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 were suggested. However, these approaches focus predominantly on later development phases. Models specifying the requirements with respect to timing without focusing on a specific solution are missing. For example, few models allow the specification of the allowed jitter of a system. In this paper, we identify requirements that are necessary to express 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 to model timing requirements and present an according 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
C.4 [Computer Systems Organization]: Performance of Systems; D.2.1 [Software Engineering]: Requirements/Specifications—timing requirements

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

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 is not addressed adequately in model-driven development processes. Similar to the abstractions made in programming languages [20], several modeling languages do not treat timing as a first-order entity. Whether timing requirements are satisfied in the system is typically only checked during test phases at the end of development. One major reason is that most model-driven approaches applied to embedded systems lack in a systematic and rigorous approach to specify and verify timing requirements at an earlier development stage [18]. Domain-specific languages for real-time systems consider timing, but are predominantly used for the implementation phase, e.g., to enable automatic code generation, and not for requirements analysis. Recently, a number of generic approaches emerged that include timing behavior (e.g., MARTE [19]). However, these approaches focus again 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 hence problems in later development phases.

Based on an analysis of different domains, such as automotive, automation and multimedia, 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 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 specified accurately and completely 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 offers also a quantitative metric on how well the
timing requirements are matched. This is especially useful in the context of soft real-time systems with potentially contradicting requirements, e.g., cost vs. timing. Section 6 concludes this paper and discusses potential future work.

2. REQUIREMENTS

Within this section, the requirements on a solution to model timing requirements are identified. The results are used in the following sections to evaluate existing modeling languages and to derive an appropriate solution.

2.1 Method to Derive the Requirements

The requirements were derived by analyzing timing requirements of applications from different domains. These domains included for the area of predominantly hard real-time systems the automotive, avionics and automation domain, for the area of soft real-time systems amongst others the multimedia domain. Furthermore, different demonstrators used by research groups in real-time systems were evaluated (e.g., inverted pendulum, tunneling ball), since these demonstrators typically represent characteristic problems for different domains. Finally, existing modeling approaches from different domains were evaluated.

2.2 Requirements

One key criterion was that the approach to specify timing requirements does not restrict the solution space for the concrete system. Only by this requirement we can guarantee that when specifying the desired timing behavior, the developer does not seek for solutions, but focuses on the underlying requirements of the system to build.

Requirement 1. The temporal behavior of the system should be specified in a 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, e.g., from using concrete controllers or networks. 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 generic model of computation that does not 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 2. To describe the timing behavior of an application, the interactions between functional components of the system, as well as between these components and the environment must be described.

Requirements are typically described by specifying use cases and the externally observable behavior of the system. Based on these use cases, one can define the system and its core functions. These functions can be treated similar to actors in actor-oriented design [1]. 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 refer to the term event for any of these interactions.

Requirement 3. The developer must be able to specify the characteristics of all initial events triggering computations of the system.

We assume that initial events, e.g., periodic clocks or interrupt sources, trigger the computation of a system. To determine the system load under which the system should still operate, it is necessary to specify according properties of these initial events. Such properties describe amongst others whether events are aperiodic, periodic or occurring according to some pattern. For aperiodic events, e.g., the probability of events and the maximum/minimum distance between two consequent events must be specified. For periodic events, e.g., the rate and the jitter/deviation of the occurrence of these events are relevant.

Requirement 4. 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 determination when exactly observed events occurred 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 µs 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 the object can be observed.

Requirement 5. 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 6. It must be possible to specify the allowed jitter of a system with respect to the desired temporal behavior.

By defining the allowed jitter, it is also possible to distinguish between soft and hard real-time systems. While for hard real-time systems the boundaries between acceptable timing and non-acceptable timing are usually quite tight, soft real-time systems typically can tolerate much longer 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 human’s sense of hearing has a temporal accuracy of 5 µs, the eye’s resolution of perception is limited to 40 ms. Therefore, an audio application consisting of distributed loudspeakers has to fulfil...
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. Therefore, it must be possible to specify also such dynamic real-time systems.

