Model-Based Specification of Timing Requirements

Christian Buckl
fortiss GmbH
Guerickestr. 25
Munich, Germany
buckl@fortiss.org

Irina Gaponova
Technische Universität
München
Boltzmannstr. 3
Garching, Germany
gaponova@in.tum.de

Michael Geisinger
fortiss GmbH
Guerickestr. 25
Munich, Germany
geisinger@fortiss.org

Alois Knoll
Technische Universität
München
Boltzmannstr. 3
Garching, Germany
knoll@in.tum.de

Edward A. Lee
University of California
at Berkeley
545Q Cory Hall
Berkeley, CA, USA
eal@eecs.berkeley.edu

ABSTRACT

In the past, model-based development focused mainly on functional and structural aspects of the system to be developed. Recently, several approaches to include timing aspects have been suggested. However, these approaches are typically applied in later development phases. Models specifying the requirements with respect to timing without focusing on a specific solution are missing. For example, few models support the specification of the allowed jitter of a system. In this paper, we identify requirements on languages for modeling the desired timing behavior of hard and soft real-time systems by analyzing different application domains. Based on these results, we evaluate existing approaches with respect to their suitability and present a suitable approach. Finally, this paper describes the application of the suggested approach in the context of an example from the automation domain.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies; Languages*

General Terms

Design, Performance

Keywords

Model-Based Development, Real-Time Systems, Requirements Analysis

1. INTRODUCTION

Correct timing is essential for embedded systems. It is necessary to clearly specify and formally verify timing requirements before performing detailed system design. However, this issue has not been addressed adequately in model-driven development processes. Many modeling languages lack an integrated notion of timing [21]. Whether timing requirements are satisfied is typically only checked during testing at the end of development. One major reason is that most model-driven approaches applied to embedded systems in early design-phases lack a systematic and rigorous approach to specify and verify timing requirements [19]. Domain-specific languages for real-time systems consider timing, but are often used for the implementation phase (e.g., automatic code generation) and not for requirements analysis. Recently, a number of generic approaches have emerged that include timing behavior (e.g., MARTE [20]). However, these approaches focus predominantly on later development phases or abstract from details such as allowed jitter, which are essential for requirements analysis. Models specifying timing requirements without focusing on a specific solution (in the problem space) are missing. As a result, timing is typically specified only informally during requirements analysis, potentially leading to incomplete or contradicting requirements. A formal approach would help to avoid these issues and resulting problems in later development phases.

The first main contribution of this paper is the identification of requirements that are necessary to express timing requirements for hard and soft real-time systems. The discussion is based on an analysis of different domains, such as automotive, automation and multimedia and is explored in Section 2. Existing approaches are evaluated with respect to these requirements in Section 3. The second main contribution is the presentation of TReqS, a modeling language for specifying timing requirements, in Section 4. To point out the applicability and effectiveness of this approach, a concrete example from the automation domain is used in Section 5 to discuss the implications of the suggested approach. We can show that timing requirements of this use case can be accurately and completely specified in our model. A complete specification of the timing requirements ensures that design decisions can be evaluated with respect to timing issues at all phases of the development and not only during testing. Furthermore, the suggested approach also offers a quantitative metric on how well the timing requirements are

EMSOFT'10, October 24–29, 2010, Scottsdale, Arizona, USA.

matched. This is especially useful in the context of soft real-time systems with potentially contradictory requirements, e.g., cost vs. timing. Section 6 concludes this paper and discusses potential future work.

2. REQUIREMENTS

Within this section, requirements on a modeling language to specify the required temporal behavior of embedded systems are identified. The results are used in the following sections to evaluate existing modeling languages and to derive a suitable modeling language.

2.1 Method to Derive the Requirements

We derive requirements by analyzing timing constraints of applications from different domains. The analysis covers hard real-time systems (the automotive, avionics and automation domains) and soft real-time systems (amongst others the multimedia domain). Furthermore, different demonstrator setups used by research groups in real-time systems are evaluated (e.g., inverted pendulum, tunneling ball), since these setups typically represent characteristic problems. Finally, existing modeling approaches are evaluated.

2.2 Requirements

One main criterion is that the approach to specify timing requirements should not restrict the solution space for the concrete system. The developer should focus on identifying requirements of the system to build prior to exploring possible solutions.

REQUIREMENT R1. *The temporal behavior of the system should be specified in problem space, that is to say it should be described independently of a specific solution.*

The goal of this paper is to define a modeling language to specify timing requirements in early phases of the development. Thus, the approach should abstract from concrete solutions and especially the used IT platform (controllers, networks). The approach should enable the specification of timing requirements originating from the controlled environment that have to be satisfied by the HW/SW system to perform the control task. Therefore, the model has to abstract concrete execution times of individual components and rather use logical time as a basis to describe the desired timing behavior. To achieve this goal, the model should be based on a model of computation that expresses exactly what is required and no more so as to minimally restrict the solution space. A time-triggered model of computation, for example, would restrict the application areas and the way of implementing the system. The choice of a concrete model of computation to realize the application should take place in later phases of the development process.

REQUIREMENT R2. *Timing requirements must be specifiable in a composable and modular manner.*

Requirements are typically specified by use cases describing the externally observable behavior of the system. Based on these use cases, the system and its core functions can be defined to specify the intended functionality. To describe timing requirements in a composable and modular manner, it is useful to separate functionality and timing. This approach can be motivated by the fact that functions and timing requirements are not directly correlated: the same function may have different timing requirements in different applications. A composable description of the functionality is supported by actor-oriented design [1, 18]. Note that the decomposition of the system into functions is only done at a very coarse level in the requirements analysis phase to describe the basic functionality of the system [6]. Timing requirements can be mapped to this system description by specifying constraints on the temporal interaction between functional components of the system, as well as between these components and the environment. In the following, we will use the term **event** for any of these interactions.

