Electronic Communications of the EASST
Volume 66 (2013)

Proceedings of the Automated Verification of Critical Systems (AVoCS 2013)

Integrating model checking and UML based model-driven development for embedded systems

Zamira Daw, Rance Cleaveland, and Marcus Vetter

15 pages

Guest Editors: Steve Schneider, Helen Treharne
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Integrating model checking and UML based model-driven development for embedded systems

Zamira Daw[1], **Rance Cleaveland**[2], **and Marcus Vetter** [1]

[1] EMB-Lab, Hochschule Mannheim - University of Applied Sciences Mannheim, Germany
[2] Computer Science, University of Maryland, Maryland, USA

Abstract: This paper discusses issues associated with integrating model checkers into a model-based development environment for embedded systems. The environment, DMOSES, is based on a formalization of UML Activity Diagrams and is used to generate correct and efficient code from such models; a key application area is the medical-device domain. A recent effort has focused on introducing formal reasoning into the development flow so that modelers can assess the correctness of their models before generating code from them. The verification of system requirements is shown using a case study of an infusion pump. This paper discusses issues involved in integrating model checkers into DMOSES and reports on a performance evaluation of two model checkers in particular: NuSMV and UPPAAL

Keywords: UML activity diagrams, model checking, UPPAAL, NuSMV, embedded systems

1 Introduction

The importance of software in embedded-system engineering has been growing steeply, as the ongoing decline in the price of microprocessors has made the inclusion of computing capability economically feasible in more and more applications. The migration of control functionality into software has afforded device designers opportunities for richer and more sophisticated functionality, and indeed, in industries such as automotive, aerospace and medical-device, software is integral. At the same time, the production of such software is imposing greater costs, and risks, on such companies. On the cost side, the (increasingly complex and sophisticated) software must be written; on the risk side, software errors can lead to annoyance and, worse, safety issues for users, and to liability and warranty exposure for device manufacturers.

To cope with costs of software production, embedded-systems developers are turning to the use of *model-driven development* (MDD). In MDD approaches, designers first build models of the software to be constructed, then employ synthesis tools to generate the source code for the application automatically. The use of modeling languages such as MATLAB®/Simulink® for MDD is already widespread in industries such as automotive, and the Unified Modeling Language (UML) [24] is also gaining credibility, and adherents, for this purpose in the embedded-system domain, due to its status as a non-proprietary, independently maintained standard. UML includes several graphical sublanguages, a precise abstract syntax given via a metamodel, and a semantics described in prose form. Various enrichments to UML also support the modeling of real-time behavior, which is essential for control-system modeling.

The models constructed during MDD may be seen as system specifications; formally verifying that these models meet system requirements offers an appealing approach to coping with the abovementioned risks associated with embedded software. This observation has stimulated a variety of efforts aimed at integrating *model checking* into MDD methods [25]. Model checking allows a mathematical verification of a system according to a set of requirements. In order to introduce formal verification in MDD processes, one must create or adapt methods that allow the verification of models like UML, which commonly do not have a formal semantics. In this context, special attention has focused on model checking of UML *activity diagrams*. An activity diagram is a graphical representation of processing flows defined by nodes and edges, together with constructs for choice, iteration and concurrency. A node represents a function that takes inputs and converts them into outputs. Several approaches have proposed the use of model checkers such UPPAAL [2], NuSMV [3] and SPIN [18] for the verification of activity diagrams [11, 20, 12, 21, 19, 16]. The verification is typically carried out by a tool chain that is composed of a mapping of the UML models into the mathematically well-defined language supported by the model checker, followed by an analysis of the translated model using the model checker. Such a translation-based approach necessarily ascribes a formal semantics to activity diagrams, and indeed much of the work in the aforementioned papers focuses on these semantic issues. In particular, performance aspects of the tool chains (mapping + model checker) for UML activity diagrams are typically not considered, making the decision about which model checker to use a difficult one for tool-chain builders.

The purpose of this paper is to present the results of a comparative study of the integration of two different model checkers, NuSMV and UPPAAL, into a UML-based MDD framework for embedded software. The MDD method, DMOSES [8], supports the automated generation of real-time control software from activity diagrams extended with information regarding execution time, parallelism and priority. In support of this, the paper formalizes UML 2.x activity diagrams, whose semantics are based on Petri nets, via translations into *timed automata* (UPPAAL) and the NuSMV input language. The performance of both tool chains using the model checkers NuSMV and UPPAAL is evaluated on the verification of a set of benchmark UML activity diagrams; the UPPAAL-based tool chain, which demonstrates significantly better performance on the benchmarks, is then used to analyze the controller of an infusion pump. The remainder of this paper is structured as follows. Section 2 discusses related work for verification of UML models using model checking, while Section 3 reviews the DMOSES method. Section 4 presents the integration approach for automatic verification of DMOSES\UML models, and Section 5 presents the translations from DMOSES activity diagrams into the timed automata and the NuSMV language. Section 6 shows an experimental evaluation of the proposed translations using test models as well as the verification of system requirements by means of an infusion pump. We conclude the paper and present future work in Section 7.

2 Related Work

This section summarizes related work on model checking for UML activity diagrams. It should be noted that the semantic account of these diagrams changed dramatically from UML 1.x [23], which uses state machines as the underlying mathematical model, to UML 2.x [24], in which Petri nets are the foundational theory. Accordingly, the following discussion distinguishes which version of UML the different approaches target.

