S. Götz, L. Linsbauer, I. Schaefer, A. Wortmann (Hrsg.): Software Engineering 2021 Satellite Events,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2021 1

# Model Checking AMALTHEA with SPIN[1]

Jan Steffen Becker[2]

**Abstract:** Although the AMALTHEA meta-model has emerged in the past years to an industrially adopted exchange format for (also safety-critical) embedded software architectures, model checking has been seldom applied to AMALTHEA models so far. This work presents a simple and extensible translation approach from AMALTHEA to PROMELA models for analysis with the widely used SPIN model checker. This is a first step towards applying standard state-of-the-art model checkers to AMALTHEA.

**Keywords:** AMALTHEA; model checking; timing verification; safety; PROMELA

## 1 Motivation

During many industrial and scientific research projects in the past years, AMALTHEA [Ec20] has established itself as a modeling standard for multi-core hardware/software (HW/SW) systems. It focuses on timing behavior of these systems and is supported as an exchange format by a growing number of industrial and academic tools. Although not bound to the automotive domain, AMALTHEA supports many concepts from the AUTOSAR [Na09] and OSEK [OSEK] standards and is therefore primarily targeted to automotive embedded software.

By their nature, many automotive embedded systems are safety-critical (driver assistance systems, for example). Safety standards such as ISO 26262 require a high demand of verification effort during development of those systems. In this paper, a verification approach for AMALTHEA models is proposed that translates AMALTHEA models into PROMELA code for verification with the state-of-the-art model checker SPIN [Ho97]. In contrast to classical simulation tools, model checkers such as SPIN are able to do a full state space exploration of the input model and therefore proof safety properties (such as task deadlines) formally. Even if a full state space exploration is not possible, experience shows that model checking often finds corner cases that are hard to catch using classical simulation methods.

The SPIN model checker is state-of-the-art for explicit state model checking of concurrent systems. Its input language, PROMELA, is a high-level language that allows a high degree of non-determinism. The key elements of PROMELA are parallel processes that communicate

[1] This work has been funded by the *Federal Ministry of Education and Research* (BMBF) as part of *PANORAMA* (reference no. 01IS18057G).

[2] OFFIS – Institute for Information Technology, Oldenburg, Germany, jan.steffen.becker@offis.de

via message channels and shared variables. As such, it is dedicated to the verification of concurrent systems.

The goal of this paper is to demonstrate a simple approach for model checking AMALTHEA models using SPIN. The focus in the translation is on simplicity and not on performance (in terms of optimizing the PROMELA code for model checking with SPIN). Key elements of the approach are the following:

- It provides a way to model integer time stopwatches in PROMELA, together with a simple but safe approximation for dense time. This is independent from the AMALTHEA translation.
- The approach can be extended for other scheduling strategies that can be provided as a library.
- The translation is template-based, so it can be extended easily to support additional AMALTHEA elements.

This work is structured as follows. It starts with a short introduction of AMALTHEA and PROMELA in Sections 2 and 3. Then, in Section 4, the approach for modeling time is described, with the possibility to stop clocks and with an extension that allows the application of dense time models. The translation of AMALTHEA to PROMELA is finally presented in detail in Section 5 and a few experimental results are reported in Section 6. The paper closes with a short summary of related work in Section 7.

## 2 The AMALTHEA Meta-Model

The AMALTHEA Meta-Model is an emerging standard for the exchange of multi-core HW/SW architectures between analysis, simulation and design tools. It mainly focuses on timing behavior. It covers many concepts that originate from the AUTOSAR and OSEK standards. AMALTHEA is divided into a couple of sub-models that target different parts of the design. In this work, the following are relevant.

**Software model** describes the software units (tasks and runnables) of the system including an abstraction of the control flow.

**Hardware model** defines the layout of the executing hardware environment, with a focus on communication paths.

**OS model** characterizes the operating system (OS), i. e., the task schedulers, synchronization mechanisms, and timing overheads caused by OS activities such as context switches. In this paper, schedulers and OS Events are used from the OS model.

**Stimuli model** contains the activation patterns for tasks. This publication considers time-triggered (i. e. periodic) stimuli only.