REQUIREMENT R3. *The developer must be able to specify the characteristics of all initial events triggering computations of the system.*

We assume that initial events, such as periodic clocks or interrupt sources, trigger the computation of a system. To determine the load under which the system should still operate, it is necessary to specify properties of these initial events. Events may be aperiodic, periodic or occurring according to a certain pattern. For aperiodic events for example, the probability of events and the minimum/maximum distance between two succeeding events may be specified, while for periodic events, the rate and the jitter/deviation of the occurrence of these events are relevant.

REQUIREMENT R4. *It must be possible to state timing requirements related to the accuracy of sensor data.*

Real-time applications need accurate sensor data for their calculation. They might require either precise information on when observed events occur or a minimal sampling rate to obtain precise measurements. An example for the first case is monitoring the rotation of a crankshaft in engine control. To obtain a precision of 0.1° in case of a maximal rotation of 6000 rpm, a temporal accuracy of about $3\,\mu$s is required [16]. An example for the second case is the detection of events in a surveillance network. The minimal sampling rate depends on the time period for which an object can be observed.

REQUIREMENT R5. *It must be possible to relate the point in time when operations are to be executed with respect to the point in time of the initial event.*

To model end-to-end latency requirements, it is necessary to model the desired point in time when certain operations should be executed. An example is a welding robot that has to perform several actions in a timed sequence.

However, every system can only match the desired timing with bounded accuracy. Hence, it must be possible to state the allowed jitter for the according operation:

REQUIREMENT R6. *It must be possible to specify the allowed jitter of a system with respect to the desired temporal behavior.*

Defining the allowed jitter makes it possible to distinguish between soft and hard real-time systems. While the boundaries between acceptable timing and non-acceptable timing are usually quite tight for hard real-time systems, soft real-time systems can typically tolerate much larger jitter intervals. Since the acceptable jitter is very application-dependent, it is not possible to make any application independent assumptions. A very good example for huge differences is the multimedia domain. While the human's sense

of hearing has a temporal accuracy of about 5 μs, the eye's resolution of perception is limited to 40 ms. Therefore, an audio application with distributed loudspeakers has to fulfill much harder jitter constraints to achieve a correct spatial resolution than a 3D video application.

In some cases, the requirements on the temporal behavior of the system depend on the state of the system or the environment. It must be possible to specify such dynamic real-time systems.

REQUIREMENT R7. *It must be possible to specify real-time systems whose timing depends on the inner state or the environment.*

This requirement can be motivated by an example from the automation domain. Let us consider a conveyor belt that can operate at different velocities. Robots are picking up objects on the belt based on light barriers. If the light barrier is mounted within a certain distance d from the place to pick up the object and the velocity of the conveyor belt is v (assuming no changes of the velocity between detection and pick up), the time when the robot should pick up the object is $t = d/v$. Hence, it must be possible to specify this timing requirement as a *dynamic delay* in the model. To estimate the worst- and best-case requirement, it must be possible to specify the minimal and maximal time delay for this operation.

The final requirement is related to the fact that it might be necessary to find an optimal overall solution when different requirements contradict each other.

REQUIREMENT R8. *It must be possible to specify a metric to evaluate how well a concrete implementation matches the requirements.*

While in hard real-time systems, the requirements are very often strict (meaning all requirements *must* be met), soft real-time systems might be described by loose requirements. In this case, a developer's task is to optimize the system with respect to these requirements. Besides timing requirements, the developer has to take into account constraints such as costs for this optimization process.

R8 does not only affect jitter as described in the context of R6, but also the latency. A good example with relaxed requirements on the end-to-end delay, but stricter requirements on the jitter is video streaming. It is possible to buffer video frames if a longer latency is acceptable, however the jitter requirements *have to be* matched to ensure a continuous video stream. Nevertheless, users will not accept too long end-to-end delays, especially in the context of live broadcast. Hence, a metric must be used to describe that the satisfaction of the user will decrease with an increasing latency.

3. RELATED WORK

This section discusses modeling languages used for specification of real-time systems in different domains. These languages are on the one hand evaluated with respect to the identified requirements and on the other hand used to derive a concept for timing requirements specification. The section starts with an overview of relevant models of computation, e.g., described in the context of the Ptolemy project [11]. Subsequently, the concepts in FOCUS, MARTE, and AUTOSAR are discussed.

3.1 Models of Computation (MoC)

3.1.1 PTIDES

Programming Temporally Integrated Distributed Embedded Systems (PTIDES) [22] is a programming model for distributed embedded systems based on a global, consistent notion of time. PTIDES provides a very good basis for defining an approach to model timing requirements.

The implementation of distributed systems with PTIDES is based on a common notion of time known to some precision. PTIDES uses Discrete Event (DE) semantics, which means that actors interact by timestamped events. The DE model of computation is very general and hence a good basis for describing many kinds of real-time systems. In general, a strict temporally ordered execution of events at each actor is applied, however this temporal ordering can be relaxed in PTIDES for stateless actors. PTIDES uses two notions of time: model/logical time and real/physical time. Sensor events get a timestamp in logical time, which is related to the physical point in time when the event triggers the sensor. The accuracy of the timestamp cannot be specified, but depends amongst others on the specific hardware used ($\nvDash$ R4). The frequency of sensor events can be specified to enable analysis ($\vDash$ R3). Actors can increment the logical time by a specified amount, but separate delay actors are supported as well ($\vDash$ R2). In addition, dynamic delays are available ($\vDash$ R7). PTIDES supports two kinds of actuators: those that take actions immediately upon receiving an event (time stamp corresponds to deadline), and those that use the time stamp of the received event to take action at that time (time stamp corresponds to deadline $\vDash$ R5). Jitter or a metric are not considered ($\nvDash$ R6, R8). Another issue is the interpretation of network components within a distributed system as output actors. This simplifies analysis/implementation as only the computation on the individual nodes must be considered. However, this means that end-to-end delays for a computation within a distributed system have to be divided into several delays to satisfy the timing constraints of the network components. For distributing these delays, the developer needs explicit information, such as worst-case execution times ($\nvDash$ R1).