Eshuis [10, 11] proposes a symbolic model checking approach built on NuSMV for activities defined in UML 1.5 that is based on an adaptation of UML semantics for workflow modeling based on the use of hypergraphs. The mapping from UML to the NuSMV language is based on these hypergraphs. Similar to Eshius, we formally specify activity-diagram behavior by first defining a denotational model for the diagrams (e.g. hypergraphs) and then using a translation into model-checker notations to define the execution semantics. In contrast to Eshius, we use the UML 2.x. Since the Eshius approach is based on state-machine semantics, it cannot be directly applied to UML2.x activity diagrams.

NuSMV has been used for the verification of UML 2.x activity diagrams in other approaches. Lam [20] presents a light mapping of activities into NusMV, in which each model element is represented by a state machine. A similar mapping is presented in Grobelna [12], in which activity diagrams are transformed into basic Petri nets and verified using NuSMV. A state-reduction technique is also proposed to alleviate explosion during the transformation process. SAT-based Bounded Model Checking (BMC) are used to verify hierarchical state machines by encoding the different levels in [9]. This approach is not further considered because the focus is different from our work and we use only the complete model-checking approach of NuSMV instead the SAT-based Bounded Model Checking (BMC) due to our interest in full verification. However, BMC can be added in future works.

Latella [21] proposes the use of SPIN for the verification of UML hierarchical state charts, and this work has been extended for verifying UML 2.x activity diagrams in Jing [19]. Guelfi [16] proposes a PROMELA translation that considers timed execution of UML 2.x model elements. Timers are defined using integer variables in this modeling (the other works do not typically consider time). In contrast to Guelfi, we consider multiple tokens and limited buffers in order to allow the verification of overwriting data, which is a crucial aspect for embedded systems.

Since the UML 2.x activities semantics are based on Petri nets (PN), verification approaches for PNs can be partially used for activities. The SPIN tool is used in Riberio [22] for model checking of embedded systems modeled with synchronous and interpreted Petri nets. Cortes[5] verifies system properties, including timing behavior, described in PRES+ (Petri nets based Representation for Embedded Systems) using HyTech [17]. PRES+ considers execution time and value requirements in its modeling. This tool is not maintained at the moment. Gu [13] analyzes embedded real-time systems modeled by timed petri nets (TPN). The verification is based on a translation from TPN to timed automata and carried out using UPPAAL. This approach models both execution time and multiple tokens. They also studied multiprocessor scheduling analysis using NuSMV [15] and UPPAAL [14]. They concluded that NuSMV has a better performance for this application.

All of the above mentioned approaches only use one model checker, except for Gu [14, 15]. This fact and the different considered features of the models, make it difficult to compare tool chains for model checking UML activity diagrams. However, the literature does show that NuSMV and SPIN are often used for the verification of these diagrams, while in contrast, UPPAAL is used for timed Petri Nets. In this work, we propose the comparison between two tool chains based on NuSMV and UPPAAL for timed extensions of activity diagrams. NuSMV has been chosen because they already showed a detailed description of UML activities in [11] and because its better performance in contrast to UPPAAL for scheduling analysis in [15]. UPPAAL has been chosen for its ability to model time.

3 Model-Driven Development with DMOSES

DMOSES (Deterministic Models for Signal-processing Embedded Systems) is a model-driven development method for embedded systems based on UML behavioral models [8]. This method extends UML semantics in order to ensure precise accounts of embedded-system behavior. These UML/DMOSES models may then be transformed into source code that can be executed on different hardware platforms (MCUs and FPGAs). The development process is supported by an eclipse plug-in (see www.dmoses.org).

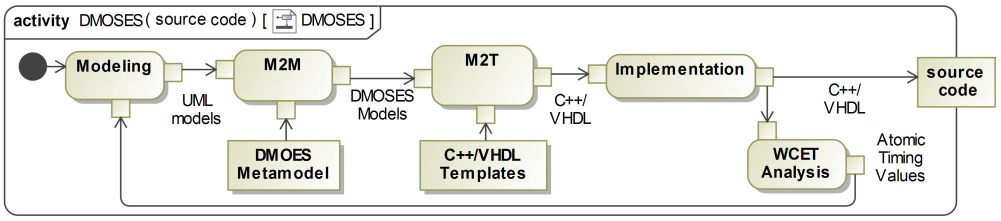


Figure 1: Overview of the DMOSES development method

Figure 1 shows an overview of the process that contains the most relevant steps for this work. Embedded systems are modeled using interconnected UML activities and state machines, which are extended using the DMOSES profile. These models are transformed into source code by a model-to-model transformation (M2M) and a model-to-text transformation (M2T) step. The M2M step translates UML models into DMOSES models that are then transformed into source code in the M2T step. The DMOSES models are intermediate models that combine the extended UML semantics with their implementation. DMOSES models are independent of the hardware platform on which they run. M2T uses different templates for each programming language. The templates are based on frameworks that implements the behavior defined by the models according to the extended UML semantics. UML atomic elements are defined at the code level in the implementation step. Existing atomic elements can be chosen from libraries. New elements can be implemented into wrapper structures automatically created by the code generator. Thus, the level of abstraction of the UML models depends on the functionality of the atomic elements, which can represent anything, from a simple mathematical operation to a complete algorithm. After the implementation step, the source code is ready to be compiled or synthesized. The DMOSES method also integrates timing analysis based on the analysis of atomic element [7]. The atomic analysis is performed by worst-case execution time (WCET) tools that analyze the source code of the atomic elements. Thus, extended UML models are enriched with the real information about the execution time. That allows further analysis about timing requirements relevant for real-time embedded systems.

Extended UML Activity Diagrams UML activity diagrams are composed of nodes and edges. Nodes are divided into *control* nodes and *action* nodes. Control nodes are used to manipulate flow processing (e.g *DecisionNode*), while action nodes represent a specific functionality. Among the action nodes included are: *SendSignalAction*, *AcceptEventAction*, *Action* and *Call-BehaviorActions*. *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 functionality of the action is implemented outside of the model. Activity diagrams may also be composed hierarchically.