**Mapping model** brings all the parts together by describing the allocation from software and data to hardware.

In the context of this work, the most important part is the software model. Its basic elements are so-called *tasks* and *runnables*. A tasks is the smallest unit of software that is managed by a scheduler in the OS and assigned to a core for execution. A task is in most cases a sequence of runnables which in turn can be seen as an equivalent of a function. Tasks and runnables are modeled as so-called *activity graphs* in AMALTHEA, whereby a single *item* in this graph is an abstraction of low level processor instructions, such as memory access, computations, or synchronization primitives. The relevant parts of the software model are described in more detail during the translation in Section 5. A complete documentation of AMALTHEA can be found online[3].

## 3 PROMELA and the SPIN model checker

This section gives a short overview of the PROMELA syntax and semantics used in this paper. For a more comprehensive definition the reader is referred to the SPIN manual [Ho04] or the extensive online documentation[4].

The primary elements of PROMELA are *processes*. A process is declared using the keyword **`proctype`**. Active processes (marked with the keyword **`active`**) are instantiated automatically, other processes must be instantiated explicitly from within other processes.

The process body is a sequence (or, more precisely, a graph) of statements separated by `;` or `->`[5]. The syntax is a mixture of C and Dijkstra's *guarded commands*. A statement is *enabled* if it can be executed in the current state, otherwise it *blocks*. In PROMELA, Boolean expressions are statements that do not have a side-effect and are enabled whenever they evaluate to **`true`**. Branching is possible using **`if`** and **`do`** blocks. A **`do`** block is repeated until a **`break`** statement is reached. A branch in an **`if`** or **`do`** block is enabled whenever its first statement, called its *guard* is enabled. If more than one branch is enabled, one is selected non-deterministically; if none is enabled, the whole block blocks.

Normally, in each step one non-blocked process is selected non-deterministically to execute one statement. A sequence of statements can be made atomic by surrounding them with an **`atomic`** block. However, atomicity is lost, i. e., another process can gain control, when a statement blocks, and resumed when the blocking statement becomes enabled (which does

[3] `https://www.eclipse.org/app4mc/help/app4mc-1.0.0/index.html`

[4] `http://spinroot.com/spin/Man/promela.html`

[5] As a convention, `->` is used after the first statement in an `if` or `do` branch. Beware that the `->` is overloaded in PROMELA and also used as part of an inline if-then-else.

not mean that the process gains control immediately). An **atomic** block is enabled whenever its first statement, the guard, is enabled.

Processes communicate via shared variables and message channels. In this work, only a special type of channel is used, so-called *rendezvous channels*. In a rendezvous, a message is passed between two processes if the send and receive statements are executed at the same time. Otherwise, send and receive statements block until the rendezvous can be executed. The advantage of rendezvous channels via buffered communication is that no message queue needs to be stored in the state vector, so they are very cheap during verification. Messages are sent using the `!` and received using the `?` operator. A message consists of multiple fields separated by commas on the right-hand side of the operator. By convention, the first field contains the message type, and the remaining fields message parameters.

Special statements used in this work are the **skip** statement that exists for syntactic reasons and does nothing, the **else** statement that is only valid as a guard in an **if** or **do** branch and enabled when no other branch is enabled, and the **timeout** statement that is enabled only if no other process can execute.

## 4 Modeling Time in PROMELA

By design, PROMELA does not have a notion of time. In the following, it is described how time progress can be explicitly modeled, though. Although many approaches for modeling clocks in PROMELA have been proposed [BD98; MMJ11; TC96; Tr07], none fits all the needs of this paper: Because we need to model preemptive processes with non-deterministic execution times, we need a `wait` function that suspends the calling proces for a non-deterministic amount of time drawn from some interval [`CMIN`, `CMAX`]. Furthermore, time is only accounted when an enablement condition `COND` evaluates to `true`.