**Requirement 7.** 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 with different velocities. Robots pick up objects on this 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 different requirements might be contradicting and hence it is necessary to find an optimal overall solution.

**Requirement 8.** It must be possible to specify a metric to evaluate how good a concrete implementation matches the requirements.

While in hard real-time systems, the requirements are very often strict in a sense that all requirements must be met, soft real-time systems might be described by soft requirements. In this case, the task of developers is to optimize the system with respect to these requirements. Besides timing requirements, the developers have to take into account also requirements such as costs for this optimization process. This requirement does not only affect the jitter as described in the context of requirement 6, 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, the users do not want to wait too long for videos, especially in the context of simulcast (simultaneous broadcast of multimedia signals that belong together over different transmission channels). 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 will start with an overview over 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) [21] is a programming model for distributed embedded systems based on a global, consistent notion of time. The main idea of PTIDES is to enable the analysis of existing systems. Due to this focus not all requirements stated in Section 2 are matched. Nevertheless, PTIDES is 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 to describe all kinds of real-time systems. However, the strict temporally ordered execution of events at each actor is too restrictive from some applications. 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 triggered the sensor. The accuracy of the timestamp cannot be specified, but depends amongst others on the specific hardware used (≠ Requirement 4). The frequency of sensor events can be specified to enable analysis (|= Requirement 3). Actors can increment the logical time by a specified increment. Dynamic delays (|= Requirement 7) are available as well. The execution semantics of PTIDES requires that output actors process all events before physical time reaches the logical time attached to the event. Therefore, the logical time refers to the deadline, but due to the possible earlier execution not to the desired timing (≠ Requirement 5). Hence, jitter or a metric are not considered (≠ Requirements 6, 8). Another issue is the interpretation of network components within a distributed system as output actors. This simplifies analysis as only the computation on the individual nodes must be considered for analysis. Yet, it has the effect that end-to-end delays for a computation within a distributed system must be divided into several delays to satisfy the timing constraints of the network components. For distributing these delays, the developer requires explicit information, such as worst-case execution times (≠ Requirement 1).

**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, i.e., it allows platform-independent timing annotations. The relation to physical time is guaranteed at the interaction points between actors and hence can be interpreted as desired timing (|= Requirement 5). However, the Giotto approach does not satisfy all requirements. Due to its time triggered nature, it is generally not suitable for event-triggered systems (≠ Requirement 1). Further, it does not address any metrics and allows neither dynamic delays nor the specification of jitter (≠ Requirements 6–8).

**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 reaction time. The
time scale is presented in terms of ticks. Every tick, inputs are read and outputs are generated instantaneously. The precision of the time scale is limited to the minimal interval between two input events (length of a tick), which has to be large enough to ensure the synchrony assumption (\(\neq\) Requirement 4). Jitter and dynamic delays are not addressed (\(\neq\) Requirements 6, 7). The very strict model of computation simplifies the analysis of real-time systems considerably, but restricts the solution space regarding the implementation (\(\neq\) Requirement 1).

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. The parallel composition of automata can be described and the synchronization mechanisms are specified through signals, for example. They 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. Formal verification techniques are well elaborated for timed automata [3].

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

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 (\(\neq\) Requirement 5, 6). 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 (\(\neq\) Requirement 7).

Finally, Timed Automata do not allow the specification of a metric to evaluate the degree of satisfaction of a system with respect to its timing requirements (\(\neq\) Requirement 8).

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 by time discrete data streams. The communication is instantaneous, directed, reliable and data preserving. Every component is specified as a stream processing function: a mapping of the input stream to a set of possible output streams. By allowing the specification of several possible output streams and hence non-determinism, also temporal non-determinism can be specified.