UML activity-diagram semantics has been extended by the DMOSES profile to allow the addition of information at the model level using stereotypes such as *asyc* (presence of concurrent processing), *priority*, *WCET* (the maximal execution time of a model element), *platform* (executing hardware platform), etc. The complete semantics of the extended activity is shown in detail in [8]. The extended UML semantics allows modeling the functionality and the execution of the system separately, thereby facilitating the management of concurrent processing. Moreover, this semantics ensures also a unique implementation.

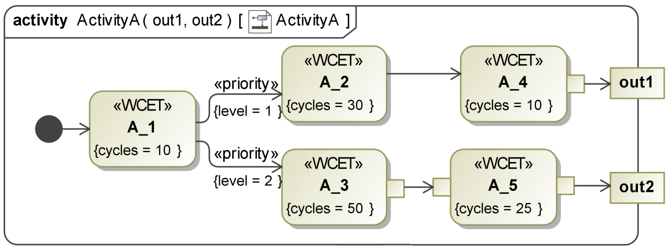

activity ActivityA (out1, out2) [ActivityA]
«WCET» A_1 {cycles = 10 }
«priority» {level = 1 }
«priority» {level = 2 }
«WCET» A_2 {cycles = 30 }
«WCET» A_3 {cycles = 50 }
«WCET» A_4 {cycles = 10 }
«WCET» A_5 {cycles = 25 }
out1
out2

Figure 2: UML activity that defines an execution order (*priority*) and execution times (*WCET*)

The semantics of activity diagrams is token-based. Action nodes need "token" (potentially bearing a value) to be present in some number of its input edges in order to fire; after doing so, the node deposits a (value-bearing) token on some number of its output edges. The number of tokens depends on the type of the node and on their own connections [24]. For example, a *ForkNode* requires a token in all its inputs for its execution, while the merge node requires only one token. The execution of the *ActivityA* (Figure 2) begins with the token firing by *InitialNode* (black circle), as consequence, *A_1* is invoked. After the execution that takes 10 cycles, *A_1* offers two tokens in its outputs. The absence of the stereotype *async* determines that tokens are fired sequentially. Therefore, the first token with highest *priority* is fired, which invokes *A_2*. Note the highest *priority* corresponds to the lowest number. The following edge is fired only after the flow processing of the previous edge is finished. The processing of a flow finishes when it reaches a *FlowFinalNode* or the flow does not have nodes that can be executed due to the lack of tokens. The execution of *A_2* is followed by *A_4*, which sets the out pin *out1* with a data denoted as a square. The execution of the *ActivityA* finishes if and only if all outputs are set. Therefore, the processing returns to the action *A_1* that fires the second token, thereby invoking *A_3*. The execution order of the activity of figure 2 is {*A_1, A_2, A_4, A_3, A_5*}.

4 A Framework for Integrating Model Checking into DMOSES

This section develops a framework for integration of model-checking techniques into DMOSES. The approach relies on the use of an intermediate format, *flow diagrams*, together with a translation of DMOSES models into flow diagrams. Incorporating a model checker into the DMOSES environment then may be done by giving a translation of flow diagrams into the model checker's input notation. The DMOSES process (Figure 1) has been extended by adding the formalization process shown in figure 3.

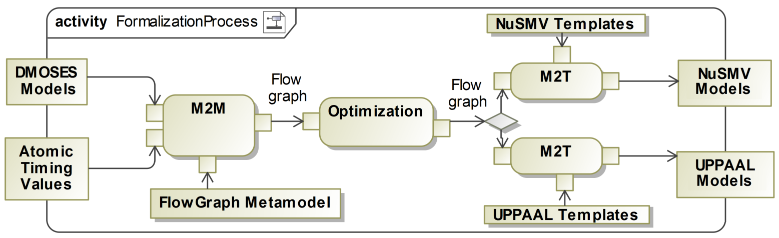


Figure 3: Extension of the DMOSES process for the automatic formalization of UML activities using NuSMV language and UPPAAL language

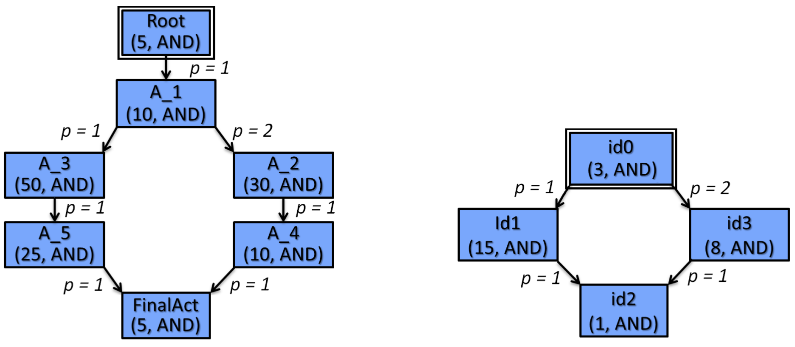
The formalization process uses the following indirect mapping strategy: M2M, Optimization, and M2T. This strategy facilitates the addition of new input models (e.g. UML models) as well as target languages (input languages for model checkers). Intermediate models (i.e. flow graphs) abstract only the relevant information for the formal description, reducing the complexity of the model and facilitating optimizations and further transformations. DMOSES models are transformed into graphs based on the FlowGraph Metamodel (A similar transformation used for timing analysis purposes in the DMOSES method is showed in [7]). The vertices of the graph are enriched with timing information obtained by the analysis of the source code of the atomic elements. 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). The resulting graphs are translated into the corresponding languages required by NuSMV and UPPAAL using M2T transformations. The translation is based on templates, which depend on the description of the UML activity behavior in these languages (Section 5).

