Abstract

Today, an increasing demand for high quality real-time software for complex, safety-critical systems results from the fact that more ambitious and complex technical systems are built. In the development of such systems, a crucial step is the design and formal verification of the real-time interaction of the components to ensure the correctness and safe operation of the overall system. We propose to ease this cumbersome design step by supporting the analysis of conflicts between timed scenarios and the scenario-based synthesis of the required component behavior. Such a synthesis procedure, which requires as its input a set of timed scenarios, results in a set of Statecharts that realize the given scenarios. By supporting even parameterized timed scenarios, the approach permits to avoid too early decisions on specific values for the time constraints while still being able to check them. The paper describes the application of the approach and the available prototypical tool support by means of a running example.

1. Introduction

The ever increasing complexity of technical systems and their software leads to a demand for automated support for the production of high quality real-time software in this domain. During the early design phase, a number of scenarios are usually developed to identify and describe possible or required interaction behavior of the system components and its software. Then the manual design of the operational model of the system interaction (usually in form of some sort of state machines) is a crucial and costly next step.

We therefore propose to support and automate the design of the state-based model of the interaction using scenario-based synthesis. Several approaches exist which permit to use the information provided by a given set of scenarios to synthesize the operational state-based behavior of the components (cf. [12, 14, 18, 20]). In the considered real-time domain, however, the time constraints within the scenario descriptions are an essential part besides the synthesis of the resulting state model. Timing information such as worst-case execution times (wcet), deadlines, or timeouts have thus to be integrated when attempting to synthesize operational models from timed scenarios. The situation is even more difficult, as we intend to support parameterized time constraints to enable their step-wise refinement. Otherwise, the final values for all time constraints have to be set upfront during the design while in practice the analysis of the trade-offs between different alternative solutions is required.

Several proposals to check timed system models against scenario descriptions with time have been made (cf. [13]). For the scenario-based synthesis including time constraints, in contrast, only very restricted proposals exist today. The approach proposed in [17] synthesizes only global solutions in form of a single automaton for non-parameterized scenarios. It employs a timed automata dialect which is not sufficient to describe the intended operational behavior, because angelic non-determinism is assumed and no concept for specifying progress conditions is supported. In [5, 8] the play-out of life sequence charts (LSC) with timers is presented. However, the play-out engine also constructs only a global behavior for non-parameterized LSCs.

We present in this paper the results of a prototype implementation of a scenario-based analysis and synthesis tool for parameterized timed scenarios which enables to derive a distributed realization. It realizes and extends the concepts proposed in [6]. In addition to the original proposal, conditional behavior within the sequence diagrams is supported and UML 2.0 sequence diagrams and a real-time extension of statecharts are considered as input and output. The supported operational target model are real-time statecharts (RTSC) [4], which are supported by the real-time ver-
sion of the Fujaba CASE tool. RTSC have a well-defined real-time semantics based on timed automata [9]. Code generation [4] as well as real-time model checking [10] are currently supported.

The structure of the paper is as follows: In Section 2, we describe how a subset of the UML 2.0 sequence diagrams is used to describe the required system behavior of our example and how the timing constraints are formalized. We then outline how consistency and causality for a given set of formal time constraints are checked and how alternative parameter setting are systematically studied in Section 3. Then, in Section 4, the employed notation of parameterized real-time statecharts is introduced. The synthesis of parameterized real-time statecharts and further possible analysis activities are sketched in Section 5. We close the paper with a final conclusion and an outlook on future work.

2. Sequence Diagrams with Time and Triggers

A number of notations for scenario description techniques with timing constraints exists, such as UML 1.4 sequence diagrams [13], message sequence charts with timing constraints [3], life sequence charts (LSC) with timers [5, 8], or action diagrams (timing diagrams) [11]. In the different notations, timing constraints are described in quite different ways (cf. [3, 8]). Timers with reset and timeouts, delay intervals for events and activities, drawing rules, or timing markers in form of boolean expressions that constrain particular events or the whole diagram are possible options.