3.1.2 Giotto

Giotto [12] provides a programming abstraction for hard real-time applications which exhibit time periodic and multimodal behavior, e.g., embedded control systems. Similar to PTIDES, the model is based on a concept of logical time and supports timing annotations independent of the platform and the functionality ($\vDash$ R2). The relation to physical time is guaranteed at the interaction points between actors and hence can be interpreted as desired timing ($\vDash$ R5). However, the Giotto approach does not satisfy all requirements. Due to its time triggered nature, it is generally not suitable for event-triggered systems ($\nvDash$ R1). Furthermore, it does not address any metrics and allows neither dynamic delays nor the specification of jitter ($\nvDash$ R6–R8).

3.1.3 Synchronous Languages

Synchronous languages are based on a synchrony hypothesis which assumes that the interval between two consequent input events is strictly greater than the response time. The time scale is presented in terms of ticks. Inputs are read and outputs are generated instantaneously in each tick. The

precision of the time scale is limited to the minimal interval between two input events (tick length), which has to be large enough to ensure the synchrony assumption ($\not\models$ R4). Jitter and dynamic delays are not addressed ($\not\models$ R6, R7). The very strict model of computation simplifies the analysis of real-time systems, but restricts the solution space ($\not\models$ R1).

3.1.4 Timed Automata and Petri Nets

Automata are a widely used and well known formalism for specification of reactive systems. They present operational semantics and are used for describing the behavior of a system. Since timed automata [2, 15, 3] were introduced, the paradigm of automata has been successfully used for describing timing semantics of systems. Timed automata present systems as a set of states, an initial state, a set of clocks, actions (state transitions), a set of ages and a set of invariants assigned to the states. Parallel composition of automata is supported, e.g. by using signals as synchronization mechanism. Timed Automata are often used with temporal logic annotations, e.g., branching-time temporal logic [4], because this technique supports usage of model checking tools for formal verification of safety/liveness/reachability properties.

Petri Nets are based on an extension of automata theory such that the concept of concurrently occurring events can be expressed. One of their extensions, Time Petri Nets, is used for specifying timing semantics [13]. Cassez and Roux show how a special kind of Time Petri Nets, i.e., where time is associated with transitions, can be translated to Timed Automata [9].

Timed automata are usually used for analyzing system behavior rather than for specification. This is the root cause why no distinction is made between desired timing and allowed jitter ($\not\models$ R5, R6). Instead of specifying the desired timing and allowed jitter for state transitions, only time intervals are used for timing specifications. Dynamic delays can only be specified based on states. Therefore, only discrete dynamic delays can be modeled ($\not\models$ R7). Finally, Timed Automata do not support the specification of a metric to evaluate the degree of satisfaction of a system with respect to its timing requirements ($\not\models$ R8).

3.2 FOCUS

FOCUS [7] is a framework for formal specification and development of distributed interactive systems. Systems are represented by component networks. Similar to synchronous systems, the individual components interact via time discrete data streams. The communication is instantaneous, directed, reliable and data preserving. Every component is specified as a function that describes a mapping of the input stream to a set of possible output streams. A separation of functionality and timing is not intended, but could be achieved by distinguishing between components with zero delay (functions) and components with no effects in the value domain of the data streams (timing) ($\models$ R2). Temporal non-determinism can be specified by allowing several possible output streams.

The possibility to use arbitrarily small time intervals as a basis for data streams and the support of non-determinism enable the specification in problem space ($\models$ R1). Similar to timed automata, the approach does not distinguish between desired timing and jitter ($\not\models$ R5, R6). The environment and related requirements, dynamic delays and a metric are not considered ($\not\models$ R3, R4, R7, R8).

3.3 MARTE

Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [20] introduces a domain view for time modeling and defines standard UML elements to express defined timing concepts of real-time and embedded systems. MARTE has very elaborate concepts for timing specication. It provides logical as well as chronometric clocks (related to physical time). Moreover, not only a single time base, but also multiple time bases can be used. However, important aspects such as jitter, dynamic delays, or required accuracy of sensor data are not supported as modeling entities ($\not\models$ R4, R6, R7), but have to be specified using the Clock Constraint Specification Language (CCSL) described in Annex C of the specification. Hence, developers have to manually specify concepts such as jitter, which introduces complexity and leads to a lack of common semantics with respect to these aspects. In addition, metrics are not covered ($\not\models$ R8).

3.4 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) [5] is an open industry standard for automotive E/E architectures developed by automotive manufacturers and suppliers. The current AUTOSAR release 4.0 introduces timing extensions developed in the TIMMO project [14]. The basic entities in the extension are events ($\models$ R1). Events can be semantically connected to event chains; however the causal relationship between triggering events and follow-up events cannot be described in AUTOSAR. The exact timing of events and event chains as well as their concurrency can be described using the Timing Augmented Description Language (TADL). TADL comprises language constructs for describing constraints for events (e.g., periodic, sporadic) and event chains ($\models$ R3). Delay constraints (i.e., age), reaction and timing constraints can be described as well ($\models$ R5). Furthermore, input and output synchronization constraints for input or output events are available.

The basic concepts of the AUTOSAR timing extensions are very useful for specification of timing requirements. However, some requirements are not satisfied. It is not possible to specify jitter or dynamic delays ($\not\models$ R6, R7). Furthermore, no means for specifying a metric with respect to the satisfaction of timing requirements is available ($\not\models$ R8).

3.5 Summary

Table 1 summarizes the evaluation results. While some requirements are addressed quite well, others are not covered at all. The main reason is that most of the approaches are used for later phases in the development process. Requirements that are still valid in these phases are commonly addressed. The requirements that are usually not addressed are modeling of sensor data accuracy (R4), allowed jitter (R6) and definition of a metric (R8).

The most promising approach is to combine and extend PTIDES and AUTOSAR based on event-based specification of systems. In the following, we will present a solution that covers all stated requirements.

