represents the median and the point represents the mean. In the distributed translation, the mean and the median are more similar than in the centralized translation. That is because the ratio between reachable states and verification time is higher in the centralized translation. Fig. 5(c) shows that the distributed translation has a better performance for all experiments than the centralized control translation. The order of the difference grows with the size of the activity. Figs. 5(a) and 5(b) show a difference of factor four in Act65.

5.1.2. PES

The performance of the translation for PES is shown in Fig. 5(d). Similar to the UPPAAL distributed control translation, the mean and the median values of the translation for PES are very similar. However, the offset of the verification remains almost constant for activities with different sizes. Note that the verification time is in between both translations for UPPAAL and is only 2 times slower in Act65 than the best UPPAAL tool chain.

5.1.3. SPIN

The performance of the three translations for SPIN is shown in Fig. 6. All translations present a small deviation from the mean. This results from the verification approach of SPIN, which first creates a problem-specific verification program. Therefore, all translations for SPIN have a significant offset, which represents the time for the creation of the program. Note that the offset considerably increases with the size of the activities. The deviations from the mean correspond to the execution time of the problem-specific verification program. Fig. 6(e) shows the percentage of the verification in relation to the total time (creation of the program plus verification).

Fig. 6(d) compares the performance of the channel-based translation and the variable-based translation against the flat translation, which shows a better performance in the box-diagram. The figure indicates that the flat translation and the variable-based translation have a similar performance. Furthermore, the results reveal that the channel-based translation is

---

Fig. 5. Performance of the tool chains for TA–model checkers. Pairs represent the number of vertexes and edges corresponding to an activity after the optimization.

less optimal than the other translations, in particular for bigger activities. Note that the difference between the performances increases with the activity size.

5.1.4. NuSMV

The presented translations for NuSMV show a very low performance in comparison with the other evaluated model checkers, even for small activities. Therefore, we only analyzed the performance of 20 activities verified by the modular translation and 40 activities verified by the flat translation.

Fig. 7 shows the comparison between the performances of the flat translation and the modular translation. The vertical axis is logarithmic since the difference between the performances is very large. The result reveals that the flat translation is definitely the best translation for NuSMV.

5.2. Tool-chain evaluation

The translations with the best performance for UPPAAL, PES, and SPIN are compared in Fig. 8(a). The horizontal axis corresponds to the verification time of the translation for UPPAAL, while the vertical axis is shared for the translations of PES and SPIN.

The figure shows that the translation for UPPAAL achieves the best performance since all points are above the reference line. The translation for PES requires less time for all the activities than the translation for SPIN.

NuSMV is separately compared due to the large difference in the performance to the other model checkers. Fig. 8(b) shows the comparison between the best translations of NuSMV and UPPAAL. The horizontal axis corresponds to the verification time of UPPAAL, and the vertical axis presents logarithmically the verification time of NuSMV. This figure shows only the comparison for small activities since the large difference is already present there. Note that UPPAAL-axis only reaches 0.4 seconds.

Since the TA-model checkers show a better performance, we verify the same models without the time component (cycles = 0), in order to determine if the results are caused by the TA-model checkers handling of real time. The mean
Fig. 6. Performance of the tool chains for SPIN. Pairs represent the number of vertexes and edges corresponding to an activity after the optimization.

Fig. 7. Performance comparison of the activity translations for NuSMV.
and the median of the difference between the verification time for timed and non-timed models is shown in Table 8. The results show that the differences in the verification time are similar to the ones with the time component, which reaffirms the previous performance evaluation.

Preliminary experiments were conducted in order to analyze the influence of OR-Type nodes in the performance of the tool-chains: the distributed-translation for UPPAAL and the variable-based translation and the flat-translation for SPIN. These experiments have the same structure as the presented benchmarks, with the exception that all nodes are OR-Type nodes. Results of this preliminary study show that verification time of the distributed translation for UPPAAL is still the best. Furthermore, the flat-transition and the variable-based transition for SPIN can only verify activities with up to 77 nodes and 140 nodes, respectively, because of memory limitations. This preliminary study suggests that memory and verification time in UPPAAL for checking reachability of a node is only related to the distance between the root and the node. By contrast, the performance of SPIN is influenced by the position of the firing within the if-statements (go to firing_e) by the search strategy (depth-first vs. breadth-first).