In this paper we consider only a restricted subset of the UML 2.0 sequence diagrams [15. p. 444] to describe parameterized timed scenarios. The UML 2.0 sequence diagrams permit a duration observation of the form \( d = \text{duration} \), which can be used to measure the time required for a specific message transfer. In-between two points on the lifeline of a UML sequence diagram, a duration constraint \( \{t..u\} \) with a lower and an upper bound can be specified. Additionally, we can store the current time within a time observation \( t = \text{now} \) and later reference this measurement using a time constraint \( \{t..t+3\} \).

Another relevant aspect of scenarios employed in the paper are techniques for conditional behavior such as triggers for sequence diagrams. They have been first proposed for life sequence charts (LSC) [5]. Another proposal is triggered message sequence charts (TMSCs) [16]. In this paper, we use the assert block of the UML 2.0 sequence diagrams [15, p. 444] to describe the conditional behavior of parameterized timed scenarios. An assert block describes that a specific part of the scenario is mandatory when the preceding part of the scenario has occurred. While the sequence of events before the assert block may happen or not, the assert block is required to happen once the preceding sequence of steps has been observed.

To resolve conflicts between different asserted behaviors, we further assign priorities to each sequence diagram. Therefore, the synthesis procedure will automatically resolve conflicts between scenarios with different priority. A conflict between multiple scenarios with the same priority will in contrast result in non-deterministic behavior.

Our running example, employed in the following to outline the application of the synthesis approach, describes the typical behavior of a controller and related watchdog (cf. Figure 1). It includes the initialization, regular operation, and shutdown as well as the emergency reaction of the watchdog, when no heart beat has been received within the specified time frame.

The initialization of the watchdog after the controller is put online is described in the sequence diagram initialization. The regular behavior, which essentially describes that the controller periodically sends his heart beat (sane message) to the watchdog, is depicted in the sequence diagram Regular. To shutdown the watchdog, a stop message is send to the watchdog which subsequently goes offline as depicted in Shutdown sequence diagram. Finally, the sequence diagram Reset specifies that the watchdog will send a reset to the controller if no sane message is received within the time frame \( \{WTL .. wtu\} \).

Checking consistency for the parameterized timed scenarios for arbitrary constraints is obviously not feasible. Thus, we restrict the annotations of the sequence diagrams as follows: For variables/parameters \( v \in R \), constants \( A \in IR \), and expressions \( \text{expr} \) of the form \( A + \sum_{i \in I} v_i \), a duration constraint must have the form \( \{A_1 \ldots \text{expr}_j\} \). For a time constraint, the condition must hold that a related time observation \( t = \text{now} \) exists such that the constraint is equivalent to the form \( \{t + A_1 \ldots t + \text{expr}_j\} \).

To formalize the timing constraints, we define a time constraint graph (TCG) as a pair \((N, E)\) with \( N = N_p \cup N_a \) of sets of nodes which are either possible \((N_p)\) or asserted \((N_a)\) and \( E = E_a \cup E_t \cup E_w \cup E_r \cup E_c \) the set of edges (see Figure 2 for an example). The possible nodes are depicted by grey nodes while the asserted nodes are white. For edges, we distinguish five different kinds with lower and upper bound expressions \( e_l \) and \( e_u \): (1) Activity edges which describe a possible delay caused by local computation denoted by \((e_l, e_u, a, n') \in E_a\). (2) Timer edges which describe a possible wait for a local trigger denoted by \((n, e_l, e_u, t, n') \in E_t\) (We further refer to edges which are either activity or triggered edges as local ones, denoted by \(l\)). (3) Communication between different processes is denoted by a wait edge \((n, e_l, e_u, w, n') \in E_w\) which leads to a shared node representing the successful synchronous communication. (4) Restrictions within a single process which enforce the specified constraint are modeled by an.
Figure 1. Scenarios of initialization, regular operation, shutdown, and reset

edge \((n, e_l, e_u, r, n') \in E_r\) (5) If a specific constraint should only be checked but not enforced, a check edge \((n, e_l, e_u, c, n') \in E_c\) has to be used. When visualizing the graph, we label the arc simply using the lower and upper bound expression as well as the type (cf. Figure 2).

The nodes which relate to a state within an assert block in the sequence diagram become assert nodes \((N_a)\) if they have any predecessor node. All other nodes become possible nodes \((N_p)\).