4. TIMING REQUIREMENTS SPECIFICATION (TREQS)

Based on the identified requirements and relevant concepts from related work, this section presents TReqS, an approach for **T**iming **Req**uirements **S**pecification.

Table 1: Evaluation Results

	R1 Problem Space	R2 Composability	R3 Initial Events	R4 Accuracy	R5 Desired Time	R6 Jitter	R7 Dynamic Delay	R8 Metric
PTIDES	no	yes	yes	no	desired time or deadline	no	yes	no
Giotto	no	yes	no	no	yes	no	limited	no
Synchronous Languages	no	yes	no	no	implicit	no	no	no
Timed Automata	yes	yes	no	no	mixed with R6	mixed with R5	limited	no
FOCUS	yes	possible	no	no	mixed with R6	mixed with R5	no	no
MARTE	yes	yes	yes	no	yes	possible (CCSL), but complex	possible (CCSL), but complex	no
AUTOSAR	yes	yes	yes	no	yes	no	no	no

4.1 Time

Just like PTIDES, TReqS uses two different notions of time, namely physical time and logical time.

Physical time relates to the time notion of external observers and hence is the basis for timing requirements on the system to build. Timing requirements can be classified whether they relate to relative or absolute timing. In case of **absolute timing**, the system must share a time base with the environment, for example by using time information from GPS signals. Systems with **relative timing** requirements must internally synchronize clocks so that the system behaves as if a single clock would be available. The absolute value of this clock is not relevant, however.

The concept of **logical time** is used to describe the desired timing of operations. All events between functions are equipped with timestamps describing the logical time for the interaction. The logical time can be incremented by specific model elements to describe the desired timing behavior.

The developer can relate logical time and physical time whenever necessary. Accuracy requirements on initial events determine the minimal accuracy of timestamps (logical time) with respect to physical time that must be achieved. In addition, the developer can specify when events are processed by actors. This is achieved by stating the allowed jitter and thus binding the logical time of the event to physical time.

4.2 Model of Computation

As discussed in Section 3, two models of computation seem to be especially relevant to model systems in a generic way: actor-oriented design [1, 18] with event-based communication and state machines. Since the two most promising approaches PTIDES and AUTOSAR use both event-based execution, TReqS is based similarly on Discrete Event (DE) semantics [17, 8] with the exception that we do support both in- and out-of-order execution ($\models$ R1).

Actors are the active model elements in TReqS. They can have an arbitrary number of associated input and output *ports*, indicated by small boxes at their periphery. Input ports are shown on the left side, while output ports are on the right.

In the DE model, actors – with their input and output ports interconnected by order-preserving channels – communicate via events over these channels. An *event* is a timed token $e = (x, t)$, where x is the *token* (payload) and t is the *timestamp*[1]. The value t specifies the point in (model/logical) time at which the event has occurred respectively should take effect. Note that when two events occur at the same (logical) time, their timestamps will be the same. The DE model of execution relies on the execution of events in a timed fashion: no event can be executed at a specific actor if an event with an earlier timestamp may still arrive at this actor. In our approach, we relax the DE semantics with respect to this aspect: both in-order (using merge actors) and out-of-order execution is supported.

An actor is executed when events are available[2] at its input ports, which are then consumed by the actor. The actor then returns an arbitrary number of events on its output ports. Ports are the only way for an actor to exchange events with its environment. Actors may have an internal state.

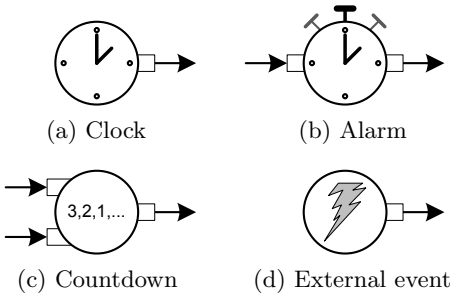
Although execution of an actor takes time in a concrete implementation, an actor does not consume any time in the TReqS model. The same assumption applies for channels: events are delivered instantaneously over a channel (i.e., delivery takes no time from the modeling point of view). Time behavior is specified explicitly using the methods introduced in the next section.

4.3 Primitive Model Elements

TReqS extends the DE model with various primitives to allow intuitive specification of timing requirements in problem space. We also slightly modify the semantics of the timestamp of an event: timestamps represent **desired timing**, which is our understanding of the favored processing time of an event. We will show that our approach is flexible enough to model hard as well as soft real-time systems. It allows the specification of tolerable jitter and the formulation of a metric for soft real-time systems. In the following, the different types of actors are discussed.

[1]The timestamp is just a modeling artifact and may not need to be present in a concrete implementation of the system.

[2]The point in time when events are available depends on physical/logical time and allowed jitter of the event, see Section 4.3. An actor consumes always the event with the earliest timestamp, if several events are available at the actor.



(a) Clock (b) Alarm

(c) Countdown (d) External event

Figure 1: Selection of source actor types

4.3.1 Source Actors

Source actors may produce timed tokens at arbitrary points in time, which is not allowed for actors in general. They have at least one output port and may have input ports. When a source actor fires an event $e = (x, t)$, t is set to the current physical time τ. Since the timestamp cannot be absolutely accurate [16], the required accuracy ε can be specified by the developer for each source actor based on the timing requirements of the application by stating an interval $[\varepsilon_e, \varepsilon_l]$ with $\varepsilon_e \leq 0$ and $0 \leq \varepsilon_l$ ($\models$ R4). To satisfy the accuracy requirement, the system must guarantee that if the triggering event occurs at physical time τ, the following equation holds: $\tau + \varepsilon_e \leq t \leq \tau + \varepsilon_l$. The interval will usually be symmetric ($-\varepsilon_e = \varepsilon_l$).

In addition, the developer can state whether the timestamp can refer to relative time or must be related to absolute time. In the latter case, the system requires externally synchronized clocks, e.g., GPS-based clocks.