5.3. Infusion pump

A common way to administer medication to patients in clinical settings is via infusion pumps. Infusion pumps are considered safety-critical systems since a failure during operation can lead to the degradation of the patient’s health or even their death due to over- or under-medication. Therefore, a formal verification of the system is highly desirable. This case study presents the development of the control system of an infusion pump using the DMOSES method. The infusion pump system is modeled using extended UML activities, and automatically generated for the ARM7 hardware platform (LPC2368).

The main functionalities of the infusion pump system are: the execution of the infusion and the monitoring of the infusion process. The execution of the infusion controls the speed and the on/off time of the motor that performs the infusion of the medication (Fig. 9). The diagrams present ParameterSets that are represented by the marked pins. The user gives the Volume per hour (VH) in mL/h and the infusion time using a keyboard, which is controlled and processed in node Input Processing. Based on this information, the motor is controlled. Furthermore, the current injected volume as well as the set VH is shown on a display. The runtime monitoring of the infusion process verifies that the infusion is performed as expected. The current status of the infusion is determined on the basis of different sensors, which measure the velocity of the motor, battery level, and the position of the injection. In case of an error, the system reacts in a way that the patient cannot be injured (e.g. alarm, stopping the motor). Part of this reaction can be seen in the activity Motor_control, in which the current frequency of the motors is compared against the set frequency of the motors, followed by the sending of signals to a state machine that controls the reaction to an error. An example of such a reaction is the signal KOV (keep open vein), which indicates to activity System Control that the velocity of the motors has to be set to a specific value that maintains an open vein without administration of medication.

The execution of the infusion is modeled via UML activity diagrams, and the monitoring of the infusion process is modeled by interconnected UML activities and UML state machines. In this paper, we only analyze the models corresponding
to the execution of the infusion since we are evaluating only the verification of activity diagrams. Table 9 shows the total time of the reachability verification of the execution of the infusion for the best translations of UPPAAL, PES, and SPIN. UPPAAL continues to achieve a better performance than the other tools. PES performed worse in relation to SPIN and UPPAAL than in the previous experiments.

6. Discussion

In contrast to a direct description of systems by using model-checker languages, the performance of translation-based verification from models can be strongly affected by choices in the translation strategy as well as by the model checker itself. This is due to the fact that small differences in the translation of a model element are magnified when there are many such elements in a given model. Therefore, in our study we analyzed multiple translations schemes for the same model checker as is shown in Section 4. The results concretely demonstrate that the choice of these artifacts can have a substantial impact on the performance of the system verification.

The distributed translation for UPPAAL differs from the centralized translation in two main aspects: 1) elimination of an intermediate TA, 2) the combination of the executing and firing behavior into one TA. Both aspects contributed to a 21% performance improvement, which makes the difference for large models. This difference is attributable to the fact that the centralized approach incurs more communication overhead since the supervisory TA handles all decisions regarding execution and token firing. Nevertheless, the centralized translation allows a more simple and faster generation and is easier to understand.

The UML activity translation for PES is similar to the distributed translation for UPPAAL. However, PES can leverage the advantage of synchronizing multiple transitions. To model the same behavior, a TA in UPPAAL requires additional states in order to synchronize different channels in one transition. It is worth to note that PES allows more expressiveness in the specification of the requirements, although the user interface is primitive.

The presented channel-based translation for SPIN has only one channel for the communication between the nodes, unlike some related works, in order to reduce memory. However, the results show that the verification time is strongly affected by the use of channels for this kind of application. The flat and the variable-based translation have similar performance. That indicates that a modular structure such as the variable-based translation, which is easier to understand and generate, does not have substantial influence in the performance in comparison to the flat structure.

![UML models of an infusion pump developed with DMOSES.](image)
In contrast to SPIN, the performance of NuSMV is significantly affected by the modularity of the structure. Although the flat translation for NuSMV performs better than the modular translation, it is not close to any other model checker in our experiments. This is attributed to the fact that there is a proportional relation between the state space and the variables, which represent execution time and multiple tokens. This indicates that NuSMV is not recommended at present for verifications of timed/untimed UML activity diagrams.