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 (\(\equiv\) Requirement 1). Similar to timed automata, the approach does not distinguish between desired timing and jitter (\(\neq\) Requirement 5, 6). The environment and related requirements, dynamic delays and a metric are not considered (\(\neq\) Requirements 3–8).

3.3 MARTE

MARTE [19] is a UML profile describing a general framework for representing time and related concepts. MARTE introduces a domain view for time modeling and defines standard UML elements to express defined timing concepts. The approach targets modeling real-time and embedded systems. MARTE has very elaborated concepts for timing specification, providing logical/chronometric (related to physical time) clocks, single/time bases, and timing constraints for single events/time bases. The focus of these constructs is to specify a specific system (e.g., the timing properties of a specific clock) and hence rather suitable for later development phases (\(\neq\) Requirement 1). Important timing aspects relevant for the analysis phase, required accuracy of sensor data, jitter, dynamic delays or metrics are not covered by the profile (\(\neq\) Requirements 4, 6–8).

3.4 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) [5] is an open industry standard for automotive E/E architectures developed in partnership between automotive manufacturers and suppliers.

In the current AUTOSAR release 4.0, new timing extensions developed in the TIMMO project [14] are introduced. The extension is based on events as basic entity (\(\equiv\) Requirement 1). 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 (\(\equiv\) Requirement 3); delay constraints, i.e., age (the latest response to the latest, the most updated data), reaction (the first reaction to the first data) and timing constraints (\(\equiv\) Requirement 5); and input and output synchronization constraints (meaning that referred input or output events have to be synchronized).

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 (\(\neq\) Requirements 6, 7). Furthermore, no mean for specifying a metric with respect to the satisfaction of timing requirements is available (\(\neq\) Requirement 8).

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 addressed generally, while other requirements, e.g., the necessity of a metric, are not addressed at all.

The most promising approach is to combine and extend the approaches PTIDES and AUTOSAR based on event-based specification of systems.
<table>
<thead>
<tr>
<th>Req. 1</th>
<th>Req. 2</th>
<th>Req. 3</th>
<th>Req. 4</th>
<th>Req. 5</th>
<th>Req. 6</th>
<th>Req. 7</th>
<th>Req. 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Space</td>
<td>Component Based</td>
<td>Initial Events</td>
<td>Accuracy</td>
<td>Desired Time only deadline</td>
<td>Jitter</td>
<td>Dynamic Delay</td>
<td>Metric</td>
</tr>
<tr>
<td>PTIDES</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Giotto</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>limited</td>
</tr>
<tr>
<td>Synchronous Languages</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>implicit</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Timed Automata</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>mixed with Req. 6</td>
<td>mixed with Req. 5</td>
<td>limited</td>
</tr>
<tr>
<td>FOCUS</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>mixed with Req. 6</td>
<td>mixed with Req. 5</td>
<td>no</td>
</tr>
<tr>
<td>MARTE</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 1: Evaluation Results

4. TIMING REQUIREMENTS SPECIFICATION (TREQS)

Based on the identified requirements and relevant concepts from related work, this section presents TReqS, an approach for timing requirements specification.

4.1 Time

TReqS uses analogously to PTIDES two different notions of time: logical time and physical time.

**Physical time** relates to the time notion of external observers and is hence 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, e.g., by using time 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 however not relevant.

To describe the desired timing of operations, the concept of **logical time** is used. 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] 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 not require an execution order based on timestamps (≡ Requirement 1).

Within TReqS, **Actors** are the active model elements. 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 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 ($\models$ Requirement 2). The value $t$ specifies the point in (model/logical) time at which the event has occurred respectively should be executed. 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, since some applications do not require an in-order execution.

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 amount of events on its output ports. Ports are the only legal 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 from the modeling point of view. 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

We extend the DE semantics with various primitives to allow intuitive specification of timing requirements in prob-

\(^1\)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 time, logical time and allowed jitter of the event, see Section 4.3.
be absolutely accurate \[16\], the required accuracy have specific requirements on the accuracy of source actors, synchronized clocks, e.g., GPS-based clocks. The timestamp can refer to relative time or must be related to absolute time. As no physical system can operate with infinite precision, 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.

