# Modeling and Verifying Real-time Properties of Reactive Systems

Fenglin Han, Peter Herrmann, Hien Le
Norwegian University of Science and Technology
Department of Telematics, Trondheim, Norway
Email: {sih|herrmann|hiennam}@item.ntnu.no

***Abstract*—SPACE is a model-driven engineering technique for reactive distributed systems. It enables to develop system models from reusable building blocks, formal analysis by model checking as well as automated transformation to executable code. In this paper, we describe an extension of the SPACE formalism which allows to model and verify also real-time behavior. In particular, one specifies real-time constraints in the interface descriptions of the building blocks, so-called Real-Time External State-Machines (*RTESMs*). The RTESMs are translated to guards, clocks and invariants of Timed Automata which can be analyzed by means of the model checker UPPAAL. The approach is explained by a component protecting an electrical motor controller system against overspeed. In particular, we prove that by keeping certain maximum response times, this system guarantees that the speed of the motor stays within certain limits.**

## I. Introduction

Model-based engineering is considered as practical to create high-quality distributed software since it enables stepwise development with varying degrees of abstraction as well as simulation, verification and evaluation. This is of particular importance for the design and verification of real-time embedded software used in safety critical systems. For instance, Buttazzo [1] claims that real-time systems are more vulnerable than other kinds of systems and names three aspects facilitating the design of high quality software: The design of software before building it, complexity reduction, and the enforcement of system compatibility within the design.

The engineering technique SPACE [2] and its tool-set Arctis [3] make a model-driven development process supporting these three issues possible. System behavior is modeled by UML 2 activities (see [4]) that, in a Petri net-like fashion, express behavior by tokens flowing via the edges of a graph [5]. The approach is scalable since an activity may contain *building blocks* that each represents an own activity. The tokens may jump between the two activities, to which a building block refers. The building blocks support also the reuse of submodels since they allow to model recurrent behavior once and to use the resulting building blocks later in various application models. As discussed in [6], in average 70% of a system model can be developed by reusing blocks from the Arctis libraries. The easy reuse of building blocks is supported by External State Machines (ESMs) [6] which describe the interface behavior of the building blocks. Thus, a user who wants to integrate a building block into a system model, does not need to understand its internal behavior, i.e., the activity it refers to, but only its ESM. Further, the ESMs are an effective means to mitigate the state space explosion problem which may occur when model checking that a system model fulfills certain properties [3]. They make it possible to replace most of the activities by the more abstract ESMs in the model checker runs which reduces the state space to be checked effectively. Besides analysis by model checking, Arctis allows to automatically translate a system model into executable code. Currently, it supports the generation of Java code but extended versions for C resp. C++ are under development.

Our previous work has been centered on modeling functionally correct applications without considering real-time properties. In particular, we have used a self-developed model checker to verify basic functional properties (e.g., whether a building block complies with its ESM) while we apply the model checker TLC [7], which is based on Lamport's Temporal Logic of Actions [8], for more complex proofs (see [9]).

In this paper, we discuss how the advantages of a modular specification and verification in SPACE can be exploited for the development of real-time systems. In particular, we introduce an extension of the ESMs to so-called Real-Time ESMs (RTESMs) that make the specification of certain real-time constraints possible. This allows to specify the real-time constraints, a building block requires from its environment as well as the real-time properties guaranteed by itself. Moreover, we added UPPAAL [10] to the set of model checkers supported by Arctis and realized an automatic transformation of each building block and RTESM into timed automata [11] that models relevant time constraints like state invariants, transition guards and clock updates. The real-time properties stated in the RTESMs are translated into formulas of the temporal logic *Timed Computation Tree Logic* (TCTL) [12]. Together the building blocks of a system model form a network of timed automata that can be analyzed by UPPAAL for meeting the TCTL formulas.

The article is arranged as follows: Section II presents the SPACE method. For that, we introduce a building block of a real-life embedded system. In Sect. III, we sketch the Timed Automata and their verification with UPPAAL. Further, the RTESMs are presented. Section IV discusses the compositional character of the real-time behavior verification and presents the results of proving the real-time aspects in our example. We verify in Sect. V that the compositional model checks are sufficient to guarantee that the real-time properties are also met by the overall system. The paper is completed by references to related work in Sect. VI followed by some concluding remarks.

## II. Arctis Building Block Model

In this section, we introduce the model-based development approach using SPACE and Arctis by showing a component

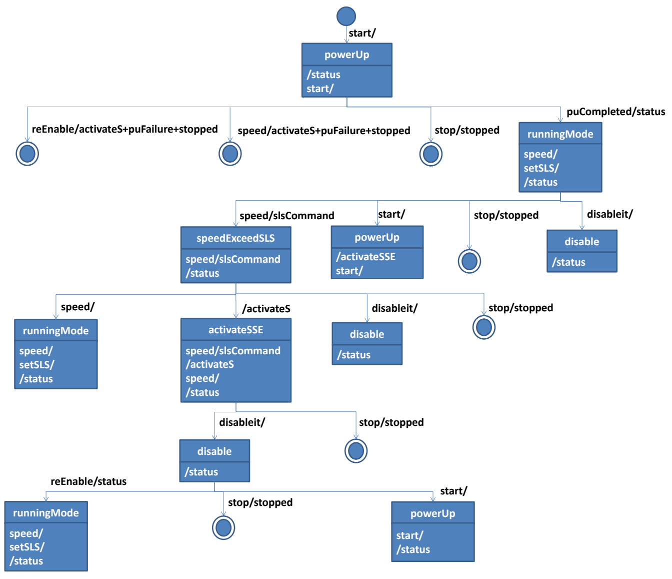

Fig. 1. The External State Machine (ESM) of the SLS block

of a control system for electrical motors which has been developed by Asea Brown Boveri, Ltd. (ABB).

The modeling and verification includes a list of functional units and test cases of a motor controller system, which cooperate with each other realizing the start of the control software. We ignore the big view of system collaboration and concentrate on the Safety Limited Speed System (SLS) component. The $SLS$ complies with the safety standard IEC 61800-5-2 [13] in order to guarantee that the speed of a motor stays below a configurable maximum limit. To achieve that, this component is able to reduce the motor speed or even initiate the stopping of the motor if necessary.

## A. *Interface of the SLS Building Block*

To understand the functionality of the SLS block, we look first on its External State Machine (ESM) which is depicted in Fig. 1.[1] The ESM reflects nicely that, according to standard IEC 61800-5-2, the SLS has six different control states. Thereby, the starting (●) and termination (◉) nodes of the ESM refer to the state *idle* while the other states are represented by vertices containing the corresponding state names:

- *idle:* The motor control system is switched off.
- *powerUp:* The motor control system is starting.
- *runningMode:* The motor is running normally, i.e., it is below its maximum speed limit.
- *speedExceedSLS:* The motor runs above its permitted speed limit but did not yet exceed the maximum time period after which it has to be shut down by executing the Safe Stop Emergency (SSE) handler, another system component, that manages the execution of emergency stops of the motor.
- *activateSSE:* The SSE handler was triggered and the motor was shut down.
- *disable:* The SLS block is disabled after the power for the motor was removed and it cannot produce any torque again.