Due to the outlined distinction between possible and asserted nodes we distinguish the following cases for transitions: A transition is \textit{possible} iff its source and target nodes are possible. A transition is \textit{asserted} iff its source node is asserted. A transition leading from a possible to an asserted node is a \textit{trigger}.

The outlined restriction for the UML 2.0 sequence diagrams essentially ensure that all time observations can be mapped to edges of a TCG. An example for such a mapping is presented in Figure 2, where the TCG of the shutdown scenario of Figure 1 is presented.

Besides the controller and watchdog instances of the sequence diagrams, we add the behavior of the channel in between them. For the communication channel, the behavior of a blocking channel with a parameterized delay and fixed buffer size 1 is simply added.

Activities on the lifeline of a sequence diagram are mapped to activity edges with a fixed minimal lower bound \(d\) and a freely chosen upper bound variable (e.g., \(I_{\text{of}}\)). Steps without an activity are mapped accordingly to a timer edge. The minimal time duration \(d\) is added to the lower bound to ensure that no infinite fast execution of transitions is assumed (\(d_{\text{con}}\) is the standard delay of the controller and \(d_{\text{wa}}\) is the standard delay for the watchdog).

Communication is mapped to specific synchronization nodes in-between sender and channel as well as channel and receiver. Wait edges lead to these synchronization nodes. For the additional edges leading back from these synchronization nodes, activity edges with lower and upper bounds set to zero are used. Within the channel, a channel delay is denoted by an activity edge.

All constraints which refer to two points on the lifeline of the same process are mapped to restriction edges using the variables or constants specified within the sequence diagram (see edge \textit{shutdown Time}). A constraint which involves two different processes is mapped to a check edge. Again, the variables or constants specified within the sequence diagram (see edge \textit{messageDeliveryTime}) are used in the TCG edge.

\[\begin{align*}
\text{Figure 1. Scenarios of initialization, regular operation, shutdown, and reset}
\end{align*}\]
3. Analysis

For non-parameterized timed scenarios, a serious problem is consistency. A scenario is only consistent when a time consistent behavior for it exists. In [2], an efficient decision procedure for this problem has been presented, which does, however, not take into account that the timing of parallel tasks can in reality only depend on their interaction in the past. The notion of causality [11] thus demands that a realization exists that ensures correct operation without global knowledge or knowledge about the future.

For the case of parameterized timing constraints, we even have to consider the dependencies which result from parameters which are present in multiple scenarios. To address the problem of consistency and causality, we use linear inequalities as proposed in [6]. For \( n \) variables \( v_1, \ldots, v_n \) over \( IR \) a system of linear inequalities is defined by \( m \) inequalities with coefficients \( \alpha_{i,j} \in IR \) of the form \( \alpha_{i,1} v_1 + \ldots + \alpha_{i,n} v_n \leq b_i \). A set of linear inequalities is feasible iff an assignment for all \( v_i \) exists which fulfills all \( m \) inequalities.

The basic idea is to encode the propagation of the upper and lower bounds through a TCG using constants, variables, and sums over variables by assigning bounds \((A, b)\) to each node. For each specific edge leading from a node \( n \) with the assigned bounds \((A, b)\) to a node \( n' \), we proceed as follows: (1) In case of an activity edge \((n, L, u, a, n')\) or timer edge \((n, L, u, t, n')\) we assign \((A + L, b + u)\) to \( n' \). If two edges leading from nodes \( n \) and \( n' \) with assigned bounds \((A, b)\) and \((A', b')\) to a node \( n'' \), we additionally proceed as follows: (2) For two waiting edges \((n, L, u, w, n'')\) and \((n', L', u', w, n'')\) we add a new variable \( v_n \) and two inequalities \( v_n \geq b + L \) and \( v_n \geq b' + L' \) and finally assign \((\max{(A + L, A' + L')}, b + u)\) to \( n'' \). (3) In the case of an activity edge \((n, L, u, a, n'')\) or timer edge \((n, L, u, t, n'')\) and a restriction edge \((n', L', u', r, n'')\) we restrict the lower bound using the propagation and propagate the upper bound of the local edge by assigning \((\max{(A + L, A' + L')}, b + u)\) to \( n'' \). (4) Finally for an activity edge \((n, L, u, a, n'')\) or timer edge \((n, L, u, t, n'')\) and a check edge \((n', L', u', c, n'')\) leading from \( n \) resp. \( n' \) to \( n'' \) we only propagate the local edge and simply assign \((A + L, b + u)\) to \( n'' \).