Definition 1 A *flow graph* is a tuple $G = (V, E, v_0)$ where:

1. V is a finite non-empty set of *vertices*. A vertex is a pair $v = (r, f)$ where r is a non-negative real number representing the WCET, and $f \in \{OR, AND\}$ is a logic function that determines if at least one edge or all edges are required for the execution.
2. E is a finite non-empty set of tuples of form $e = (v, v', p)$ where v is the source vertex, v' the target vertex and p the priority representing by a positive integer.
3. $v_0 \in V$ is the distinguished *start vertex*.

Flow graphs aim to abstract the processing flow from interconnected UML activities and state machines. The flow graph is a directed graph. Each element of the UML activity is transformed into a vertex or a set of vertices according to the semantics. For instance, vertices corresponding to *SendSignalActions* are connected to the respective *AcceptEventActions*.

After transforming UML activity diagrams into flow graphs, an optimization step is carried out in order to reduce unnecessary states that can lead to a state explosion. The optimization step aims: 1) integration of multiple abstraction levels, 2) reduction of the number of vertices. *CallBehaviorActions* instance a type of functionality described by an activity, thereby allowing multiple levels of abstraction. Vertices corresponding to *CallBehaviorActions* (e.g. *Pump* in figure 8) are substituted by the graph of their type (e.g. activity *Motor_control*). In order to reduce the number of vertices, consecutive actions are grouped into one vertex that integrates intrinsic characteristics (e.g. execution time). Two actions are consecutive if the outputs of the first action activate only the second action and the second action can be only activated by the first action. Furthermore, the cycles assigned to an action are reduced by using the least common multiple of the entire system. Figure 4 shows an example of the transforming process from the activity in 2, into the flow graph in 4a and into the optimized flow graph in 4b.



(a) Flow graph (b) Optimized flow graph

Figure 4: Resulting flow graph and optimized flow graph from the UML activity of figure 2

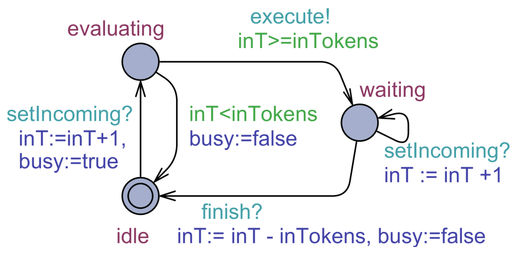
5 Translating Flow Graphs into UPPAAL and NuSMV

In this section, we presented two translations from flow graphs into model-checker languages based on the extended UML activity semantics. An activity diagram describes a system behavior with the help of interconnected elements. Each element represents a functionality that does not depend on the system, on which it is currently used. However, its execution point is directly related to the connections, which is specific to the system. The behavior described by an activity can, therefore, be separated between system specific and nonspecific. These parts of the behavior are called *system* and *token management* respectively. That facilitates the formalization of activity diagrams. The *system* defines the interaction between components and the *token management* specifies the behavior of each component. This interaction is given in the UML diagram by edges and priorities. The translation from UML activity into model-checker languages ensures a total transition relation for elements that have a related execution according to the UML standard. Edges represent a bidirectional relationship between components. The transition from source node to target node is called *forward transition*. It is triggered after finishing the execution of a component. In contrast, the transition from target node to source node is called *backward transition*. It is triggered when the flow processing is finished (see previous section).

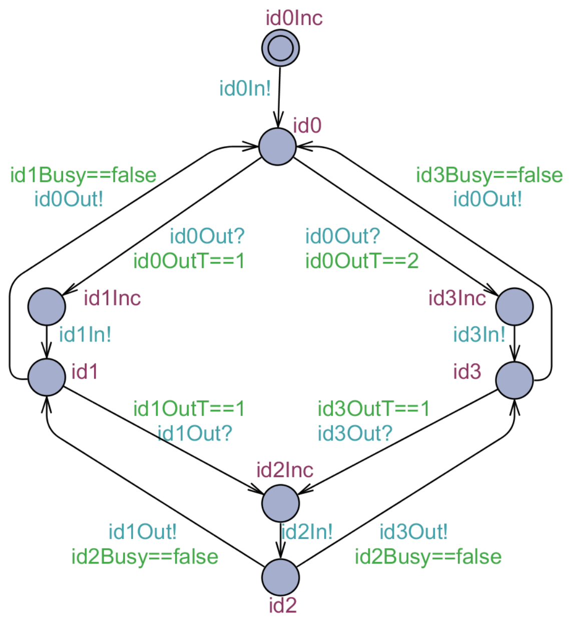
The *token management* specifies the beginning, the end and the result of a component execution. The *token management* is divided into the processes: *incoming*, *calculate* and *outgoing*. *Incoming* specifies the receiving of tokens as well as the beginning of the execution. *Calculate* defines the execution duration in machine cycles, which is given in the model. *Outgoing* specifies the firing process after the execution.

5.1 Timed Automata (TA)

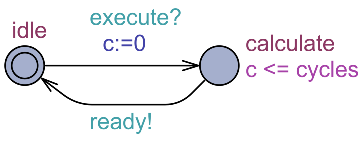
TA was proposed in [1] as a formal language to model the behavior of real-time embedded systems. The UPPAAL model checker can verify safety and liveness properties from systems modeled using TA. In order to transform a UML activity into a TA, the behavior of the *system* is formalized in terms of TA by the *system* TA (Figure 5b) and the *token management* is formalized by three TAs incoming, calculate and outgoing (Figure 5a, 5c, 5d). These three TAs are instantiated for each vertex in the graph.