The transfer of tokens between the activity of a building block and its environment is modeled by so-called pins. That are syntactical constructs used in both the activity modeling the behavior of the block and the one using it. The ESM of a building block models the interface behavior of a block by linking each ESM transition to a number the block's pins. The activity to which the block refers, may only carry out a sequence of token flows if the pins passed in this sequence are exactly those linked to an ESM transition that is executable in the current ESM state. For example, the initial transition from state *idle* to state *powerUp* must only be carried out if a token passes the pin *start*. Transitions not changing the state of the ESM are listed in the state identifiers, e.g., in state *powerUp* two transitions caused by tokens passing one of the pins *status* or *start* may be executed without changing the ESM state.

In the transition markings, a pin identifier before the slash symbol (e.g., *start /*) marks that the transition is triggered by a token flow originating from the environment of the building block. In contrast, if there is no pin ahead of the slash (e.g., */ status*), the transition is initiated by the block itself which starts a flow towards its environment. A combination of different pins on a single transition specifies that several pins are passed by token flows in the same transition. For instance, *speed / activateS* + *puFailure* + *stopped* expresses that the transition is triggered by an incoming token passing the pin *speed* which in the same step leads to tokens leaving the block via the pins *activateS*, *puFailure* and *stopped*.

Altogether, our block uses 12 different pins to interact with its environment. They are sketched in the following:

- *start:* The start sequence of this block is initiated. The token uses a long integer as parameter which describes the maximum speed limit of the motor.
- *puCompleted:* A token passing this pin confirms that the powering up phase of the motor is finished.
- *speed:* The current speed of the motor is sent as a long integer value in the tokens passing this pin.
- *slsCommand:* A token leaving the SLS block through this pin contains a string which is a command to the motor to reduce its speed.
- *activateS:* A token passing this pin activates the Safe Stop Emergency (SSE) function stopping the motor.
- *puFailure:* By this pin, the environment of the SLS is notified of a failure in the powering up phase due to an incoming speed or re-enabling signal.
- *status:* Tokens passing this pin send status information about the SLS block in form of integer values.
- *setSLS:* By tokens passing this pin, the maximum speed limit may be altered. The tokens contain a long integer carrying the new maximum speed.
- *disableit:* This pin is used to disable the SLS block.

[1]To avoid the problem of a proper vertex placement when displaying an ESM, Arctis uses the hierarchical style listed in Fig. 1.

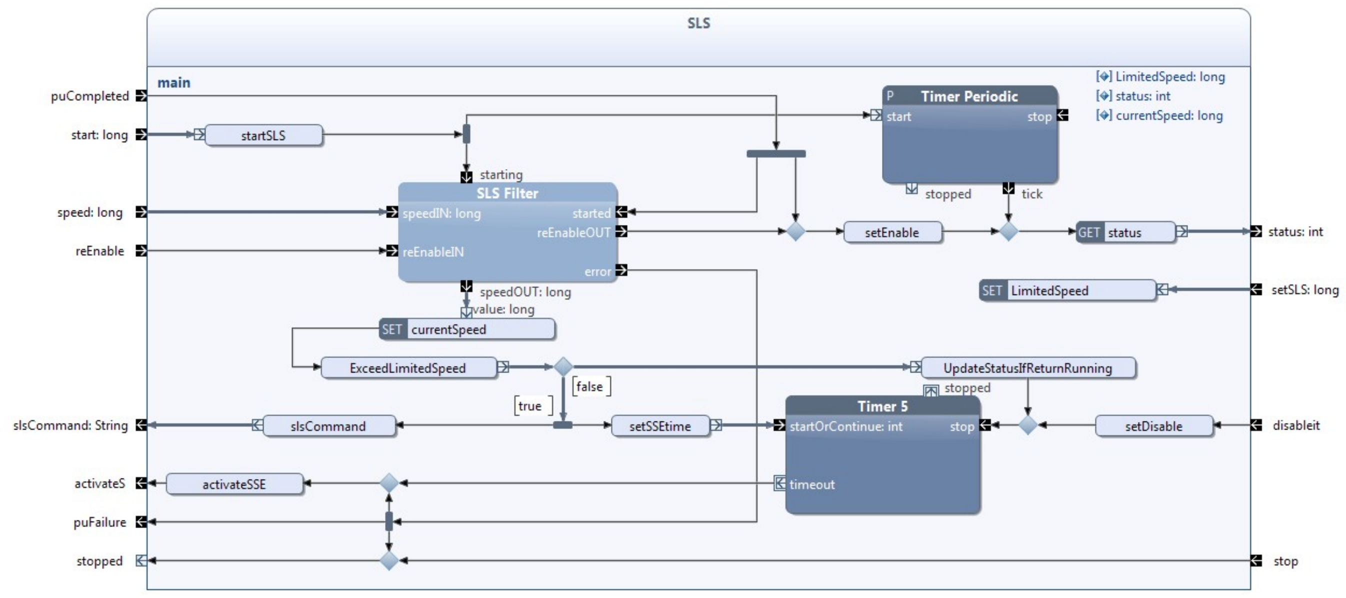

Fig. 2. Activity of Secure Limited Speed (SLS) Building Block

- *reEnable:* A token through this pin re-enables the SLS block.
- *stop:* Flows passing this pin switch off the SLS block.
- *stopped:* This pin is a notification that the SLS block is terminated.

The block is started by a token passing pin *start* and remains in state *powerUp* until a notification is received via *puCompleted*. Thereafter, the system is in state *runningMode* in which the current speed is received periodically. If the speed exceeds the given speed limit, the SLS block switches to state *speedExceedSLS* which leads to control commands via pin *slsCommand*. The block stays in the state *speedExceedSLS* until either the speed falls below the limit again after which it is set to state *runningMode* or a timeout happens. In the latter case, a token is sent via *activateS* to initiate an emergency stop and the SLS block is set to the state *activateSSE*. Further, the system may be disabled as indicated by tokens via pin *disableit*. Thereafter, the block will be in state *disable* until it is re-enabled via pin *reEnable* and reaches state *runningMode*. In addition, the SLS notifies its environment about its status by flows through pin *status*. It is terminated either by an initiative from the environment which may send tokens via pin *stop* or by erroneous speed or re-enabling signals during the power up phase. In the latter case, the environment is also notified by a failure message via pin *puFailure* and an emergency stop command via *activateS*.

## B. Behavior of the SLS Component

As sketched in the introduction, system and building block behavior is modeled in SPACE and Arctis by UML 2 activities [4]. The activity of the SLS block is depicted in Fig. 2. It contains the blocks *Timer Periodic* and *Timer 5* which were taken from an Arctis standard library and describe two different kinds of timers. The block *SLS Filter* was created by the developer of the SLS block to handle the special treatment of speed and re-enabling messages occurring in the power up phase. *SLS Filter* is a so-called *shallow block* for which only the pins and the ESM have to be specified while Arctis generates the block behavior automatically.