The main idea is to use a global time increment variable `delta` and PROMELA's `timeout` statement. Processes that want to wait for an amount of time, communicate the minimum time delay until their next action to a *timing supervisor* process. This process uses the **timeout** statement to initiate and coordinate a time increment whenever no other (untimed) progress is possible in the system. Listings 1 and 2 show the PROMELA code for the waiting process and the timing supervisor. Their interaction is visualized in Figure 1. The `clk` variable ist used to track elapsed time. The Boolean `step` variable in lines 6 and 9 of Listing 1 works as a synchronization barrier. Initially, `step` is 1. The expression `step == 0` cannot be passed unless the timing supervisor process initiates a time step by setting `step` to 0. When this happens, all the processes that are in a waiting state start executing again and adjust `delta_min` and `delta_max` in order to agree on the minimal and maximal delay until the next action. Afterwards, the supervisor process selects a time increment `delta` and resets `step` to 1, which starts a new discrete step. In the proposed implementation, one of $\{\delta_{\min}, \delta_{\min} + 1, \delta_{\max}\}$ is selected (provided $\delta_{\min} < \delta_{\max}$), which allows reaching all possible time increments as well as to "shortcut" larger ones in order to find safety violations earlier.

Waiting processes stay in a loop for multiple time steps, whereby they participate in the election process and increment their clock only if the COND constraint evaluates to `true`. This allows later on in Section 5 to disable time progress for preempted tasks.

```
1  inline wait(CMIN, CMAX, COND) {
2    atomic {
3      clk = 0;
4      do
5      :: clk >= CMIN -> COND; break
6      :: clk < CMAX  -> COND && step == 0;
7         delta_min = MIN(delta_min, MAX(CMIN - clk,1);
8         delta_max = MIN(delta_max, CMAX - clk);
9         step == 1; clk = clk + delta
10     od
11   }
12 }
```

List. 1: Waiting for an amount [CMIN, CMAX] of time.

```
1  do
2  :: timeout -> d_step {
3       delta_min = MAX_TIME; delta_max = MAX_TIME;
4       step = 0 };
5     timeout;
6     atomic {
7       if
8       :: true -> delta = delta_min
9       :: delta_min < delta_max -> delta = delta_max
10      :: delta_min < delta_max -> delta = delta_min + 1
11      fi ; send_event(TIME_STEP,delta); step = 1;
12    }
13 od
```

List. 2: Timing supervisor

It has been shown by [Fe07] that many problems, including safety properties such as satisfaction of deadlines, are undecidable for preemptive scheduling in dense time. As a consequence, there is no exact encoding technique for dense time clocks that guarantees a finite state space. Hence, classical abstraction techniques such as region automata [AD94] or difference bound matrices (DBMs) that are exact for timed automata, can be adopted to preemptive scheduling only in some special cases [Fe07] (e. g. deterministic response times that usually do not apply to AMALTHEA) or for the price of (theoretically unbounded) over-approximations [CL00]. The approach in this work is to provide a simple but safe integer-time over-approximation for the dense time clock values. Because clock values are only compared to integer constants, it is sufficient to store the integer part of the clock values in the state space, provided that for any reachable state $(s, \nu)$ with real-valued clock vector $\nu_{\mathbb{R}}$, the state $(s, \lfloor \nu \rfloor)$ is reachable in the discretized approximation. The approximation is based on the following observation: Assume that in some dense time run, a clock $c$

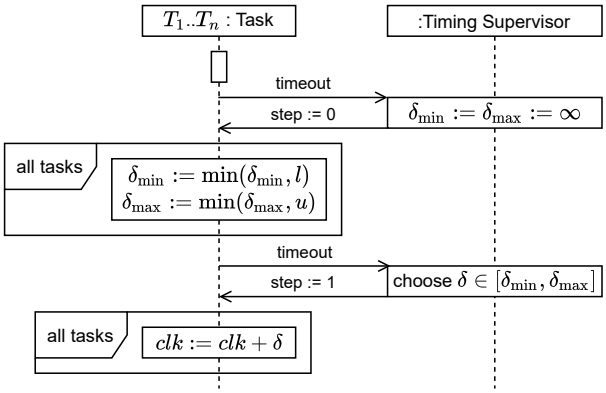


Fig. 1: Interaction of tasks with timing supervisor

is resumed in some step $i$ and runs until being stopped in some step $j > i$. Let's denote the values of $c$ in steps $i$ and $j$ by $c_i \leq c_j \in \mathbb{R}_{\geq 0}$, and the time stamps of these steps by $t_i \leq t_j \in \mathbb{R}_{\geq 0}$. It holds $c_j - c_i = t_j - t_i$ which implies

$$\lfloor c_j \rfloor = \lfloor c_i \rfloor + \lfloor t_j \rfloor - \lfloor t_i \rfloor + e \text{ with } e \in \{-1, 0, 1\} \tag{1}$$