To check for consistency, we have to derive a set of inequalities, using the above described propagation for all subgraphs between the start and end node of restriction and check edges. In addition, for each communication by means of two wait edges, the related preceding subgraphs (which start with a single node and reach this node without ever having a front with less than two nodes) has to be considered to detect possible timing problems. For each of these subgraphs we start the propagation by assigning \((0,0)\) as the upper and lower bound to the initial node. Then the propagation outlined above is employed to derive a set of inequalities. Note that due to the occurrence of the same parameters within the different subgraphs the resulting inequalities are also related.

Besides the propagation, we have to additionally take into account the following inequalities for the above considered cases (2), (3), or (4) for a combination of edges leading to the final node: (2) For two waiting edges \((n, L, u, w, n'')\) and \((n', L', u', w, n'')\) with \((A, b)\) and \((A', b')\) assigned to \( n \) resp. \( n' \), we additionally add \( A + u \geq b' + L' \) and \( A' + u' \geq b + L \) to the set of inequalities. (3) In the case of an activity edge \((n, L, u, a, n'')\) or timer edge \((n, L, u, t, n'')\) and a restriction edge \((n', L', u', r, n'')\) with \((A, b)\) and \((0,0)\) assigned to \( n' \), we additionally add \( b + u \leq u' \) to check the upper bound while the lower bound is guaranteed by the restriction itself. (4) Finally for an activity edge \((n, L, u, a, n'')\) or timer edge \((n, L, u, t, n'')\) and a check edge \((n', L', u', c, n'')\) with \((A, b)\) and \((0,0)\) assigned to \( n'' \), we additionally add \( A + L \geq L' \) for the lower bound and \( b + u \leq u' \) for the upper bound.

After deriving all these sets of inequalities, we finally combine them and employ the simplex algorithm to check.
whether a solution is feasible. If such a feasible solution exists, we have found a witness for the consistency of the different timed scenarios which also respects causality (cf. [11]). For our example, the Java-based tool prototype requires only 0.28 sec. on a standard PC to generate the linear inequalities for 7 subgraphs, which result in an overall set of 107 linear inequalities. Solving the system of linear inequalities took only 0.08 sec. Checking consistency and therefore ensuring that the different timed scenarios are not contradicting each other w.r.t. the used time parameters is, however, not the only benefit of the outlined procedure.

During the further design, restrictions on the valid parameter set may be identified. For example, an upper bound on the detection of the timeout and initiation of a controller reset by the watchdog of the form RDU + wtu ≤ 1000 may be added. The outlined approach then permits to check whether a consistent and causal solution for the timed scenarios and this linear inequality exists.

For each parameter \( v \), we use the optimization function \( f(\vec{v}) = v \) (resp. \( f(\vec{v}) = -v \)) for the simplex algorithm to determine its maximum \( v_{\text{max}} \) (resp. minimum \( v_{\text{min}} \)) for the derived set of inequalities. For the values \( d_{\text{con}} = 5 \), \( d_{\text{ch}} = 20 \), \( d_{\text{ua}} = 10 \) for the minimal transition times and lower bound for the reset timeout \( WTL = 500 \), upper bound for the reset timeout \( wtu = 600 \), the upper bound for the controller initialization time \( ITU = 50 \), the upper bound for the message delivery \( MDU = 350 \), the upper bound for realizing the reset \( RDU = 300 \), and upper bound for the shutdown time \( SDU = 100 \), we get the following minima and maxima: \( \text{lsnOf}_{\text{min}} = 5 \), \( \text{lsnOf}_{\text{max}} = 95 \), \( \text{bd}_{\text{u}}_{\text{min}} = 20 \), \( \text{bd}_{\text{u}}_{\text{max}} = 260 \), \( WCU_{\text{min}} = 10 \), and \( WCU_{\text{max}} = \infty \).

In addition, we should estimate lower bounds for each parameter \( v \) to avoid that the optimization proposes unrealistically fast behavior. This is done by adding additional constraints to the set of linear inequalities. Checking the resulting system of linear inequalities ensures that infeasibility is detected immediately.