The 12 pins introduced above are specified as *parameter nodes* on the edge of the activity in Fig. 2 that we will also call “pins” for simplicity. A flow passing pin *start* is forwarded to the operation action *startSLS* that contains a Java method of the same name. In this method, the limited speed value is stored in the long integer variable *LimitedSpeed* while the block status, represented by the correspondent variable, is set to the value *active*. Thereafter, the token proceeds to a fork node in which it is duplicated. One token copy reaches the block *SLS Filter* to enable error handling of speed and re-enabling messages during power up. The other copy starts the block *Timer Periodic*. From now on, this block will, in intervals, create tokens leaving its pin *tick* and being forwarded via a merge node to the get variable action *GET status*. Here, the current value of variable *status* is stored as a token parameter. Afterwards, the tokens are sent to the environment via pin *status*. The completion of the powering up phase is notified by a token entering the block via pin *puCompleted*. This token is duplicated in a fork. One copy switches the block *SLS Filter* into normal mode such that speed and re-enabling messages are normally treated while the other copy adjusts the status variable and issues a status output to the environment.

Speed messages reaching pin *speed* are forwarded to the block *SLS Filter*. If the block is in the power up phase, the token is forwarded via pin error to a fork which generates three copies forwarded to the pins *activateS* via an operation adjusting the status, *puFailure* and *stopped*. The pin *stopped* is a terminating pin. When it is passed by a token, all remaining tokens in the system are removed and the three inner blocks are set to their respective idle states. Thus, the SLS building block is re-initialized. If the power up phase is already completed, the speed message is forwarded from block *SLS Filter* to the set variable action *SET currentSpeed* storing the parameter of the token. Thereafter the flow reaches the operation *ExceedLimitedSpeed* in which the current speed is compared with the speed limit. The token leaving this block carries a boolean value. If the motor is on overspeed, it is forwarded from the decision node via the edge *true* to a fork. One of the copies is forwarded to the operation *slsCommand* which generates the speed reduction command leaving the block via pin *slsCommand*. The other token starts block *Timer 5* to enable an activation of the SSE system if the overspeed stays for a certain period of time, i.e., 1000 ms. If the motor runs in normal speed, the token leaves the decision via the edge *false* which updates the status and stops the timer. A timeout of the timer in block *Timer 5* leads to a token passing pin *activateS*.

Disabling the SLS block by sending a token through pin *disableit* switches off *Timer 5* since an activation of the SSE is not allowed in the state *disable*. The block can be enabled again by a flow via pin *reEnable* which during power up leads to an error handling while, otherwise, the block is re-enabled followed by a status message. Further flows enable to change the limited speed via a token passing pin *setSLS* and to stop the SLS building block using the pins *stop* and *stopped*.

Using the built-in Arctis model checker [3], we could easily prove that the activity listed in Fig. 2 fulfills the ESM depicted in Fig. 1 such that the block correctly realizes its interface behavior. This block can now be combined with similar blocks of the motor control unit and, utilizing the Arctis code generation, program code can be created.

## III. Real-Time Extension

The two key performance requirements of the Safety Limited Speed (SLS) component can be stated as follows:

1) Except for the idle and power up phases, a reduce speed command (*slsCommand*) should always be executed when a speed notification showing overspeed of the motor is detected.
2) The activation of the Safety Stop Emergency (SSE) component shall be executed not later than 1000 ms after overspeed was detected, as long as the speed does not fall back below the limit in this period.

The first requirement holds obviously since one can easily see from the activity in Fig. 2 that every speed input showing overspeed leads to a slsCommand. Formally, that can be verified using the model checker TLC [7].

The second requirement describes a typical hard real-time requirement that, before now, could not be modeled and verified directly by the tool-set Arctis. To prove properties like this one, we had to extend the SPACE syntax and the analysis capabilities of Arctis in a way that real-time properties including bounded response-time guarantees can be modeled and verified as well (see also [14]–[16]). While the semantics of SPACE is based on Lamport's Temporal Logic of Actions (TLA) [8], we refrained from using the corresponding way to model real-time [17], [18]. The problem is that in this method, real-time is specified by real numbered values which would lead to systems enclosing huge (or even infinite) numbers of states that exceed the ability of TLC and other model checkers. Thus, we would need to use either manual verification or rely on theorem provers requiring a significant verification guidance such that carrying out the proofs would be long-lasting and tedious. Therefore we decided to base the system extension on Timed Automata [11] and the verification tool UPPAAL [10] which are sketched in Sect. III-A. We decided to extend the ESMs by annotations that make it possible to model real-time properties which have to be kept by both the building block to which the ESM belongs and its environment. The resulting *Real-Time External State Machines* (RTESMs) are introduced in Sect. III-B. Together with the activities, they are transformed to Timed Automata which form the input for the UPPAAL-based proofs. The corresponding mapping is described in Sect. III-C.

### A. Timed-Automata and UPPAAL

Several approaches to model real-time properties are available, e.g., IO automata [19], hybrid automata [20] and timed statecharts [21]. We decided to apply *Timed Automata* (TA) [11] which were employed by Rajeev Alur and David Dill in 1990 since TAs fit excellently with SPACE and Arctis and provide a powerful model checking environment as we will discuss below. Timed Automata are extended finite state machines which allow to specify real-time values as *environment clock variables*. These variables are synchronized with the clock such that they express natural time elapsing. Each variable may be set or reset when the state machine carries a certain transition. Further, one may define so-called *clock invariants* restricting the time, a state machine may rest in a certain state of a TA.

UPPAAL [10] is a modeling and verification tool for Timed Automata which is suited to verify that a system fulfills certain real-time properties. It enables to express systems consisting of various timed state machines, so-called *templates*, which can run in parallel and interact via synchronization channels. A TA transition can be amended by the following annotations:

- *Guards:* A transition may only be taken if its guard is true.
- *Synchronization:* Timed automata exchange signals with each other synchronously by *send* (!) and *receive* (?) signal pairs. The synchronization supports binary transmissions from a timed automaton to another one as well as broadcasts. In the latter case, a signal is sent by one timed automaton and received by various others.
- *Updates:* The environment variables are updated when a transition is carried out.

The clocks, invariants, guards, and updates of the timed automata are represented in UPPAAL as annotations of the state machines using a C-like syntax. Further, timed properties to be verified are expressed as formulas in a subset of the branching-time temporal logic *Timed Computation Tree Logic* (TCTL) [12].

### B. Real Time External State Machines

We extend the ESMs in SPACE to *Real-Time External State Machines* (RTESMs) by introducing also environment clock variables, clock invariants and updates. This well-arranged concept of the Timed Automata allows to model real-time

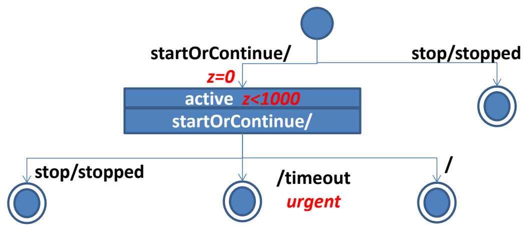

Fig. 3. Real-Time External State Machine (RTESM) of block *Timer 5*

properties to be fulfilled by the interface behavior of building blocks in an easily understandable way. Like in the TAs, the states may contain clock invariants determining how long the RTESM may stay in a state. However, the initial and terminating states must not contain clock invariants. The RTESM transitions may contain updates to manage the environment variables.