It is important to note that very often applications do not have specific requirements on the accuracy of source actors, but instead have requirements on the end-to-end delay, overall jitter or synchronization accuracy in case of distributed sensor sources. In this case, the developer does not need to specify the accuracy of the source actor, but specifies timing requirements according to their root cause. This technique can result in different accuracy requirements for a single source actor. In this case, the system must satisfy the most stringent one.

Event sources are annotated with statistical information about how often and with which probability they output events depending on their input, e.g., periodic/aperiodic, deterministic/spontaneous ($\models$ R3). Requirements on source actors, such as minimal sampling rate, can be specified as well ($\models$ R4).

Typical examples for source actors are shown in Figure 1:

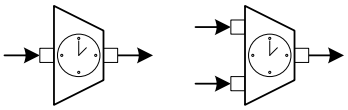
(a) The clock actor emits events at a configurable, fixed frequency.

(b) The alarm actor can be configured to emit events at given points in time.

(c) When it receives an event on its 'enable' input port, the countdown actor emits an event after a configurable time interval c has passed if it does not receive an event on its 'disable' input port during that time interval.

(d) External events may originate from various sources (e.g., interrupts, sensors).

According to our definition of logical time, t is the *desired time* to *process* the event. This can be easily motivated by the fact that many applications would profit from an immediate control reaction to an event. Hence, the desired time to process the event is t, the time at which the event occurred. Since not in every case the desired time for performing the reaction to the initial event triggering the computation is the timestamp of the initial event, our approach allows incrementing the logical timestamp.



(a) Static delay (b) Dynamic delay

Figure 2: Delay actor types

4.3.2 Delay Actors

Delay actors (compare Figure 2) ignore the token x of a received event, but can read and modify its timestamp t ($\models$ R2). Their purpose is to increment the timestamp of a received event by a value $d \geq 0$ and output an event with the incremented timestamp. This makes it possible to shift the desired processing time of an event to some time in the future (hence making the model feasible in real world) and to specify requirements on end-to-end delays ($\models$ R5).

If d is a constant, then the actor is called a **fixed delay actor**. d may also be adjustable, in which case we call the actor a **dynamic delay actor** ($\models$ R7). In the latter case, d is configured via a dedicated input port. Whenever an event $e = (x, t)$ is received on that input port, x with $x \geq 0$ is interpreted as the new delay value and t as the desired time at which the new delay value becomes valid. To make the model analyzable, the developer must state the minimal and maximal delay of the delay actor.

Note that the timestamp per se does not have an impact on the point in time when events are processed by actors. This guarantees maximum freedom concerning the implementation. The desired time/timestamp is only considered when the developer binds the logical time to physical time. As no physical system can operate with infinite precision, binding the logical time to physical time is done by specifying the jitter which is tolerable for the application.

4.3.3 Jitter Intervals

A **jitter interval** $[j_{\min}, j_{\max}]$ can be associated with channels to specify the maximal deviation from desired timing ($\models$ R6). The value 0 must be included in all jitter intervals, because it corresponds exactly to desired timing (i.e., no jitter). Hence $j_{\min} \leq 0 \leq j_{\max}$.

If the jitter interval is specified for the end of a channel, the event $e = (x, t)$ must be processed by the actor at a point in time τ related to physical time satisfying the condition $t + j_{\min} \leq \tau \leq t + j_{\max}$. If the according actor represents an actuator, the event must be processed in a way that τ is the point in time when the output becomes observable by the environment. If no jitter intervals are specified, e can be processed immediately by the actor.

A jitter interval at the beginning of a channel (typically a source actor) specifies timing requirements on the precision of the timestamp of events originating from that actor.

Note that even if jitter intervals specify deferred processing of an event, they have different semantics than delays,

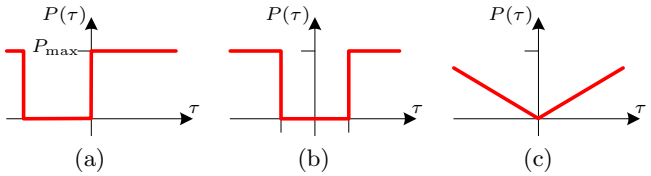


Figure 3: Time/penalty functions; (a) hard real-time system with fixed deadline; (b) hard real-time system with bounded jitter; (c) soft real-time system

since delays refer to desired timing and jitter refers to tolerated deviation. Thus, if a model only uses jitter intervals, but no delay actors, this means that the application should be executed as quickly as possible, but with a bounded maximum latency (specified by the jitter interval on the last channel). Using delay actors however introduces causal constraints on the end-to-end delay between the time of an initial event and the visibility of its effect in the environment. If combined with jitter intervals, the tolerated deviation in the end-to-end delay can be expressed.

4.3.4 Time/Penalty Functions

In most cases, the specification of a jitter interval with hard bounds is only an abstraction of the real requirement. Especially in soft real-time systems, the quality of a system decreases gradually if the jitter increases. Thus, it is useful to specify a metric for the quality of a system (including bounds for tolerable jitter) instead of specifying jitter intervals. To specify this metric, we use a concept similar to Time-Utility-Functions as suggested in Jensen's research group [10]: **time/penalty functions**.

A time/penalty function $P(\tau)$ can be used instead of specifying a jitter interval $[j_{\min}, j_{\max}]$ to specify time constraints on the processing of event $e = (x, t)$. For hard real-time systems, it has the following properties:

$$\forall \tau : P(\tau) \geq 0 \quad \wedge \quad P(\tau) = \begin{cases} 0 & \tau = 0 \\ P_{\max} & \tau - t \leq j_{\min} \\ P_{\max} & \tau - t \geq j_{\max} \end{cases} \qquad (1)$$

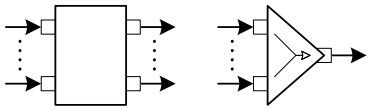
where $P_{\max}$ is the penalty limit. The exact value does not matter in a concrete implementation, but we can assume it to be a very large number ($P_{\max} \approx \infty$). For soft real-time systems, the time/penalty function may not reach the penalty limit (if $j_{\min} \approx -\infty$ and $j_{\max} \approx \infty$). The developer may specify any function that matches (1). A set of standard time/penalty functions is illustrated in Figure 3.