Figure 2 shows a fictive evolution of some clock $c$ that contains all three cases for $e$. Values of $c$ are displayed at begin and end of each interval where $c$ is running, indicated by gray bars. Note, that the number of task preemptions within some time interval is usually bounded, which in turn limits how the error accumulates over time. Choosing a smaller step size reduces the error.

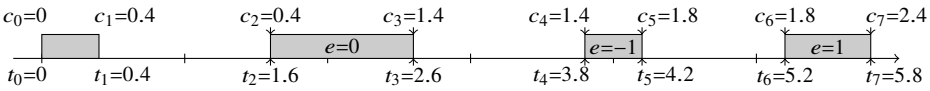


Fig. 2: Illustrating example for Equation (1)

In a dense time interpretation of the PROMELA model, `delta` = $\lfloor t' \rfloor - \lfloor t \rfloor$ contains the difference between the integer parts of the current and last time points, and the `clk` variable tracks the integer part of a dense time clock. Listing 3 extends the `wait` function from Listing 1 and adds adjustments to `clk` for the different possibilities for the error $e$. This is done by tracking (in the `suspended` variable) whether the clock was suspended during the last time step, and, in that case, allow alternative time increments $clk := clk + \delta \pm 1$.

## 5 Encoding of AMALTHEA

In this section, the translation of an AMALTHEA model into PROMELA is described. Additionally to the timing supervisor which is part of the initial PROMELA process, a

```
1  inline wait(CMIN, CMAX, COND) {
2    atomic {
3      clk = 0; suspended = false;
4      do
5      :: clk >= CMIN -> COND; break
6      :: clk < CMAX  ->
7         if
8         :: COND && step == 0 ->
9            delta_min = MIN(delta_min, MAX(CMIN - clk, 0));
10           delta_max = MIN(delta_max, CMAX - clk);
11           step == 1;
12           if
13           :: true -> clk = clk + delta
14           :: suspended && clk + delta > 0 -> clk = clk + delta - 1
15           :: suspended -> clk = clk + delta + 1
16           fi; suspended = false
17        :: !COND && step == 0 && !suspended -> suspended = true
18        fi
19     od
20   }
21 }
```

List. 3: Waiting for an amount [CMIN, CMAX] of time, with dense time adjustment

process is created for each task, each scheduler, each time-triggered stimulus, and each timing constraint. These processes communicate via *rendezvous channels*.

AMALTHEA elements that store state information are encoded as global variables. The PROMELA encoding uses global variables for mode labels, OS events, task states, and activation counters.

**Mode labels** are used in so-called *mode switches* to model operation-mode dependent control flow and mode changes. Mode labels can be shared by multiple tasks. For each mode label $m$, the translation introduces a global variable $\texttt{var}_m$.

**OS events** are synchronization primitives belonging to the operating system. OS events are global Boolean variables and are encoded in PROMELA as such. However, the handling is special: The owner task of an event can *wait* for one or a set of events to be set. In AMALTHEA, this may either be passive waiting, i. e., the task switches to waiting state and gets activated once the event is set, or active waiting meaning the task stays in running state while waiting. Other tasks may set an OS event to inform the owner about some event, e. g., some data now being ready for processing.

**Task execution states** are stored in a global array named `states`. We assign to every task $T$ some index $ID_T$ in this array. The advantage of an array over one state variable per task is that scheduler implementations in PROMELA can be easily reused: The task

IDs (i. e. indices in the state array) are sent to a scheduler via its $\texttt{comm}_S$ channel with an activation request. So there is no need to hardcode execution state variables in the scheduler implementation.