As an example, we depict the RTESM of the Arctis building block *Timer 5* in Fig. 3. This block realizes a persistent timer running 1000 ms after being started via a token through pin *startOrContinue* until it issues a token via pin *timeout*. The time is expressed by the clock variable $z$ which is set to its initial value 0 when the transition *startOrContinue* is executed for the first time switching the RTESM from the state *idle* to *active*. When further tokens pass the pin *startOrContinue* in state *active*, $z$ is not set to 0 which models the persistence property of the timer. The state *active* is provided with the clock invariant $z < 1000$ stating that the RTESM may only stay less than 1000 ms in this state before it has to be left, i.e., an *urgent* action (*timeout*) has to be taken when the timer reaches 1000 ms. Transitions representing urgent actions are marked with the label *urgent*. These constructs allow to model bounded liveness properties [14], e.g., guaranteeing that the transition *timeout* must be executed if it, otherwise, will be continuously enabled. In Sect. IV, we show that the urgent transitions are used to find out whether a clock invariant is guaranteed by the activity of the building block to which the RTESM is assigned or by the one carrying the block identifier. To guarantee that this is unambiguously defined, each state containing a clock invariant must have at least one downstream edge marked as urgent and all urgent edges leaving the same source state have to be either triggered by the activity in the block or the one modeling the environment.

The RTESM of the block *Timer 5* states that, as long as the timer is not stopped (transition $stop/stopped$) resp. deleted together with the block including it (transition $/$), a token has to pass pin *timeout* before the 1000 ms limit is reached. That is exactly the real-time property we like to be guaranteed by the building block.

## C. Mapping from SPACE to UPPAAL

The reactive SPACE semantics [5] defines the UML 2 activities as state transition systems in a run-to-completion fashion. The states are represented by tokens resting on activity vertices and edges. A token may stop only on activity nodes modeling system elements demanding it to wait for a certain period of time. Besides initial nodes describing the token setting at system start, there are receiving nodes at which a token has to wait for a signal from an external event source.

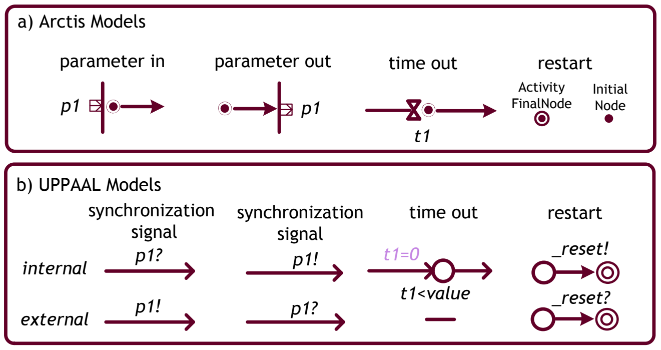

Fig. 4. Mapping between the Activity Behavior and TAs

Likewise, tokens reaching timer nodes have to wait for a fixed perioded of time. Further, a token may wait on the edge leading to a join node which, being the complement of a fork, synchronizes various tokens, i.e., only when tokens are reaching all of its input edges, one of them may pass the join. In analogy to Petri-nets, we call the nodes and edges, on which tokens may rest, *inner places*. At most one token may wait at the same time on an inner place.

A collaborative building block as discussed in [3] refers to various components which are represented in its activity by different partitions. Thus, edges crossing a partition border model the interchange of signals between the components. Since the communication is asynchronous, these edges include *queue places* on which tokens rest during transfer. In contrast to the inner places, queue places may contain various tokens at the same time. They are stored in a FIFO queue with a limited queue size.

In the SPACE semantics, a transition corresponds to a so-called activity step, in which a token starts at an inner or queue place and moves forward on the activity graph in a run-to-completion fashion until it reaches another place respectively a final node on which it is deleted. If the token passes an operation, the corresponding Java method is executed. If the pass of the token goes via a fork node, on which it is duplicated, all copies are handled within the same activity step. Likewise, the jumping of a token between activities via the pins takes place within a single activity step. The reactive nature of the semantics is guaranteed since any transition is triggered by the reception of a signal or a timeout. The semantics facilitates an automatic transformation of the activities to executable UML 2 state machines [22] from which Java code can be generated.

By our Arctis to UPPAAL mapping algorithm introduced below, an Arctis building block is transformed into a network of Timed Automata (TAs). To facilitate the understanding of this transformation, we distinguish so-called *internal TAs*, which are generated from the activities, from *external TAs* transformed from the RTESMs. The states of an internal TA correspond to the different token settings on the queue and inner places of an activity while the activity steps are mapped to the TA transitions. Some aspects of the mapping of Arctis activities to internal TAs are highlighted in Fig. 4:

- A token arriving at an activity via pin *p1* is translated to a binary synchronization channel in which the

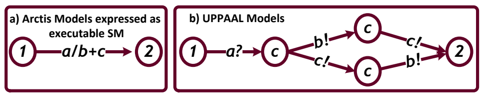

Fig. 5. Several pins in an RTESM transition

internal TA carries out a receive signal (*p1?*).

- A token leaving an activity via pin *p1* is mapped to a binary synchronization channel in which the internal TA executes a send signal (*p1!*).
- A timer *t1* in an activity is transferred to a node of the internal TA, which is provided with an environment clock variable that is set to 0 by the upstream edge to this node. Further, a clock invariant states that the TA may only be in the state if the upper bound limit of the clock is not yet reached (expressed as $t1 <$ *duration*).
- A terminated building block (e.g., by a token reaching an Activity Final Node (◉), may be restarted at any time. To model this property also in the TAs, we add special reset transitions from the final node to the initial node using a broadcast channel $\_reset$.

In practice, a UML activity is first transferred into an executable state machine using the Arctis transformation tool. Thereafter, this state machine is automatically transformed to the internal TA following the mapping sketched above.

Similar to the TAs, the RTESMs are state transition systems which allow for a direct mapping of the states including the clock invariants. The RTESM transitions are transformed to send and receive signals of the synchronization channels representing the pins with which they are annotated. As shown in Fig. 4, all token flows leaving a building block $B$ are specified in the external TAs modeling $B$'s own RTESM and the ones of its inner blocks as receive signals (*p1?*). This is the case since, from $B$'s view, the RTESMs represent the activities of its environment respectively the activities of the inner blocks which all receive the tokens sent by $B$. Likewise, token flows heading towards $B$ are modeled in the external TAs by send signals (*p1!*).

An issue to be treated when mapping RTESM transitions to transitions of the external TAs, is that the RTESM transitions may refer to various pins while TAs do not allow to combine different synchronization channels in a single transition. We address that by sequences of TA transitions following the order of the pins. This leads to intermediary states that are declared as *committed locations*, i.e., states not modeling any passing of time. For pins which may be executed in parallel, we provide separate paths enabling any communication order. According to our experience, the additional states do not have an appreciable impact on the model checker performance.

An example is shown in Fig. 5. Here, due to the RTESM transition, a building block receives first a token via pin $a$ which, in the same activity step, leads to two tokens leaving the block via $b$ and $c$ in parallel. In the corresponding external environment TA, first a receive signal via channel $a$ is received which leads to an interleaving of sending signals via $b$ respective $c$. Finally, like in the internal TAs we use special reset transitions from the final to the initial nodes to model that RTESMs can be restarted at any time.