Time/penalty functions directly deliver a metric to evaluate the satisfaction of constraints for a specific implementation ($\models$ R8). The goal of the developer is to minimize the penalty during system implementation. Since different requirements might be contradicting (e.g., the usage of more expensive hardware might reduce the jitter), the metric can be used to find an adequate solution.

It is important to note that Time/Penalty functions may not only be used to specify jitter, but also in the context of delays and accuracy of source actors. An example is the live broadcast application mentioned in the context of R8.

4.3.5 Processing Actors

Processing actors (Figure 4 (a)) cannot produce spontaneous events. They can only produce output events as a reaction to input events and may have an internal state. They are not able to modify the timestamp t of a received event, but are allowed to read it[3]. When they output an event, the timestamp of that event is set to the timestamp of the input event that caused the output event to be generated. Hence, processing actors do not consume logical time.



(a) Processing actor (b) Merge actor

Figure 4: Other actor types

Processing actors are used to represent the functionality of the system. By separating functional behavior (processing actors) and timing behavior (delay actors), the approach is compositional in the sense that adding/refining functions does not change the timing behavior and adding delay actors does not change the functionality (except from timing).

4.3.6 Merge Actors

As mentioned in the beginning of this chapter, our approach relaxes the DE semantics with respect to in-order execution of events. Since some applications rely on this in-order execution, an according actor is introduced. The **merge actor** is responsible for joining events on its input ports onto a single output port (compare Figure 4 (b)). Similarly, it can also operate on parallel channels. In both cases, the merge actor only forwards messages, when these are safe to process, meaning that no message with an earlier time stamp will be forwarded afterwards. Since the timestamp of distributed events cannot be measured absolutely accurate [16], the developer has to specify the required ordering accuracy t_α. Specifying the accuracy at the merge actor rather than at the source actor has the advantage that the requirement can be stated where the root cause lies. The developer can specify the accuracy constraint directly at the merge actor if the root cause is related to synchronization.

Without loss of generality, we discuss the functionality of the merge actor using an example with two events $e_1 = (x_1, t_1)$ and $e_2 = (x_2, t_2)$ emerging from source actors with a physical timestamp τ_1, τ_2. If the merge actor releases the two events in the order e_1, e_2, then the following equation holds: $\tau_1 \leq \tau_2 + t_\alpha$.

4.3.7 Composite Actors

Networks of source, processing, delay (and composite) actors can be hierarchically composed into so-called composite actors. This mechanism is purely of structural nature to enhance readability and does not introduce additional semantics. Note that since composed actors may contain event sources, they may also generate spontaneous events.

4.4 Application of TReqS

A requirements model based on the suggested approach TReqS can be used to determine whether a system is a correct implementation and to compare different implementations.

[3]Only for specific processing sensors: If such an actor is used in the system, the timestamp has to be present in the implementation.

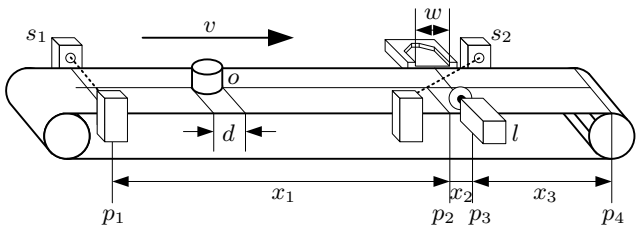
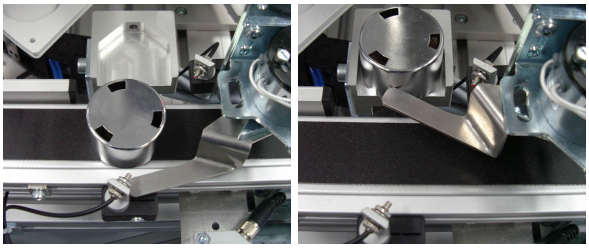


Figure 5: Schematic of conveyor belt (not to scale)



(a) Normal position (b) Pushed position

Figure 6: Detailed view of control lever

DEFINITION 1. *A **scenario** is a fingerprint of a specific execution run of a system. It contains a sequence of inputs events with associated points in time when they happen as well as a description of all execution times induced by the actors on all events.*

Relevant scenarios can be derived from the constraints specified in context of source actors. Scenarios can be weighted according to their probability to appear. Based on a concrete scenario, it is possible to evaluate a concrete implementation:

DEFINITION 2. *The **overall penalty** for a scenario is the sum of all jitter/penalty functions for all events at the point when the respective event arrived.*

A correct implementation must on the one hand assure that no jitter/latency constraint is violated and that on the other hand the overall penalty does not exceed a limit L potentially defined by the requirements analyst:

DEFINITION 3. *A system is a **correct implementation**, if for all possible scenarios of this system and all events, the penalty for each respective event is less than $P_{\max}$ and the overall penalty for each scenario is less than L.*

Based on the metric, implementations can be compared. To compare two implementations with respect to a scenario S, we define an operator $<_S$:

DEFINITION 4. *A correct implementation A is more **accurate** than a correct implementation B for a scenario S, formally $A <_S B$, if the overall penalty of A is smaller than the one of B for that scenario.*

Based on this definition and the set of possible scenarios, two systems implementing the same requirements model can be compared:

DEFINITION 5. *A correct implementation A is more **accurate** than a correct implementation B, if the sum of the overall penalty of all scenarios based on the assumptions regarding events weighted with the probability of the respective scenario is less for A than the respective value for B.*

5. USE CASE

In the following, we will illustrate the suggested modeling approach with an example from the automation domain.