The prototype implements the extended OSEK task state automaton [OSEK] shown in Figure 3. The original state automaton defined by AMALTHEA is an extension of it that introduces variants of the *running* and *preempted* states that are entered when the task is actively waiting, or preempted during active waiting, respectively. These task states are purely conceptional, however, so it is possible to use the simpler OSEK task automaton without changing analysis results. The currently executing task is in the *running* state, and tasks that are ready for execution are in the *ready* state. Initially, and after termination (until the next activation), tasks are in the *suspended* state. In the PROMELA translation proposed in this work, the task process can send *activate*, *wait*, *release*, and *terminate* messages to the scheduler process, which acknowledges them by changing the task state accordingly. At the same time, the scheduler can start or preempt other tasks, according to the scheduling strategy. In a fixed-priority preemptive scheduling, for example, it will always put the highest priority task to *running*.

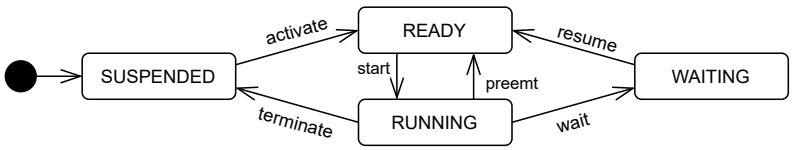


Fig. 3: Extended OSEK task state automaton

Listing 4 shows the PROMELA code frame for tasks. The variable $\texttt{trigg}_T$ counts pending activations for task $T$. For tasks with a periodic stimulus, for example, this counter is incremented by a stimulus process as shown in Listing 5. The task frame is an infinite loop. At start of the loop, the process waits for some pending activation. When there is a pending activation, the task process sends a message with an activation request, its ID and scheduling parameters (the task priority $p_T$ in this case) to its scheduler $S$, via the channel $\texttt{comm}_S$. Then, the instructions (*activity graph items* in AMALTHEA) of the task are processed. Every item in the instructions sequence is an atomic block that is guarded by the condition $\texttt{states[}ID_T\texttt{]==RUNNING}$, so they are only processed when the task has been put into *running* state by the scheduler. At the end of the loop, a *termination* signal is sent to the scheduler.

In the following, the translations of individual *activity graph items* are described. First of all, calls to other Runnables are inlined during translation.

**Ticks** are the primary element in AMALTHEA to model execution time. Ticks are an abstraction of some arbitrary instruction sequence that is characterized by an lower ($l$) and upper ($u$) execution time bound. Ticks are translated as $\texttt{wait(}l\texttt{, }u\texttt{, states[}ID_T\texttt{]==RUNNING)}$. Here we see the need for the `COND` parameter in the `wait` function: Time is only consumed when the task is in *running* state.

```
1  active proctype task_T () {
2    do
3    :: atomic {trigg_T > 0 -> trigg_T--;
4         comm_S!ACTIVATE,ID_T,p_T };
5       /* instructions sequence */
6       atomic {states[ID_T] == RUNNING;
7           comm_S!TERMINATE,ID_T,0 }
8    od
9  }
```

List. 4: Task process frame

```
1  active proctype stim_St() {
2    int clk, t_next;
3    atomic { t_next = O;
4      do
5      :: wait(t_next, t_next, true) ;
6         t_next = D;
7         /* for each stimulated task T: */
8         trigg_T = trigg_T + 1
9      od }
10 }
```

List. 5: PROMELA process for a simple periodic stimulus *St* with recurrence (i. e. period) *R* and offset *O*.

**Switches** are used to model control flow. In AMALTHEA, there are two types of switches: In a *mode switche* each branch is guarded by a Boolean condition over the current values of mode labels. In a *probability switch* each branch is taken with some fixed probability, independent from the system state. A switch is encoded as an `if` block

```
if :: states[ID_T]==RUNNING && c_1 -> /* instructions of branch 1 */
   :: states[ID_T]==RUNNING && c_2 -> /* instructions of branch 2 */
   /* ... */
   :: states[ID_T]==RUNNING && c_n -> /* instructions of branch n */ fi
```

For mode switches, $c_i$ is the enabling condition for the $i$th branch, conjunct with $\neg c_1 \wedge \cdots \wedge \neg c_{i-1}$ to make the branches prioritized. For probability switches, any branch is possible, so set $c_1 := \cdots := c_n :=$ `true`.