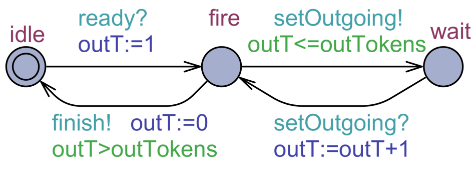
(a) *incoming* TA



(b) *system* TA (corresponding to the flow graph of figure 4b)



(c) *calculate* TA



(d) *outgoing* TA

Figure 5: TA-model for the specification of UML activities. The channels id_nIn of the *system* TA are connected to the channel *setIncoming* of the corresponding *incoming* TA as well as the channels Id_nOut are connected to the channel *setOutgoing* of the corresponding *outgoing* TA.

The *incoming* TA is responsible for receiving tokens through the *setIncoming* channel triggered within the *system* TA. The number of received tokens is saved in the variable *inT*. If this variable reaches the required number of tokens, *inTokens*, the *incoming* TA informs the *calculate* TA via the *execute* channel and changes to the *waiting* state. During this state, incoming tokens can still be received. In case that the required number of tokens for execution is reached again, the *execute* signal is sent once the current execution is finished. The *finish* signal is sent from the *outgoing* TA, which indicates that all outgoing transitions have been fired. The *calculate* TA models the time that a node needs to be executed by using a clock variable. After the execution, the *calculate* TA notifies the *outgoing* TA via the channel *ready*, which changes to the state *fire*. In case that there are tokens to be fired, the *outgoing* TA changes to the state, *wait*. This TA stays in the *wait* state until there is a transition that synchronizes with the channel *setOutgoing*. After firing all tokens, the TA returns to the state *idle* notifying the *incoming* TA that the firing process is finished.

The *system* TA (Figure 5b) is composed of states, which indicate the receiving of tokens and the activation of a node (but not its execution), and transitions, which consist in *forward transitions* (e.g. id0 to id1) and *backward transitions* (e.g. id1 to id0). The state id_nInc indicates only that an incoming has been received and state id_n indicates the activation. The representation of one vertex of the flow graph using two states in the *system* TA is because double synchronization is not allowed in one transition. The double synchronization is carried out by sending and receiving of *setOutgoing* within the *system* TA via channels id_nOut. This synchronization reduces the possible transitions, thereby decreasing the complexity in the model checking.

Forward transitions are divided in two transitions. The first transition (e.g. id0 to id1Inc) synchronizes the *system* TA (channel id_nOut) with the *outgoing* TA (channel *setOutgoing*) of the source state (e.g. id0) according to the given priority, id_nOutT. This priority is determined by the corresponding *outgoing* TA via the variable *outT*. The second transition (e.g. id1Inc to id1) synchronizes the *system* TA (channel id_nIn) with the *incoming* TA (channel *setIncoming*) of target state (e.g. id1). By contrast, *backward transitions* are synchronized with the *setOutgoing* of their target state (e.g. id0). *Backward transitions* are enabled only if the source state (e.g. id1) is finished with firing, which is indicated by the flag, id_nBusy. This flag is set in the *incoming* TA via the variable *busy*.

5.2 NuSMV Language

The NuSMV language allows the modeling of finite state systems using case structures and Boolean operations. Table 1 shows the rules for transformation of a flow graph into the NuSMV language. In order to specify UML activities, *system* and *token management* are described separately.

Table 1: Transformation Rules of flow graphs into NuSMV language

```
1   For every $x \in Vertex$
      VAR                              INIT
        $xInT$: 0..maxTokens           $xInT = \begin{cases} 1 & if v = root \\ 0 & otherwise \end{cases}$
        $xOutT$: 0..maxTokens          $xOutT$ = 0
        $xC$: -1..$xR$ -- "WCET"       $xC$ = -1
2   For every $x \in Vertex$
    DEFINE
      $xReady := (xInT < ReqTokens \mid$
                  $xOutT = xMaxOutgoings + 1)$
3   ASSIGN
      next(system) :=
        case
          system = root : $x0$; --  "Initial state"
    For edge $z \in Egde\ z = (x,y,p)$
          system = $x$ & $xOut = p$: $y$ -- "FT"
          system = $y$ & $yReady$ & $xOut = p$: $x$ -- "BT"
        esac;
4   For every $x \in Vertex$
    ASSIGN
      next($xInT$) :=
        case
         -- "finish"
          system = $x$ & $xOut = 1$ : $xIntT - ReqTokens$
          For edge $z = \{y \in Vertex | z = (y,x)\}$
           -- "receiving tokens"
            system = $z$ & $zOutT = 1$ : $xIntT$ + 1
        esac;
5   For every $x \in Vertex$
    ASSIGN
      next($xC$) :=
        case
          $xC$ = $xR$ : -1 -- "idle"
          system = $x$ & $xC! = 1$ : $xC + 1$ -- "execution"
          -- "start"
          system = $x$ & $xInT \geq ReqTokens$ & $xOutT$: 0
        esac;
6   For every $x \in Vertex$
    ASSIGN
      next($xOutT$) :=
        case
          system = $x$ & $xC = xR$ : 1 -- "start"
          -- "finish"
          system = $x$ & $xOut = xMaxOutgoings + 1$ : 0
          For edge $z = \{y \in Vertex | z = (x,y)\}$
           -- "firing"
            system = $z$ & $zReady$ & $zOutT = xP$: $xP + 1$
        esac;
```

The space of states of the *system* is determined by a state variable, which corresponds to the vertices of the graph plus a root state (Rule 3). Using a case statement, the behavior of the *system* is specified by defining transitions (Rule 3). *Forward transitions* (FT) are triggered according to the priority value p. *Backward transitions* (BT) are triggered at the end of the firing process, which is determined by the Boolean variable *xReady* (Rule 2).