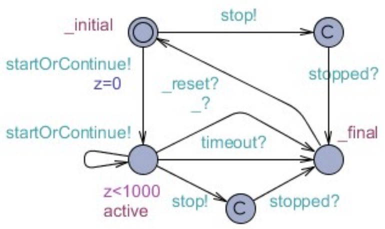

Fig. 6. External block TA of building block *Timer 5*

As an example of mapping an RTESM to an external TA, we use the RTESM of block *Timer 5* depicted in Fig. 3 and the resulting external TA shown in Fig. 6. We see that the three states of the RTESM. i.e., *active* as well as the initial and final states are mapped to three states *active*, *_initial* and *_final* in the external TA. Further, two committed locations (ⓒ) are used to treat the two RTESM transitions *stop/stopped*. Since this external TA shows the view from the block *Timer 5*, the transition *startOrContinue/* which is originated from the environment of this block is represented as a send signal.[2] The environment variable $z$ and the invariant on the state *active* are directly mapped to the external TA. The restart of the RTESM is specified by the transition *_reset?*.

To prove that a building block $B$ fulfills the real-time properties stated in its own RTESM and the ones of its inner blocks, we use the transformations introduced above to create a network of TAs. It consists of the inner TA representing $B$'s activity as well as the external TAs mapped from its RTESM and the ones of its inner blocks. In the next section, we show how this network is used to prove with UPPAAL that the real-time properties stated in the RTESMs are kept.

## IV. Compositional Verification of Real-Time Behavior

As mentioned in the introduction, an advantage of using ESMs to model the interface of building blocks is that the number of states to be inspected by the model checkers can be kept smaller than in monolithic proofs. This results from the fact that the ESMs represent the behavior of both the environment of a block and its inner blocks in a more abstract way. The verification of properties is provided in two separate steps (see [3], [9]). First, one verifies for each instance $C$ of a building block in a SPACE system model that $C$'s activity as well as the one of its environment block $B$ fulfill $C$'s ESM. Thereafter, if we want to prove that $B$ meets certain properties, we can replace the activities of its inner blocks (e.g., $C$) with their ESMs. Since these ESMs usually contain less states than the corresponding activities as they do not model internal behavior, this reduces the state space of the model checker runs significantly.

We aim at using this kind of compositional verification also for the UPPAAL-based real-time proofs. As introduced in

[2]If we create the external TA of this RTESM from the viewpoint of the environment of *Timer 5*, e.g., block *SLS* in Fig. 2, *startOrContinue/* is modeled as a receive signal.

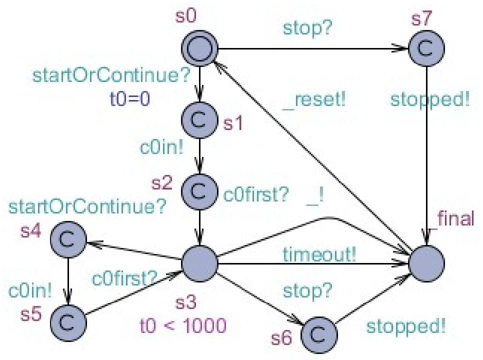

Fig. 7. Internal TA of building block *Timer 5*

Sect. III, we express real-time properties in form of states with clock invariants defined in the RTESMs. Moreover, each clock invariant is guaranteed by exactly one of the two activities to which an RTESM refers. To model which activity indeed has to guarantee a clock invariant, we use the labels of type *urgent* that are assigned to transitions leaving RTESM states with clock invariants. For instance, the clock invariant of state *active* in Fig. 3 has to be guaranteed by the block *Timer 5* since the transition */timeout*, that is labeled urgent, is triggered from this block. Below, we describe clock invariants guaranteed by the inner activity of the block as $cib$ and the ones realized by the environment activity as $cie$. The verification of the two types of clock invariants is conducted by proving that the activity of a block $B$ fulfills all clock invariants of type $cib$ of its RTESM as well as all the clock invariants of type $cie$ in the RTESMs of its inner blocks. As discussed in Sect. V, these verifications are sufficient to allow reducing the proof that a system $Sys$ fulfills a real-time property $\mathcal{P}$ to the verification that an activity $\mathcal{A}$ in $Sys$ implies $\mathcal{P}$ as long as we represent the environment of $\mathcal{A}$ as well as its inner blocks by the respective RTESMs.

To prove that a block $B$ indeed fulfills the mentioned clock invariants, the transformation tool uses the network of internal and external TAs described at the end of Sect. III. It makes first a state space exploration of the combined states of this network and mark all those combined states to which one of the external TAs participate with a state containing a clock invariant that has to be guaranteed by $B$. For instance, to prove that the clock invariant $z < 1000$ in state *active* of the external TA of block *Timer 5* in Fig. 6 is met, we extract all combined network states in which *active* is the state component of this TA. This, are altogether five states to which the internal TA of *Timer 5* (see Fig. 7) participates with its states *s1* to *s5*.

For each state of the internal TA, that refers to a marked combined state but is not a committed location, we create a TCTL invariant stating that in this state the clock invariant of the corresponding state in the external TA is fulfilled. In our example, *s3* is the only one of the five states that is not a committed location such that we create the following invariant specified in the TCTL subset accepted by UPPAAL:

$$A[](Timer5iTA.s3\ imply\ Timer5eTA.z < 1000) \quad (1)$$

Here, the designators $Timer5iTA$ and $Timer5eTA$ refer to the internal resp. external TA of *Timer 5*.

Thereafter, we verify the created TCTL invariants with UPPAAL. It accepts formula (1) since the clock invariant of state *s3* modeling a timer node guarantees that state *s6* or *_final* will be reached within 1000 time units resulting that the external TA leaves its state *active* within this period of time as well.

It is sufficient to prove the TCTL invariants generated by our tool since all other states $s$ of the internal TA fulfill at least one of the following two properties:

1) The state $s$ is a committed location. In this case, it is per definition timeless and will be left without any elapsing of time. By the built-in Arctis model checker, we proved that the activity of the analyzed block is in compliance with its ESMs. This guarantees by construction that, if an internal TA awaits a signal in a committed location, the external TA sending this signal is also in a committed location such that the signal is immediately sent.
2) The state $s$ does not participate in a combined state of the network of TAs to which one of the external TAs participates with a state carrying a clock invariant. The internal TA may rest arbitrarily long in state $s$ since that does not violate any real-time properties.

## A. Proving the Real-time Property of the SLS Block

Using the method described above, we can verify the second property stated in Sect. III, i.e., that the motor may be permanently in overspeed for at most 1000 ms until the Safety Stop Emergency (SSE) component is activated. For the proof of this property, we use a network containing the internal TA of the SLS block depicted in Fig. 8 that we call $i_{SLS}$ below. The network includes also an external TA representing the RTESM of the SLS block. It corresponds with the ESM shown in Fig. 1 but is amended with an environment clock variable $z$ and a clock invariant $z < 1000$ stating the RTESM does not remain longer than 1000 time units (i.e., milliseconds) in the state *speedExceedSLS*. Further, the external TA of the block *Timer 5*, named $eb_{T5}$ in the following, is part of the network. It corresponds with the TA listed in Fig. 6 but with permuted send and receive signals since, here, we reflect the view on the RTESM from the environment of block *Timer 5*. Finally, the network contains the RTESMs of the other two inner blocks *Timer Periodic* and *SLS Filter*.