**Label access** is not part of the PROMELA code in general, except for write access to mode labels, which is translated as `atomic { states[`$ID_T$`]==RUNNING; var`$_m$ `=` $v$ `}`, where $m$ is the label written, and $v$ the new value (AMALTHEA uses symbolic constants to represent values, which are mapped to integers during the translation). Other label access is not translated, because AMALTHEA does not model data values and therefore it does not affect the state space. Of course it is possible to consider the communication overhead in the translation, though, by inserting appropriate `wait` statements at the point of label access. Possible approaches for calculating safe bounds on the communication overhead are presented in [Hö19; Kr19]. Simulating communication overhead (e. g. in form of bus transactions) is outside of the scope of this publication.

**OS events** are handled as follows: Setting and re-setting OS events is translated the same as mode label write access above, where the Boolean variable `var`$_e$ for event $e$ is set to `true` in case of setting events, and to `false` in case of a reset. Active waiting for some event $e$, i. e., the task is continuously running while waiting, can be translated very easily to `states[`$ID_T$`]==RUNNING && var`$_e$. Because in PROMELA conditions block a

process until they evaluate to **true**, this statements causes the task process to wait for the event. Passive waiting is a bit more complex:

```
1    if :: states[ID_T]==RUNNING && var_e -> skip
2       :: atomic { states[ID_T]==RUNNING && ! var_e ->
3            comm_S!WAIT,0;
4            states[ID_T]==WAITING && var_e;
5            comm_S!RELEASE,p_T } fi
```

For safety properties, such as event chains and worst-case response times, special processes are generated that act as observers. These cannot be implemented outside the simulation model (as so-called *never claims*) as usual in PROMELA, because otherwise it is not possible in PROMELA to use helper variables for tracking elapsed time. So the observers are regular active processes. For sending events to the observers, a macro `send_event(evt, data)` is generated that logs[6] the event and sends it via rendezvouz channels to the observers. The observers handle events in an atomic sequence and acknowledge them via another rendezvouz channel. This passes control back to the process that sends the event without the possibility of being interrupted by other processes. A special message `TIME_STEP` is used by the timing supervisor to inform the observers about time increments.

For reason of space, it is not possible to show scheduler and observer implementations here. The interested reader can find examples in the source code available online (see next section).

## 6 Evaluation

In order to evaluate the approach, the translation has been applied to three example AMALTHEA models: The democar and state machine examples shipped with the APP4MC platform, and the solution to the 2019 FMTV verification challenge provided by Krawczyk et al. in [Kr19]. The AMALTHEA and PROMELA files can be found online[7]. All models have been generated with a step size of $100\mu s$, using the dense time approximation.

- SPIN is able to do a full state-space exploration of the state machine example. It has 1158 states and 1842 transitions.
- In the democar example, SPIN finds the deadline violation for `Task_10ms`. Without observers, a full state space exploration results in about $27 \times 10^6$ states and $65 \times 10^6$ transitions.
- Because a PROMELA implementation of the time-slice scheduling used in the FMTV challenge for the GPU tasks is future work, a fixed priority preemptive scheduling

[6] The prototype implementation inserts `printf` statements into the code that are emitted by SPIN when replaying a counterexample trail. The output can be used to produce an execution trace of the AMALTHEA model.

[7] `https://gitlab.com/jansbecker/ase2021`

has been used also for the SFM and Localization tasks in the model. When assigning the same priority to both tasks, a deadline violation for the SFM task is found after 15681 steps (using depth-first search).

Within the experiments, no false-positive deadline violations due to the over-approximation of dense time have been reported. So it seems that the price of over-approximation is low.

## 7 Conclusion and Related Work

In this work, a translation from AMALTHEA to PROMELA is presented that allows verification of safety properties of AMALTHEA software models using the SPIN model checker. The translation is surprisingly straight forward after making a few building blocks, such as stoppable clocks, available in PROMELA.