The comparison between the best translation for each tool shows that UPPAAL has the best performance for the verification of timed UML activity diagrams. Although PES shows great potential in the verification of the benchmarking models, the results of the verification of the infusion pump model highlights some problems of the tool. The infusion pump model has a big numbers of OR-Type nodes because of the UML ParameterSets. The number and the position of these nodes increases the number of the possibilities that the model checker has to verify. Therefore, activities in the benchmarks set with similar sizes than the infusion pump model have a smaller verification time. That indicates that the PES requires more optimization efforts for the analysis of multiple paths. UPPAAL achieves the best time for the verification of the case study. However, the difference in the verification time of the infusion pump between UPPAAL and SPIN is smaller than in the benchmark. That indicates that the analysis of multiple paths has a larger effect on UPPAAL than SPIN. However, the results of the preliminary study of OR-Type nodes show that memory consumption of SPIN limits the verification of models with a large number of OR-Type nodes. The presence of OR-Type nodes increases the memory and verification times, in particular with multiple outputs, since the possible paths that have to be checked by the model checker increase. It is worth noting that the evaluated models in the preliminary study exaggerate the usage of OR-Type in activities. Therefore, the results give us a preamble for future work, in which we want to analyze the influence of OR-Type nodes within real UML models.

It was expected that TA-based model checkers would have better performance in the evaluation of timed models. However, the results of the experiments of non-timed models showed that the timing modeling proposed for SPIN does not significantly increase the verification time of timed UML activity diagrams.

Since the presented translations are based on a semantics that addresses embedded systems and extended by the DMOSES-profile, translations have to be tailored in order to verify UML activities used in other domains. This paper presents an insight on how different schemes used in the translation can influence the verification performance, which we believe can be very helpful for the development of automatic generated formal models, in particular, from timed models or models with a token-based semantics.

7. Conclusion

This paper presents and evaluates eight translations from UML activities diagrams into the front-end languages used by the model checkers NuSMV, SPIN, UPPAAL and PES. These translations are based on a UML activity diagram semantics extended for embedded systems using the DMOSES profile. The correctness of the translations is assessed via extensive testing and manual comparison of the results with the UML 2.x standard, since no consensus formal semantics for UML exists. The performance evaluation of the translations was undertaken by in the context of a UML activity benchmark and a case study of an infusion pump. The performance of these translations was determined by measuring the verification time of reachability formulas. The results show that UPPAAL achieves the best performance, followed by SPIN. Furthermore, it can be concluded that NuSMV is not adequate to verify UML activity diagrams due to the rapid explosion of the state space. Since SPIN and NuSMV are not intrinsically able to verify timed models, we introduced timing modeling for these model checkers in order to handle time requirements. For the verification of a single processing unit (no async), the results show that the addition to the timing modeling in SPIN does not affect the verification performance. However, the analysis of the performance for multiple processing units needs further study. Signal processing carried out in the infusion pump is verified using the presented translation. In future work, the translations will be extended in order to add an execution time range, and action behavior in order to analyze in detail data constraints. Furthermore, we plan to introduce the translation for interconnected UML state machines and activities in order to verify the full infusion pump model.

Acknowledgement

The authors would like to thank Peter Fontana for his help with the PES tool.

Appendix A. Building of UML complex semantics based on OR-Type nodes and AND-Type nodes

This section shows how complex behavior of the semantics of UML activity diagrams can be built based on the behavior of OR-Type nodes and AND-Type nodes. Fig. 10 shows these complex behaviors. The graphical notation follows the notation of digital electronics for AND gates for AND-Type behavior and OR gates for OR-Type behavior.

Fig. 10(a) shows how the connection between send nodes and accept nodes can be represented by one OR-Type node, which is connected to all send nodes, and one AND-Type node, which is connected to all accept nodes. Thus, if one send node sends one token through an event all the corresponding accept nodes will receive one token. Parameter Sets allow the execution of an action from different sets of inputs or firing different sets of outputs. Actions with Parameter Sets are specified as an OR-Type node, which is invoked if at least one set has been activated (Fig. 10(c)). Every Parameter Set is
defined as an AND-Type node, therefore, it is active only if all the inputs have a token. The same behavior is assigned for outputs.

After the invocation of an activity, all initial nodes are activated and all incoming tokens of the activated Parameter Set are fired to the connected actions. This behavior is shown in Fig. 10(b). Note that the initialization of the activity is determined by the OR-Type node, which will be invoked if at least one Parameter Set node is active, and as a consequence, all initial nodes are invoked by the AND-Type node. The finalization of the activity is defined by the AND-Type node (Final) shown in Fig. 10(d)). This node is connected the initialization of the activity through an edge with the highest priority, therefore, this edge will be only active after all tokens of the activity have been consumed. After the finalization of the activity, an active Parameter Set will be invoked, followed by firing tokens in the outgoing edges.

References