The setup consists of a model of an industrial production system, which was built from Festo components. Amongst other components, the system features a unidirectional conveyor belt over which work pieces are delivered (compare Figures 5 and 6) and put into a processing station. Two optical sensors are used to detect work pieces o on the conveyor belt. For simplification reasons, we assume that only one work piece is transported on the conveyor belt at a time in contrast to the usual pipeline mode. A specification document contains the following textual requirements for the system:

- START: Sensor s_1 is mounted at the beginning of the conveyor belt to detect work pieces that are placed onto the belt by a mobile robot at position p_1. The belt should be started as soon as a work piece is detected by s_1.
- STOP+LEVER: Sensor s_2 is mounted at position p_2 near the end of the conveyor belt just in front of a lever l at position p_3 that can move a work piece into a processing station. If s_2 detects a work piece, the conveyor belt should stop as soon as the work piece is in front of the lever and the lever should push the work piece into the pickup position.
- ERROR_DETECTION: When a work piece is not detected by s_2 within a reasonable amount of time after it was detected by s_1, the belt should be stopped and the user should be notified. This is necessary to prevent the work piece from falling off the conveyor belt at position p_4.

Although the informally defined functionality seems trivial, this specification contains only very coarse (implicit) information about the required timing. To analyze the system more accurately with respect to timing, additional information has to be specified:

- The speed of the conveyor belt is $v = 6.0 \pm 0.25\,\text{cm/s}$. $\Rightarrow v_{\min} = 5.75\,\text{cm/s}$, $v_{\max} = 6.25\,\text{cm/s}$
- The distance x_1 between p_1 and p_2 is 37 cm.
- The distance x_2 between p_2 (intersection point of light barrier and center axis of conveyor belt) and p_3 is 1 cm.
- The diameter of a work piece is $d = 4\,\text{cm}$.
- The width of the pickup position is $w = 4.2\,\text{cm}$.
- The distance x_3 between p_3 and the end of the conveyor belt p_4 is 30 cm.
- The accuracy of the robot placing work pieces at position p_1 is $a = 0.1\,\text{cm}$ in both directions along the axis of the conveyor belt.

A concrete requirements model based on this information and the suggested approach is discussed in the following.

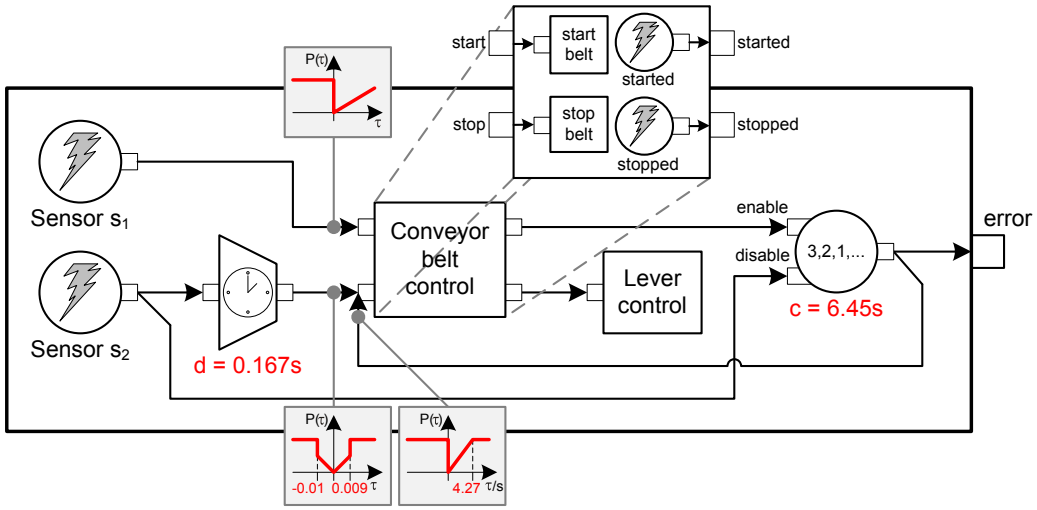


Figure 7: Use case from the automation domain

5.1 Sensor Accuracy

To detect work pieces properly, the sampling rate of sensor s_2 must be $v_{\max}/d = 1.57\,\text{Hz}$. For s_1 no specific accuracy is required due to the assumption that the conveyor belt is not in operation when the work piece is placed on the belt. Note that for specifying timing and jitter, the point in time of the physical event is used as a basis. In the context of light barriers with fixed sampling rates, this point in time would be the moment when an object reaches the barrier. Since a fixed sampling rate implies only limited accuracy, this accuracy must be automatically considered when calculating the maximal jitter/latency bounds of a concrete system. In other words: using a sensor with lower sampling rate puts higher requirements on the jitter/latency of the rest of the system. In the special case of this application, we will see that end-to-end latency requirements will demand a higher accuracy of s_2 than stated above. Using the point in physical time as a basis for the event occurrence has the advantage that for specifying the timing requirements only the end-to-end requirements have to be taken into account. During the implementation process, the developers have to ensure that the individual latency/jitter of the selected components does not contradict the requirements stated in the specification phase.

5.2 Function START

Specification of timing requirements for function START is trivial. The desired timing for starting the conveyor belt after detecting a work piece at p_1 is an immediate start. Hence no delay operator is used for specifying this function. To make the requirement feasible, the allowed jitter has to be specified by using a time/penalty function. In the case of the function START, a linear function is most appropriate, since no concrete deadline is available. This means that penalty continually increases for increased end-to-end delay between s_1 and the conveyor belt.

5.3 Function STOP+LEVER

To specify the timing requirements for this function, the earliest, optimal, and the latest point in time to stop the conveyor belt must be calculated. The optimal point in time to stop the belt is $t_{\text{opt}} = x_2/v_{\text{avg}} = 0.167\,\text{s}$ after the work piece reaches light barrier s_2. Since on both sides a margin of $m = \frac{w-d}{2} = 0.1\,\text{cm}$ exists between the work piece and the boundary of the pickup position, the earliest point in time can be set to $t_{\text{early}} = (x_2 - m)/v_{\min} = 0.157\,\text{s} = t_{\text{opt}} - 0.01\,\text{s}$, the latest point in time to $t_{\text{late}} = (x_2 + m)/v_{\max} = 0.176\,\text{s} = t_{\text{opt}} + 0.009\,\text{s}$. Based on these calculations, an appropriate delay operator (0.167 s) and an according time/penalty function can be described. The command to start the lever has similar characteristics as the START function.