Incoming tokens, outgoings tokens and the execution time are defined as integer variables named *xInT*, *xOutT*, and *xC* respectively. Incoming and outgoing variables are limited and range from 0 to the maximum number of tokens that can be saved by an UML element. Note that the maximum value has to be greater than the required number of tokens for the execution (Rule 1). The range of the clock variable is from -1 to the WCET value given in the model. The NuSMV language only enables changes to the value of variables by using case structures. Hence, a case structure has to be defined for each variable. The *incoming structure* specifies that *xInT* has to be incremented by one after triggering off the corresponding FT and decreased by the number of consumed tokens (*ReqTokens*) after the execution (Rule 4).

Within the *calculate structure*, the execution clock (*xC*) is incremented by one while the *system* is in the respective state (Rule 5). After the clock reaches the WCET value (*xR*), the *calculate structure* switches to an idle state denoted by the value -1. It remains in the idle state until the node is executed again after receiving a new set of tokens. The *outgoing structure* defines which transition has to be fired (Rule 6), which is taken into account in the *system structure*. The first transition of the *outgoing structure*, value 1, is triggered after the clock reaches the WCET value. The following transitions are triggered consecutively once the processing of the current transaction is finished indicated by the *xReady*. The last value triggers the corresponding BT.

6 Experimental Evaluation

6.1 Performance Evaluation

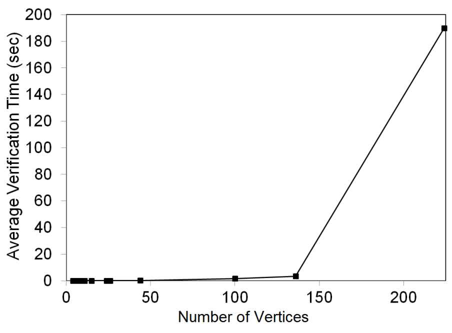
UML activities have been selected as input data to evaluate the performance of the proposed tool chains (mapping + model checker) using UPPAAL and NuSMV. The input of the tools is automatically generated based on the two translations introduced in the section 5. The performance is evaluated by measuring the verification time of deadlock freedom (Formula 1 for UPPAAL and formula 2 for NuSMV). These formulas evaluate whether the final state is reached, which represents the finalization of an activity. An average value is obtained using 10 measures.

$$\mathbf{E} <> id_{final}Cal.calculate \quad (1) \qquad \mathbf{EF}(system = id_{final} \ \& \ id_{final}C = 0) \quad (2)$$

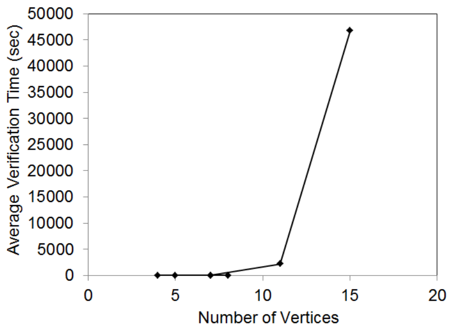
The model checking experiments were run on a Windows 7 PC with a 3.07GHz Intel Core i7 CPU and 6 GB of memory. A set of 30 activities has been created to cover a wide spectrum of model characteristics: number of vertices [1, 224], number of edges [0,290], abstraction levels [1, 5] and with or without deadlocks. This allows the comparison of the performance in relation to the variation of these characteristic (e.g. the performance by increasing the number of actions). The selected activities are created based on test models used for testing the DMOSES method.

The first experiment evaluates the verification performance in relation to the size of the model by increasing of the number of actions and connections. All actions have the same execution time (1 cycle) in order to obtain results that are independent from the real time constraints handling. The difference between UPPAAL and the NuSMV begins to become evident in relatively small activities with only 11 vertices. While UPPAAL requires 0.08 seconds to verify the formula 1, NuSMV requires 2221.77 seconds (Figure 6). The verification time is the determined by the average of the verification (average) time of activities with the same number of vertices. Analyzing the translation from UML into NuSMV, we conclude that the increase in the state space is directly proportional to the number of the incoming and outgoing edges of an action.

This happens because these parameters are represented by integer variables. Each value that these variables can take increases the state space of the FSM.



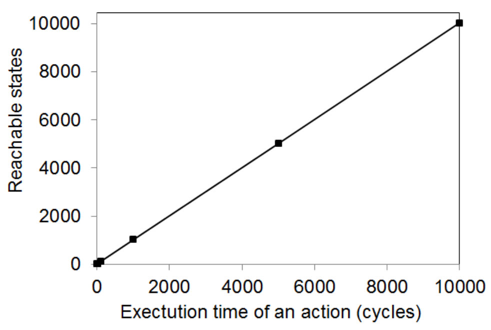
(a) UPPAAL



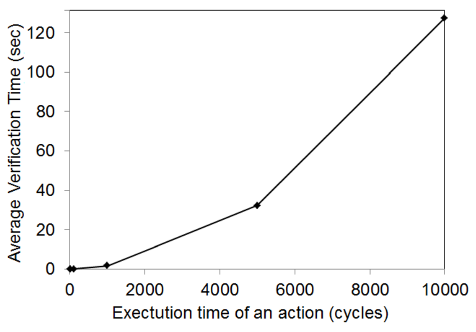
(b) NuSMV

Figure 6: Verification performance of the two presented translations from UML activities

The second experiment evaluates the verification performance in relation to execution time of the actions. The timing of only one action in the model is varied from 1 cycle to 1000 cycles. The relation between the cycles and the state space is depicted in figure 7a. The increase of the cycles influences the state space and, as a consequence, raises the verification time as shows figure 7b. The same experiment was performed using UPPAAL. No changes in the performance are evident, since the execution time is realized in one step that changes the value of the clock from 0 to the maximal time. The above-mentioned conclusion can be generalized for the time execution as well, because the time, incoming and outgoings are represented as integer variables.