If appropriate weights \( \beta_i \) (approximating the relative costs for the implementation) for the different time bound parameters \( v \) are given, reasonable suggestions for the time bounds can be computed using a solver for concave quadratic maximization problems. For \( V' = \{ v\mid v_{\text{max}} \neq \infty \wedge 1 \leq i \leq n \} \) the set of all parameters with a finite maximum, we can compute \( \max = \max\{ v_{\text{max}} \mid v \in V' \} \). The optimization function which has to be maximized is then \( \sum_{v \in V'} \max \beta_i v - \beta_i (v)^2 \), which is a concave quadratic problem that is monotonically increasing for any \( v \in V' \).

### 4. Parameterized Real-Time Statecharts

Finally, we want to derive an operational description of the real-time behavior by means of statecharts. UML 2.0 statecharts do, however, not provide appropriate modeling concepts to realize real-time constraints. The \texttt{when} and \texttt{after} constructs permit only to refer to absolute time and relative to the point in time when entering a state, respectively. If more complex time constraints which involve a whole sequence of states are considered, these concepts are not sufficient any more.

Thus, we use the extensions of standard UML statecharts provided by real-time statecharts (RTSC) [4]. A first extension is an advanced time model which provides clocks which can be reset when firing transitions or entering/leaving states. The clocks can then be measured to describe the time-dependent behavior of the statechart. In addition, each state has time-invariants which determine the permitted clock values for that state. For example, the state \texttt{initialReset} in Figure 3 has the time invariant \( t_0 \leq w\text{WaChR} \). The transitions have additional time guards which denote the valid activation times w.r.t. the current values of the clocks. In the example, the time guard of the outgoing transition of state \texttt{stateReset} is \( [t_0 \leq w\text{WaChR}] \).

A RTSC finally becomes a parameterized RTSC simply by permitting the clock constraints to contain besides constants also parameters.

Two different types of transitions exist. \textit{Urgent} transitions (solid line) fire immediately when they are triggered. \textit{Non-urgent} transitions (dashed line) in contrast can fire arbitrarily as long as they are enabled. Only when the source-state’s invariant would otherwise become false, the non-urgent transition must be executed.\(^3\) The execution of transitions, in contrast to standard timed automata models, consumes time. Therefore, the transitions are also equipped with deadlines which describe when the transition execution has to finish. The deadline can be specified relative to the point in time when the transition has been fired and absolute by referring to clocks. In the example, the outgoing transition of state \texttt{receivedSaneWatchdog} has to be finished before \( t_0 \equiv \text{WCU} \) and before the time \( \text{WCU} \) passed after triggering. It may not be finish before the time \( d_{\text{wa}} \) passed after triggering.

### 5. Synthesis

Using the TCG, we can also derive the synthesis result by identifying related states of each instance within the scenarios. As the mapping ensures that non local constraints are only checked but not enforced, we can in contrast to our earlier proposal [6] synthesize a distributed operational model which does not require that non local deadline information can be exchanged.

Like most proposed approaches for the synthesis of a distributed implementation (cf. [12, 18, 20]), we only consider the local knowledge present for each role in the scenarios. Therefore, our synthesis algorithm is efficient but incom-

---

\(^3\)This property is used within our approach to denote underspecification of the timing behavior where required.
complete and implied scenarios (cf. [19]) can occur in the overall behavior. If the logical flow of events of a given timed scenario are observed for the synthesized set of RTSC, the synthesis algorithm ensures that the specified timing constraints of the scenario are met. We have, however, to check additionally whether the synthesis result excludes (time-stopping) deadlocks and whether unwanted implied behavior exists.

For the mapping of states, we currently only support a single state, sets of possible states, or the whole state set denoted by * within the sequence diagrams. They are simply mapped to the specified state set. For timer edges, we add related non urgent transitions, while the activity edges become urgent ones. The communication expressed by a wait edge, a synchronization node, and a local processing step is collapsed into a single urgent communication transition in both related processes. For a local edge the time conditions simply result in a time guard and deadline such that the specified timing behavior is possible. In a similar fashion, the mapping for the wait edges results in a time guard which ensures the lower and upper time bound.