5.4 Function ERROR_DETECTION

Timing requirements of function ERROR_DETECTION can be modeled using a countdown actor. The countdown is loaded with a value related to the earliest point in time t_{err} to detect the error and is activated by the start event of the conveyor belt. t_{err} is derived from the following equation: $t_{\text{err}} = (x_1 + a)/v_{\min} = 6.45\,\text{s}$. The countdown actor is deactivated and reset by an event from sensor s_2. If the countdown expires due to a missing event from sensor s_2, the signal is sent to the error output of the composite actor and to the conveyor belt control actor to stop the belt. The maximal allowed jitter $j_{\text{err,max}}$ at the belt can be derived from the latest point in time to stop the belt before the work piece might fall down: $j_{\text{err,max}} = [(x_1 + x_2 + x_3 - a)/v_{\max}] - t_{\text{err}} = 4,27\,\text{s}$.

The complete application is depicted in Figure 7. The conveyor belt control is a composite actor consisting of one actuator and one sensor for starting and stopping, respectively. The sensors signal the point in time when the belt starts/stops moving.

5.5 Summary

Our experiences with the new modeling approach show that timing requirements are covered in much more detail than with traditional approaches. In particular, the existence of different actors (e.g., delays) and attributes (jitter, minimal sampling rate) and the formal approach force/motivate the developers to clearly specify all timing constraints. The developer benefits in later phases, since the probability of missing requirements with respect to timing is minimized. Furthermore, possible contradicting/overlapping requirements might be found more easily using our approach:

in the discussed example, the original plan was to push the work piece into the pickup position without stopping the conveyor belt. During specification we discovered that this requirement was not feasible in the current hardware setup. An example for overlapping requirements is the accuracy of s_2: the minimal sampling rate of 1.57 Hz, necessary to detect the work pieces properly contributes potential jitter of 0.637 s to all computations triggered by s_2. Hence a higher sampling rate is required to satisfy the jitter requirements of function STOP+LEVER.

6. CONCLUSION

We presented an approach for model-based specification of timing requirements. Based on a systematic analysis of several applications in different domains representing both hard and soft real-time systems, eight basic requirements were identified that must be fulfilled by an approach that can be used during requirement analysis to specify the intended timing of systems. Subsequently, we evaluated existing approaches with respect to these requirements. In summary, several important aspects such as the possibility to specify the allowed jitter at actuators or the required temporal accuracy of sensors are not covered by existing approaches. In addition, no metrics are available to evaluate quantitatively how well a concrete implementation fulfills the requirements. As a consequence, we suggested the approach TReqS, based on the concepts of PTIDES and AUTOSAR, to overcome these issues. The proposed modeling language was evaluated in the context of an example from the automation domain. The main result is a frontloading of the effort: additional effort is spent to create complete and non-contradicting requirements. We expect that developers benefit from this additional effort in the requirements phase during later phases of development.

As future work, we want to apply the approach in complex use cases together with industry to prove the expected benefits. Furthermore, we are already working on a semi-automatic mapping of the models based on TReqS to other models, e.g., PTIDES, to further speed up the development process. Finally, we are intending to combine the suggested approach with timed state machines.

7. REFERENCES

[1] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[2] R. Alur. Timed automata. In *CAV'99*, pages 8–22. Springer Verlag, volume 1633 of LNCS, 1999.

[3] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise, and W. Yi. UPPAAL – now, next, and future. In *Proceedings of Modelling and Verification of Parallel Processes (MOVEP'2k), Nantes, France*, pages 99–124, 2000.

[4] J. M. Atlee and J. Gannon. Analysing timing requirements. In *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, pages 117–127, 1993.

[5] AUTOSAR Administration. Release 4.0. 2009. http://www.autosar.org/.

[6] M. Broy. Two sides of structuring multi-functional software systems: Function hierarchy and component architecture. In *SERA '07: Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[7] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems*. Springer, 2001.

[8] C. G. Cassandras. *Discrete event systems: modelling and performance analysis*. Aksen Associates Inc. Publishers, Homewood, IL and Boston, MA, 1993.

[9] F. Cassez and O.-H. Roux. From time petri nets to timed automata. *Elsevier Science*, pages 1–20, 2004.

[10] R. K. Clark. *Scheduling Dependent Real-Time Activities*. Dissertation, Carnegie Mellon University, 1990.

[11] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan. 2003.

[12] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, Jan. 2003.

[13] J. James E. Coolahan. *Specification of Timing Requirements for Real-time Systems Using Timed Petri Nets*. Dissertation, University of Maryland, University of Maryland, USA, 1985.

[14] R. Johansson and M. Graphics. Deliverable D6 – TADL: Timing augmented description language version 2. In *TIMMO documentation*, pages 1–105, 2009. http://www.timmo.org/.

[15] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In *IEEE RTSS 2003*, pages 1–12, 2003.

[16] H. Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[17] E. A. Lee. Modeling concurrent real-time processes using discrete events. In *Annals of Software Engineering*, volume 7, pages 25–45, 1999.

[18] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

[19] H. K. Lee, W. J. Lee, H. S. Chae, and Y. R. Kwon. Specification and analysis of timing requirements for real-time systems in the CBD approach. *Real-Time Systems*, 36(1-2):135–158, 2007.

[20] OMG MARTE Group. A UML profile for MARTE: Modeling and analysis of real-time embedded systems, beta 2 (convenience document with change bars). In *OMG MARTE documentation*, pages 1–676, 2008.

[21] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[22] Y. Zhao, E. A. Lee, and J. Liu. A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Bellevue, WA, USA, 2007. IEEE.