Most verification tools for AMALTHEA use simulation and static analysis approaches. The most famous industrial tools of this kind are probably the TA Tool Suite[8] by VECTOR and the INCHRON Tool Suite[9]. Besides these, a number of verification techniques have been applied to AMALTHEA models in the context of case studies, e. g., [Co16; Hö19; Kr19]. Among those, [St16] is the only one known to the author that applies explicit state model checking to AMALTHEA. Compared to the PROMELA translation presented here, the model checker used in that work, RTA$_{\text{NA}_2}$, is specialized to the analysis of task networks.

Timing extensions to the SPIN model checker, respectively approaches to model time in PROMELA, have been published already some years ago. The best known ones are probably [BD98; TC96], some more recent ones can be found in [MMJ11; Tr07]. Among these, only [Tr07] handles stopwatches, which is essential for modeling preemptive scheduling. Compared to the cited work, the approach developed here is less simplistic due to the combination of clock suspension, variable step sizes and a dense time adjustment. Nevertheless, it is worth seeking for a more efficient implementation that needs fewer process interactions per time step. Also, further research investigates into other approximations for dense time that hopefully guarantee bounded errors.

## References

[AD94] Alur, R.; Dill, D. L.: A theory of timed automata. Theoretical computer science 126/2, pp. 183–235, 1994.

[BD98] Bošnački, D.; Dams, D.: Integrating real time into Spin: A prototype implementation. In: Formal Description Techniques and Protocol Specification, Testing and Verification. Springer, pp. 423–438, 1998.

[8] https://www.vector.com/de/de/produkte/produkte-a-z/software/ta-tool-suite/

[9] https://www.inchron.com/tool-suite/

[CL00] Cassez, F.; Larsen, K.: The impressive power of stopwatches. In: International Conference on Concurrency Theory. Springer, pp. 138–152, 2000.

[Co16] Concepcin, J. R.; Gutirrez, J.; Medina, J.; Harbour, M.: Calculating latencies in an engine management system using response time analysis with mast. In: 7th WATERS, FMTV Challenge. 2016.

[Ec20] Eclipse Foundation, Inc.: Eclipse APP4MC, `https://www.eclipse.org/app4mc/`, accessed Dec. 15, 2020.

[Fe07] Fersman, E.; Krcal, P.; Pettersson, P.; Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205/8, pp. 1149–1172, 2007.

[Ho04] Holzmann, G. J.: The spin model checker: primer and reference manual. Addison-Wesley, 2004.

[Hö19] Höttger, R.; Ki, J.; Bui, T. B.; Igel, B.; Spinczyk, O.: CPU-GPU Response Time and Mapping Analysis for High-Performance Automotive Systems. In: 10th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS). June 2019.

[Ho97] Holzmann, G. J.: The model checker SPIN. IEEE Transactions on software engineering 23/5, pp. 279–295, 1997.

[Kr19] Krawczyk, L.; Bazzal, M.; Govindarajan, R. P.; Wolff, C.: Model-based Timing Analysis and Deployment Optimization for Heterogeneous Multi-Core Systems using Eclipse APP4MC. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, pp. 44–53, 2019.

[MMJ11] Marquet, K.; Moy, M.; Jeannet, B.: Efficient encoding of SystemC/TLM in Promela. In: DATICS-IMECS. 2011.

[Na09] Naumann, N.: Autosar runtime environment and virtual function bus. Hasso-Plattner-Institut, Tech. Rep 38/, 2009.

[OSEK] OSEK/VDX Operating System Specification, 2.2.3, OSEK Group, Feb. 2015.

[St16] Stierand, I.; Reinkemeier, P.; Gerwinn, S.; Peikenkamp, T.: Computational Analysis of Complex Real-Time Systems: FMTV 2016 Verification Challenge. In: 7th WATERS FMTV Challenge. 2016.

[TC96] Tripakis, S.; Courcoubetis, C.: Extending Promela and Spin for real time. In: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 329–348, 1996.

[Tr07] Traulsen, C.; Cornet, J.; Moy, M.; Maraninchi, F.: A SystemC/TLM semantics in Promela and its possible applications. In: International SPIN Workshop on Model Checking of Software. Springer, pp. 204–222, 2007.