(a) Reachable State



(b) Performance

Figure 7: Impact on the state space and performance of NuSMV due to the WCET of one action

The significant difference between the presented approaches is caused by the numerical variables that are needed to encode model information, such as multiple tokens and timing behavior. This information is required to verify data overwriting and timing constraints. NuSMV targets non-numerical symbolic modeling. Therefore, UML activities for embedded systems are outside of the scope of this model checker. In contrast, UPPAAL showed better performance for this specific application domain due to a more suited handling of numerical variables.

6.2 Verification of systems requirements

The verification method proposed in this paper has been used to verify an optical tracking system for navigation aided liver biopsy taking into account medical workflows [6]. In this work, the presented verification approach has been applied to an infusion pump. An infusion pump is used to administer medicaments or nutrients into a patient's circulatory system. This medicament/nutrients infusion is classified as a safety-related system since errors in the functionality of the system can lead to degradation of the patient's health or even his death. The control system of the infusion pump is modeled using UML models and generated by using the DMOSES tool. The design is targeted to an ARM7 processor (LPC2368).

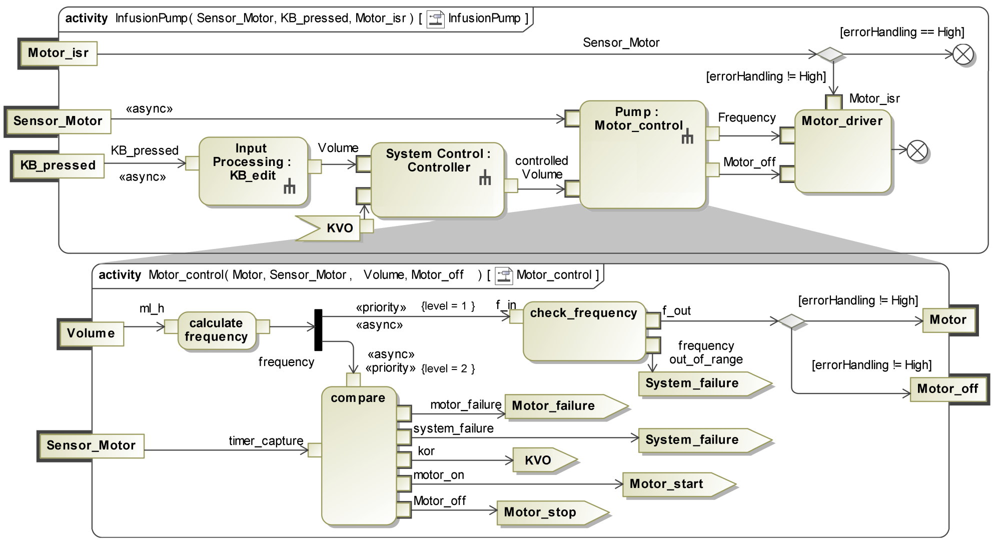


Figure 8: UML models used for the development of an infusion pump using DMOSES

The volume per hour (VH) in mL\h is set by the user using a keyboard. This value is translated into a motor frequency. A display shows the selected value of VH and the currently injected volume. In order to ensure a safe behavior, the system contains additionally sensors that monitor the current behavior of the pump. These sensors measure the velocity of the motor, battery level, and the position of the injection. Using the sensors, the control system can verify if the infusion process is following the specified settings. In case of a failure, the system reacts in a way so that the patient cannot be injured (e.g. alarm, stopping the motor). Figure 8 presents the highest abstraction level on the top and the detailed function of the *Motor_control* on the bottom. The system starts with the user input of the VH. This value is read and verified according to given requirements in *Input Processing*. After the verification, the data is processed by the action *System Control*. This action does not only consider the input data but also the status of the system. The action *Pump* calculates the frequency of the motor and verifies the resulting value against to motor requirements. The *Motor_driver* sets the motor's frequency by writing in the corresponding registers of the processor when the interrupt *Motor_isr* is triggered.

The activity *Infusion Pump* can fire the event KVO (keep vein open) via an accept action. The KVO represents a slow infusion that provides enough fluid flow to keep the end of the catheter from clotting off. In some error cases, the motor has to run in a KVO frequency. The corresponding value for the motor is calculated within the *System Control*. If the send action is processed by the same flow that has begun by the accept action, the event will never be sent. This happens because the accept action is waiting for an event and the send action is waiting for a token. This leads to a deadlock. If both hierarchical levels are not examined together, it is impossible for the developer to identify the deadlock. In larger and more complex system, this task can become very difficult and error prone if done by manual verification. In this system, the KVO event is sent in different points of the system (e.g. in Motor_control). The diagrams present parameter sets that are represented by the marked pins. Parameter sets allows grouping of pins and to execute different functionalities according to the set. For example, the value measured by the motor's sensor is processed by the *Motor_control* without waiting for tokens in the other inputs. Therefore, the parameter sets have a *OR* function assigned to them in the flow graph.