For easier understanding, we colored the transitions of $i_{SLS}$ that are synchronized with the ones of $eb_{T5}$ in red. The state *s14* on the right upper corner of the graph in Fig. 8 indicates the state that overspeed was detected but the SSE not yet triggered. This state can only be reached from state *s2* on the top center if a *startOrContinue* signal is sent to *Timer 5*. Thus, $eb_{T5}$ will be in state *active* which it has to leave within 1000 ms according to its clock invariant. The state *active* is left if $i_{SLS}$ moves to state *s2* since it sends a *c1stop* (*c1* is the instance name of *Timer 5* in activity *SLS*) signal to $eb_{T5}$. This path models that the speed has fallen below the critical limit again. Likewise, *c1stop* is transmitted to $eb_{T5}$ on the path to state *s12* modeling that the SLS block was disabled. Also the pass to *s22*, indicating that the SLS block was stopped, allows $eb_{T5}$ to leave its state *active* since then the guard of the spontaneous transition to the final state is true. If not one of these three passes are followed, $i_{SLS}$ remains in *s14* or one of the committed locations through which one circles back to

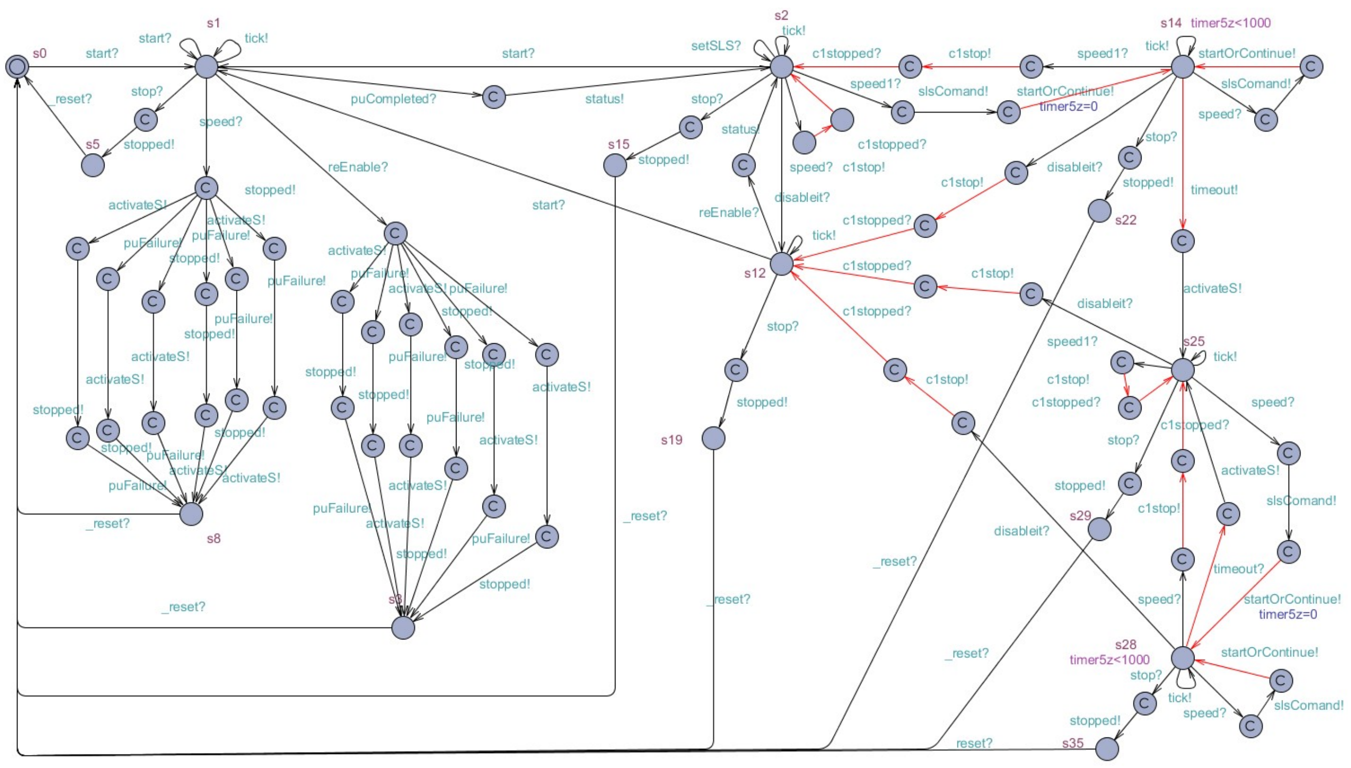
Fig. 8. Internal timed automaton of SLS building block

*s14* when a new overspeed notification is handled. Due to the clock invariant in $eb_{T5}$, a time-out signal will be sent within 1000 ms, after which $i_{SLS}$ reaches a committed location such that it immediately sends an *activateS* signal to its environment which basically guarantees the key requirement.

The states *s14* and *s28* are the only not committed states in $i_{SLS}$ which refer to state *speedExceedSLS* in the RTESM of the SLS block. Therefore the transformation tool just created the following invariant with clock *sls* describing the external interface time constraints of the *SLS* building block:

$$A[](SLSiTA.s14||s28\ imply\ SLSeTA.sls < 1000) \quad (2)$$

Due to the clock invariant of the RTESM of block *Timer 5* that we proved in formula (1), UPPAAL accepted this invariant as being correct. Running on an *Windows 7* laptop system with 8 GB memory and an Intel(R) Core(TM) 2 Duo 2.40 GHz CPU, the verification of formula (2) took 0.004 second, 27352 KB of virtual memory and 7576 KB of resident memory at peak. The proof of formula (1) needed even less computing resources.

## V. Correctness of the Compositional Verification

In Sect. IV, we pointed out that it is sufficient to prove certain TCTL invariant formulas with UPPAAL to verify that the activity of a block $B$ together with real-time properties fulfilled by $B$'s environment and its inner blocks guarantees a certain real-time property $\mathcal{P}$. To make these checks practically feasible, however, it has to be confirmed that the UPPAAL-based checks are sufficient to guarantee that $\mathcal{P}$ is also met by the system $Sys$ containing block $B$. In theory, $Sys$ might contain other building blocks which impede or delay the execution of crucial transitions leading to a violation of $\mathcal{P}$. In the following proof, we establish that our compositional concept using RTESMs as behavioral interfaces rules such real-time hampering behavior out.

A system $Sys$ in SPACE and Arctis can be seen as a tree structure of building blocks since any inner block of an activity may contain inner blocks as well. For example, the block *Timer 5* uses a shallow block (see Sect. II-B) from an Arctis library filtering out all but the first token reaching the block via pin *startOrContinue*. In this tree, an activity $\mathcal{A}_b$ is the parent of another activity $\mathcal{A}_c$ if $\mathcal{A}_c$ is the inner activity of block $c$ and $\mathcal{A}_b$ its environment activity. If we want to prove that a system $Sys$ modeled in Arctis fulfills a certain invariant real-time property $\mathcal{P}$, this corresponds to the verification of the following equation:

$$\bigwedge\nolimits_{b\in Act(Sys)} \mathcal{A}_b \Rightarrow \mathcal{P} \quad (3)$$

Here, $Act(Sys)$ refers to the set consisting of all the activities specifying system $Sys$.