The more interesting aspects of the synthesis are, however, the treatment of timing checks and restrictions. Timing constraints are established as outlined in the preceding section. The restrictions limit the possible timing and therefore have to be ensured by the synthesized real-time statecharts. At first, we have to ensure the restriction present for one edge, which is described by the upper and lower time bounds on the TCG edges. They are addressed by referring to a special clock $t_0$ which is reset when entering any state.

In the more complex case of restrictions which do affect a series of TCG edges, we use an additional clock which is set to zero when the initial node of the restriction edge is entered. This clock is finally used to adjust the time guards of the last transition of this series of TCG edges such that the overall restriction is enforced for the lower bound. Due to space constraints, the presented example is restricted to one-edge restrictions.

This can result in a model where a single RTSC state has a mix of possible, trigger, or asserted transitions with timing constraints. To respect priorities and assertions, the synthesis requires that asserted transitions exclude trigger and possible transitions, and that trigger transitions exclude lower-prioritized trigger transitions with the same events. To achieve this, the guard of a lower priority transition is refined through conjunction with the negated guard of the higher priority transition as follows: The guards $g$ of each of these constraints can be described as the conjunction $g_b \land g^u \land (\neg t_0)$ with $g_b$ the boolean guard, $g^u$ the time guard consisting only of the lower bounds for clocks, and $g^u$ the time guard with the upper bounds. To respect priorities, assertions, etc., the synthesis algorithm manages all transitions of a specific state in a list, sorted at first by the priority and secondly by the type (asserted > trigger > possible).

We then proceed as follows: (1) The first transition (with maximal priority) $t$ with guard $g_b \land g^u \land (\neg t_0)$ is taken out of the list. The further processing distinguishes the following cases: (1a) If the transition is an asserted one which additionally requires an event, the guards $g$ of all transitions of that state with lower priority are made disjoint by adding $\neg (g_b \land g^u)$ via a logical and-composition to ensure that a lower priority transition can only be executed when this transition is not possible any more. (1b) If the transition is an asserted one which requires no event, the guards $g$ of all transitions of that state with lower priority are made disjoint by adding $\neg g_b$ via a logical and-composition to ensure that a lower priority transition can only be executed when this transition is not possible at all. (1c) If we have a trigger transition which additionally requires an event $e$, the guards $g$ of all transitions of that state with lower prior-

Figure 3. Synthesized parameterized real-time statechart of the watchdog
A TCG is not satisfiable if the required sequence of events in each statechart is not possible. If a possible transition is missing, the scenario can never occur because the condition for the assertion is never fulfilled. If in contrast an asserted transition does not exist any more, the scenario is in conflict with another higher priority scenario that prevents the asserted behavior from happening.

Using the capabilities of the consistency check outlined in Section 3, we derive systematically a consistent set of values for the timing parameters. The resulting system of non-parameterized real-time statecharts can then usually be simplified, as multiple clock constraints for a single clock can be combined into a single one.

In Figure 4, the non-parameterized RTSC which is derived from the parameterized RTSC (cf. Figure 3) is presented, using the following parameter settings: \( d_{\text{waChR}} = 150.0, \ w\text{WaChSD} = \infty, \ w\text{WaChI} = \infty, \ w\text{WaChS} = \infty, \ WDU = 100.0, \ wtu = 600.0 \ WTL = 500.0, \ WCU = 40.0, \ OFF_T = 75.0, \) and \( \text{OKer} = 560 \) is used. Due to the fixed parameter values, the complex guard conditions of the parameterized RTSC have been considerably simplified.

6. Conclusion and Future Work

We present the current status of our tool support for real-time system development by means of parameterized timed sequence diagrams with conditional behavior. Static analysis ensures consistency and causality and detects scenarios that are disabled by others even for the parameterized models. The synthesis of a parameterized statechart model further completes our approach. The provided support for timing constraints with parameters further enables the systematic study of the trade-off between different design solutions w.r.t. time constraints.

We plan to improve the presented approach and prototype tool by supporting simplification rules for the generated RTSC. In addition, a tighter integration with our UML design approach [7] and model checking is planned.

Acknowledgements

The authors thank the student Sergej Tissen for his implementation of a first prototype version of the presented tool and his valuable comments and questions.

References


Figure 4. Synthesized real-time statechart of the watchdog with fix parameter values