### 4.3.1 Source Actors

Source actors are allowed to 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 also have input ports. To facilitate analysis, event sources are annotated with statistical information about how often and with which probability they produce output events depending on their input, e.g., periodic/aperiodic, deterministic/spontaneous (\[\implies\] Requirement 3). In addition, also requirements on source actors, such as minimal sampling rate, can be specified (\[\implies\] Requirement 4). 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).

When a source actor fires an event \(e = (x, t)\), \(t\) is set to the current physical time \(\tau\). Since the timestamp cannot be absolutely accurate \[16\], the required accuracy \(\varepsilon\) can be specified by the developer for each source actor based on the timing requirements of the application (\[\implies\] Requirement 4) by stating an interval \([\varepsilon_x, \varepsilon_t]\) with \(\varepsilon_x \leq 0\) and \(0 \leq \varepsilon_t\). To satisfy the accuracy requirement, the system must guarantee that if the triggering event occurs at physical time \(\tau\), the following equation holds: \(\tau + \varepsilon_x \leq t \leq \tau + \varepsilon_t\). In general, the interval will be symmetric (\(-\varepsilon_x = \varepsilon_t\)).

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.

### 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 \(t\). Their sole 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 (\[\implies\] Requirement 5).

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 (\[\implies\] Requirement 7). 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 has in first instance no impact on the point in time when events are processed by actors. This guarantees a maximum of 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 pre-
cision, 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 \([j_{\text{min}}, j_{\text{max}}]\) can be associated with channels to specify the maximal deviation from desired timing (\(\equiv\) Requirement 6). The value 0 must be included in all jitter intervals, because it corresponds exactly to desired timing (i.e., no jitter). Hence \(j_{\text{min}} \leq 0 \leq j_{\text{max}}\). 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 \(t\) related to physical time satisfying the following condition: \(t + j_{\text{min}} \leq \tau \leq t + j_{\text{max}}\). If the according actor represents an actuator, the event must be processed in a way that \(\tau\) is the point in time when the output becomes observable by the environment.

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 source actor.

Note that even if jitter intervals specify deferred processing of an event, they have different semantics than delays 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 model should be executed as fast as possible, but with a bounded maximum latency (specified by the jitter interval on the last channel).

When delay actors are used, this means that the processing conditions for each event should somehow be synchronized so that the effects of an event are always visible at almost (depending on the jitter intervals) the same relative time compared to their cause.

If no jitter intervals are specified, the event can be processed immediately by the actor.

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. In fact, in many systems and especially in soft real-time systems, the quality of a system decreases if the jitter increases. Thus, it is useful to not specify jitter intervals, but a metric that specifies the quality of a system (including bounds for tolerable jitter). To specify this metric, we use a concept analog 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 \([j_{\text{min}}, j_{\text{max}}]\) to specify time constraints on the processing of event \(e = (x, t)\). For hard real-time systems, it has the following properties:

\[
\forall \tau : P(\tau) \geq 0 \quad \land \quad P(\tau) = \begin{cases} 
0 & \tau = 0 \\
P_{\text{max}} & \tau - t \leq j_{\text{min}} \\
0 & \tau - t \geq j_{\text{max}}
\end{cases}
\] (1)

where \(P_{\text{max}}\) 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 (\(P_{\text{max}} \approx \infty\)). For soft real-time systems, the time/penalty function may not reach the penalty limit (if \(j_{\text{min}} \approx -\infty\) and \(j_{\text{max}} \approx \infty\)). The developer may specify any function that matches (1). A set of standard time/penalty functions is illustrated in Figure 3.