To utilize the compositional nature of our verifications, however, we want to verify that $\mathcal{P}$ is fulfilled by a single activity $\mathcal{A}_b$ together with its RTESMs as discussed in Sect. IV. For example, the proof in Sect. IV-A shall be sufficient to assure that the overall motor control system guarantees the real-time property stated in Sect. III. A UPPAAL-based proof, that a real-time property $\mathcal{P}$ stated as TCTL formulas is kept by a network of TAs, corresponds to equation

$$\mathcal{A}_b \wedge ci_b \wedge \bigwedge\nolimits_{c\in Chld(b)} ci_c \Rightarrow \mathcal{P} \quad (4)$$

where $ci_b$ denotes that the clock invariants in the RTESM of activity $\mathcal{A}_b$ are met by block $b$ or its environment. $Chld(b)$ refers to the set of $b$'s children in the tree mentioned above. In consequence, we have to justify that proving equation (4) is sufficient to guarantee equation (3). This can be achieved by verifying the following fomula:

$$\bigwedge\nolimits_{b \in Act(Sys)} \mathcal{A}_b \Rightarrow \forall b \in Act(Sys) : ci_b \wedge \bigwedge\nolimits_{c \in Chld(b)} ci_c \tag{5}$$

It is evident that the conjunction of equations (4) and (5) directly implies equation (3).

For the proof that equation (5) holds, we utilize the proceeding model checkers use to verify invariants. A model checker proves that the invariant is fulfilled by the initial states of a system and that none of the system transitions falsifies the invariant if it holds before. In consequence, the invariant is fulfilled by all reachable states of the system. Be $S_b$ the set of all reachable states of activity $\mathcal{A}_b$ and $Init_{S_b} \subseteq S_b$ the set of its initial states. The SPACE approach is constraint-oriented (see [23]). That means, all state designators (i.e., queue and inner places resp. variables) are assigned to exactly one activity. Thus, we can define the system state space as the Cartesian product of all state sets in the activities (i.e., $S_{Sys} \triangleq S_1 \times \ldots \times S_n$ if $Sys = \{1, \ldots, n\}$). The set of initial system states is defined as $Init_{Sys} \triangleq \{\langle s_1, \ldots, s_n \rangle : s_i \in Init_{S_i}\}$. By the function $ls \triangleq (s : S_{Sys}, a : \{1, \ldots, n\}) \rightarrow S_a$, we map a system state $s$ to the state component expressing the state of activity $a$.

Be $\widehat{T_b} \subseteq S_b \times S_b$ the set of activity steps carried out in activity $\mathcal{A}_b$. Then we define the transitions of this activity as $T_b = \widehat{T_b} \cup \{\langle s, s \rangle : s \in S_b\}$ allowing also stuttering steps in which the activity does not change its state. So, we can define the set of system transitions $T_{Sys} \triangleq T_1 \times \ldots \times T_n$ since, in a constraint-oriented model, each component takes part by either a local transition or a stuttering step in a system transition. Moreover, we use a mapping $lt \triangleq (t : T_{Sys}, a : \{1, \ldots, n\}) \rightarrow T_a$ to access the local transition of activity $a$ carried out in the system step $t$. Now we can express formula (5) as the conjunct of the two following equations:

$$\begin{array}{l} \forall s \in Init_{Sys} \forall b \in Act(Sys) : \\ \quad ci_b(ls(s, b)) \wedge \bigwedge\nolimits_{c \in Chld(b)} ci_c(ls(s, c)) \end{array} \tag{6}$$

$$\begin{array}{l} \forall s, s' \in S_{Sys} \forall b \in Act(Sys) : \\ \quad lt(\langle ls(s, b), ls(s', b) \rangle, b) \in T_b \\ \quad \wedge ci_b(ls(s, b)) \wedge \bigwedge\nolimits_{c \in Chld(b)} ci_c(ls(s, c)) \\ \quad\quad \Rightarrow ci_b(ls(s', b)) \wedge \bigwedge\nolimits_{c \in Chld(b)} ci_c(ls(s', c)) \end{array} \tag{7}$$

Here, $ci_b(s)$ states that the clock invariants of the RTESM of activity $\mathcal{A}_b$ hold in its state $s$. Equation (6) is trivially true since in the initial system state all the RTESMs are in their idle states that must not carry any clock invariants. Equation (7) is guaranteed by the TCTL invariant proofs discussed in Sect. IV. There, we proved by UPPAAL for every activity $\mathcal{A}_b$ that the clock invariants of its own RTESM as well as the ones of its inner blocks are guaranteed after carrying out any of its local transitions as long as they were preserved also before. Since $\mathcal{A}_b$ participates in a system transition either by a local activity step or a stuttering step, this implies formula (7) directly.

Thus, by combining the various proof steps discussed above, we verified that one can replace a complex UPPAAL proof using all the internal TAs of the involved Activities, i.e., equation (3), by a number of much simpler RTESM proofs, i.e., the verifications of the clock invariants fulfilling formula (7), as well as a property proof using the RTESMs of the inner blocks of an activity, i.e., equation (4). In all these proofs, a much smaller number of system states has to be checked such that the compositional verification is a useful means to circumvent the state space explosion problem.

## VI. Related work

Proposing model-driven development and verification of real-time systems using UML models is not a new idea. Similar to us, David et al. [24] extend UML statechart diagrams with real-time constructs and translate the resulting formalism (Hierarchical Timed Automata) into networks of TAs that can also be checked with the tool UPPAAL. In contrast to us, however, they do not utilize the structure of their models to reduce the verification overhead as we do by applying compositional verification. In [25], Knapp et al. describe their prototype tool HUGO/RT for the modeling of a generalized realroad crossing (GRC) problem. The control state machines of models are translated into Timed Automata in UPPAAL verifying the safety and utility properties of the GRC problem. In [18], Graw et al. suggest to use cTLA, a compositional extension of the Temporal Logic of Actions TLA [8], for the formal verification of UML models that describes real-time behaviors of a system. In [26], Furfaro and Nigro specify the translation of models in the formal language H-CRSM, which can also be used for the modular development of reactive real-time systems, to TAs in UPPAAL. In [27], Dong et al. summarize a series of patterns when modeling real-time systems using timed automata and provide a translation from a real time specification language, i.e., timed communication sequential process (CSP), to timed automata to facilitate the verification capabilities in UPPAAL.

Abstracting program code is another way to prove real-time constraints. For instance, Chaki et al. describe in [28] the tool MAGIC which is capable to abstract C-code into Labeled Transition Systems (LTSs) preserving the real-time properties of the code. Similarly, Gong et al. [29] construct UPPAAL models directly from source code in order to check various real-time, safety and liveness properties.