Liveness and safety requirements have been verified for the Infusion Pump example (Figure 8). The flow graph of this system contains 200 vertices and 288 edges. Due to the previous performance results and the size of the model, the system requirements have only been verified using the UPPAAL. The formula 3 shows an example of a safety requirement. This requirement evaluates whether the motor runs with the frequency corresponding to KVO level after the battery reaches the minimal value. The *wait* state belongs to the *outgoing* TA and the *evaluating* state belongs to the *incoming* TA. This formula is true if the execution of the action, id_{batMin}, implies the execution of the action, id_{setKVO}. The average verification time of this formula is 0,7 seconds.

$$\mathbf{E} <> id_{batMin}Out.wait \textbf{ and } id_{setKVO}In.evaluating \tag{3}$$

7 Conclusion

This work presented an approach to integrate model checking into DMOSES, a model-driven development approach for embedded systems. This MDD method has been extended in order to allow automatic verification of UML models and can be downloaded in www.dmoses.org. The translation between these models into model checker's input notation is carried out by an indirect metamodel transformation based on flow graphs. We proposed two ways to specify UML activities by using Timed Automata and the NuSMV languages. In order to choose the most adequate verification tool for the extended UML activity, we realized an empirical study to evaluate the performance of tool chains that use UPPAAL and NuSMV. The results demonstrated that UPPAAL achieves a considerable better performance than NuSMV. This is attributed to the fact that there is a proportional relation between the variables and the state space in the NuSMV translation. Such variables represent execution time and multiple tokens and are required for the verification of embedded systems. Although NuSMV has been used to verify UML activities in several approaches, the presented comparison clearly showed that the limitations of NuSMV significantly compromise the verification performance of these UML models. In order to extend the comparison, some works about the improving the variable's managements of NuSMV such as [4] may be used in future studies. The UPPAAL model checker was used to verify safety and liveness properties in a infusion pump. This device has been developed by using the DMOSES method. In future works, we will extend the proposed formal description to interconnected UML activity and state machines as well as the optimization step to improve the analysis of multiple hierarchical levels. Furthermore, we plan to include the best case execution time (BCET) and WCET in order to evaluate the entire working timing range.

Acknowledgments

The research for this paper was undertaken during a stay as visiting researcher in the Fraunhofer USA Center for Experimental Software Engineering. This research project is supported by European Regional Development Fund (ERDF) and the german state of Baden-Wuerttemberg.

Bibliography

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[2] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on UPPAAL. pages 200–236. Springer, 2004. http://www.uppaal.org/.

[3] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV: 2: An open-source tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002. http://nusmv.fbk.eu/.

[4] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. Hydi: A language for symbolic hybrid systems with discrete interaction. pages 275–278. IEEE Computer Society, 2011.

[5] Luis Alejandro Cortés, Petru Eles, and Zebo Peng. Verification of embedded systems using a petri net based representation. In *Proceedings of the 13th international symposium on System synthesis*, ISSS '00, pages 149–155, Washington, DC, USA, 2000. IEEE Computer Society.

[6] Zamira Daw, Rance Cleaveland, and Marcus Vetter. Formal verification of software-based medical devices considering medical guidelines. In *Computer Assisted Radiology and Surgery 27th International Congress and Exhibition (CARS)*, 2013.

[7] Zamira Daw, Christian Englert, Flor Alvarez, Josef Börcsök, and Marcus Vetter. Model-driven timing analysis and verification for safety-critical embedded systems. In *Proceedings of the 24th Internactional Congress on Condition Monitoring and Diagnostics Engineering Management*, 2011.

[8] Zamira Daw and Marcus Vetter. Deterministic UML Models with Interconnected Activities and State Machines for Embedded Systems. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009*, 2009.

[9] J. Dubrovin and T. Junttila. Symbolic model checking of hierarchical UML state machines. In *Application of Concurrency to System Design, 2008. ACSD 2008. 8th International Conference on*, pages 108–117. IEEE, June 2008.

[10] R. Eshuis and R. Wieringa. Tool support for verifying uml activity diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, July 2004.

[11] Rik Eshuis. Symbolic model checking of UML activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.

[12] Iwona Grobelna, Micha Grobelny, and Marian Adamski. Petri nets and Activity Diagrams in logic controller specification - transformation and verification. *Mixed Design of Integrated Circuits and Systems MIXDES 2010 Proceedings of the 17th International Conference*, pages 607–612, 2010.

[13] Zonghua Gu and Kang G. Shin. An Integrated Approach to Modeling and Analysis of Embedded Real-Time Systems Based on Timed Petri Net. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 350–359, 2003.

[14] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking. In *SEUS*, pages 263–272, 2007.

[15] Nan Guan, Zonghua Gu, Mingsong Lv, Qingxu Deng, and Ge Yu. Schedulability Analysis of Global Fixed-Priority or EDF Multiprocessor Scheduling with Symbolic Model-Checking. In *ISORC*, pages 556–560, 2008.

[16] Nicolas Guelfi and Amel Mammar. A formal semantics of timed activity diagrams and its promela translation. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, pages 283–290, Washington, DC, USA, 2005. IEEE Computer Society.

[17] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[18] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004. http://spinroot.com/spin/whatispin.html.

[19] Li Jing, Li Jinhua, and Zhang Fangning. Model Checking UML Activity Diagrams with SPIN. In *International Conference on Computational Intelligence and Software Engineering (CiSE)*, pages 1–4, 2009.

[20] Vitus S. W. Lam. A formalism for reasoning about UML activity diagrams. *Nordic J. of Computing*, 14(1):43–64, January 2007.

[21] Diego Latella, István Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.

[22] Oscar R. Ribeiro, Joao M. Fernandes, and Luis F. Pinto. Model checking embedded systems with promela. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:378–385, 2005.

[23] Unified Modeling Language (UML). Version 1.5. Object Management Group, 2003.

[24] Unified Modeling Language (UML). Version 2.4. Object Management Group, 2012.

[25] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.