4.3.5 Processing Actors

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

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

Our approach relaxes the DE semantics with respect to inorder execution of events, as mentioned in the beginning of this chapter. 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)). It can be configured to respect a certain ordering accuracy \(t_o\) on the timestamps of the events. 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. If the root cause is of synchronization nature, then the developer can specify the accuracy constraint directly at the merge actor.

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 \(t_1, t_2\). If the merge actor releases the two events in the order \(e_1, e_2\), then the following equation holds: \(t_1 \leq t_2 + t_o\).

4.3.7 Composite Actors

Networks of source actors, processing actors, delay actors (and composite actors) can be hierarchically composed.

\(^3\)Only for specific processing sensors: If such an actor is used within the system, the timestamp has to be present in the implementation.
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.

**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 \( P_{\text{max}} \) and the overall penalty for each scenario is less than \( L \).

Based on the metric, it is also possible to compare two implementations. 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 a 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. A previous textual specification document contains the following requirements of 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 (e.g., by a mobile robot) at position \( p_1 \). The belt should be started as soon as a work piece is detected by \( s_1 \).
- **LEVER_STOP:** 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:** It should be possible to detect errors, e.g., when a work piece is not detected by \( s_2 \) within a given amount of time after it was detected by \( s_1 \) to prevent the work piece from falling off the conveyor belt at position \( p_4 \). In case of an error, the belt should be stopped and the user should be notified.

For simplification reasons, we assume that only one work piece is transported on the conveyor belt at the same time. Although this informally defined functionality seems trivial, the specification does only contain very coarse (implicit) information about the required timing. To analyze the system with respect to timing more accurately, additional information must be specified:

- The speed of the conveyor belt is \( v = 6.0 \pm 0.25 \text{ cm/s} \).
  \[ \Rightarrow v_{\text{min}} = 5.75 \text{ cm/s} \text{ and } v_{\text{max}} = 6.25 \text{ cm/s} \]
- 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.
5.1 Sensor Accuracy

To detect work pieces properly, the sampling rate of sensor s2 must be

\[ v_{\text{max}}/d = 1.57 \text{ Hz}. \]

For s1 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 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 the maximal jitter/latency bounds of a concrete system. In the special case of this application, we will see that end-to-end latency requirements will demand a better accuracy of s2 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

The timing requirements of function START are quite simple to specify. The desired timing for starting the conveyor belt after detecting a work piece at p1 is an immediate start. Hence no delay operator is used for specifying this function. To make the requirement feasible, the allowed jitter must 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.

5.3 Function LEVER_STOP

To specify the timing requirements of 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 s2. Since on both sides a margin of

\[ m = w/d = 0.1 \text{ cm} \]

exists between the work piece and the boundary of the pickup position, the earliest point in time can be set to

\[ t_{\text{early}} = (x_2 - m)/v_{\text{min}} = 0.157 \text{ s} \]

The latest point in time to

\[ t_{\text{late}} = (x_2 + m)/v_{\text{max}} = 0.176 \text{ s} \]

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

\[ t_{\text{err}} \]

to detect the error and is activated by the start event of the conveyor belt.\[ t_{\text{err}} \]is derived from the following equation:

\[ t_{\text{err}} = (x_2 + a)/v_{\text{min}} = 6.45 \text{ s}. \]

The countdown actor is deactivated and reset by an event from sensor s2. If the countdown expires due to a missing event from sensor s2, 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 \( f_{\text{err,max}} \) at the belt can be derived from the latest point in time to stop the belt before the work piece might fall down:

\[ f_{\text{err,max}} = (x_1 + x_2 + x_3 - a)/v_{\text{max}} \] - \[ t_{\text{err}} = 4.27 \text{ s.} \]

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 require-
ments 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 LEVER_STOP.

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 the approaches PTIDES and AUTOSAR, we proposed a modeling language for specification of timing requirements to overcome these issues. The application of the approach was discussed 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 state machines.

7. REFERENCES