## VII. Conclusion and Future Work

In this paper, we presented an extension of our model-based engineering approach SPACE for reactive systems to support also the description and model checker-driven verification of real-time properties. In particular, we extended the External State Machines (ESMs) describing the interfaces of our building blocks to Real-Time ESMs (RTESMs) which enable to specify invariant real-time properties fulfilled by a building block and its environment. This approach allows for an automated transformation to Timed Automata and, in consequence, verification using the model checker UPPAAL. As shown, the approach makes it possible to use compositional verification reducing the state space to be checked significantly.

In its present state, our approach does not consider time-delays caused by executing the activity steps and, in particular, the Java methods which are supported by our current platform. In the ongoing research, we propose to take the execution time into consideration, and associate a building block with a task model. For that purpose, we can apply code level benchmarking techniques to evaluate and predicate the *best cast execution time (bcet)* and *worst case execution time (wcet)*. In the model level, we propose to translate building blocks into Timed Automata as Task models (see [30]), such that the performance and schedulability of a building block can be analyzed by such task models.

Already in its present state we consider our approach meaningful since it allows to detect real-time flaws in system designs which can already be found and corrected in the early development phase of system modeling. Due to the compositional verification, we can use the approach for real-life applications like the speed control protection mechanism for motors introduced in this paper. As missing real-time constraints are a major issue for the violation of safety properties, we consider this work as a suitable extension to our endeavor for the creation of safe embedded systems (see, e.g., [9]). The Arctis tool including standard libraries of building blocks is available from Bitreactive AS.[3]

## References

[1] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[2] F. A. Kraemer, "Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks," Ph.D. dissertation, Department of Telematics, Norwegian University of Science and Technology (NTNU), 2008.

[3] F. A. Kraemer, V. Slåtten, and P. Herrmann, "Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services," *Journal of Systems and Software*, vol. 82, no. 12, pp. 2068–2080, 2009.

[4] Object Management Group, "Unified Modeling Language: Superstructure, Version 2.3," 2010.

[5] F. A. Kraemer and P. Herrmann, "Reactive Semantics for Distributed UML Activities," in *Formal Techniques for Distributed Systems, Joint 12th IFIP WG 6.1 Int. Conf. (FMOODS10) and 30th IFIP WG 6.1 Int. Conf. (FORTE10)*, ser. Lecture Notes in Computer Science, J. Hatcliff and E. Zucca, Eds., vol. 6117. Springer, June 2010.

[6] ——, "Automated Encapsulation of UML Activities for Incremental Development and Verification," in *Proceedings of the 12th Int. Conference on Model Driven Engineering, Languages and Systems (MoDELS)*, ser. LNCS, A. Schürr and B. Selic, Eds., vol. 5795. Springer-Verlag, Oct. 2009, pp. 571–585.

[7] Y. Yu, P. Manolios, and L. Lamport, "Model Checking TLA+ Specifications," in *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME99)*. London: Springer-Verlag, 1999, pp. 54–66.

[8] L. Lamport, *Specifying Systems*. Addison-Wesley, 2002.

[9] V. Slåtten, F. A. Kraemer, and P. Herrmann, "Towards Automatic Generation of Formal Specifications to Validate and Verify Reliable Distributed Systems: A Method Exemplified by an Industrial Case Study," in *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE11)*. ACM, 2011, pp. 147–156.

[10] J. Bengtsson, F. Larsson, P. Pettersson, W. Yi, P. Christensen, J. Jensen, P. Jensen, K. Larsen, and T. Sorensen, "UPPAAL: A Tool Suite for Validation and Verification of Real-Time Systems," 1996.

[11] R. Alur and D. Dill, "Automata for Modeling Real-Time Systems," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, M. Paterson, Ed. Springer Berlin / Heidelberg, 1990, vol. 443, pp. 322–335.

[12] R. Alur, C. Courcoubetis, and D. L. Dill, "Model-Checking for Real-Time Systems," in *5th Symposium on Logic in Computer Science (LICS90)*, 1990, pp. 414–425.

[13] IEC, "International Standard 61800-5-2, Adjustable Speed Electrical Power Drive Systems — Part 5-2: Safety Requirements – Functional," July 2007.

[14] L. Aceto, A. Burgueno, and K. Larsen, "Model Checking via Reachability Testing for Timed Automata," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, B. Steffen, Ed. Springer Berlin / Heidelberg, 1998, vol. 1384, pp. 263–280.

[15] L. Aceto, P. Bouyer, A. Burgueno, and K. G. Larsen, "The Power of Reachability Testing for Timed Automata," *Theoretical Computer Science*, vol. 300, pp. 411–475, May 2003.

[16] M. Lindahl, P. Pettersson, and W. Yi, "Formal Design and Analysis of a Gear Controller," in *4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 1384. Springer Verlag, 1998, pp. 281–297.

[17] M. Abadi and L. Lamport, "An old-fashioned recipe for real time," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1543–1571, 1994.

[18] G. Graw, P. Herrmann, and H. Krumm, "Verification of UML-based real-time system designs by means of cTLA," in *Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC2K)*. Newport Beach: IEEE Computer Society Press, 2000, pp. 86–95.

[19] N. A. Lynch and N. Shavit, "Timing-Based Mutual Exclusion," in *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1992, pp. 2–11.

[20] T. A. Henzinger, "The Theory of Hybrid Automata," in *11th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1996, pp. 278–292.

[21] K. Kesten and A. Pnueli, "Timed and Hybrid Statecharts and their Textual Representation," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer-Verlag, 1992, pp. 591–620.

[22] F. A. Kraemer and P. Herrmann, "Transforming Collaborative Service Specifications into Efficiently Executable State Machines," *ECEASST*, vol. 6, 2007.

[23] R. Kurki-Suonio, *A Practical Theory of Reactive Systems — Incremental Modeling of Dynamic Behaviors*. Springer-Verlag, 2005.

[24] A. David, M. O. Müller, and W. Yi, "Formal Verification of UML Statecharts with Real-Time Extensions," in *Fundamental Approaches to Software Engineering (FASE02)*, ser. Lecture Notes in Computer Science, vol. 2306. Springer-Verlag, 2002, pp. 218–232.

[25] A. Knapp, S. Merz, and C. Rauh, "Model checking - timed uml state machines and collaborations," in *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2*, ser. FTRTFT '02. London, UK, UK: Springer-Verlag, 2002, pp. 395–416.

[26] A. Furfaro and L. Nigro, "Model Checking Hierarchical Communicating Real-Time State Machines," in *10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA05)*, vol. 1, Sept. 2005, pp. 6 pp. –370.

[27] J. S. Dong, P. Hao, S. Qin, J. Sun, and W. Yi, "Timed automata patterns," *Software Engineering, IEEE Transactions on*, vol. 34, no. 6, pp. 844–859, 2008.

[28] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of Software Components in C," *IEEE Transactions on Software Engineering*, pp. 385–395, 2003.

[29] X. Gong, J. Ma, Q. Li, and J. Zhang, "Automatic Model Building and Verification of Embedded Software with UPPAAL," in *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE Computer Society Press, Nov. 2011, pp. 1118–1124.

[30] C. Norstrom, A. Wall, and W. Yi, "Timed Automata as Task Models for Event-Driven Systems," in *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, 1999, pp. 182 –189.

[3] http://www.bitreactive.com