Nobody's perfect: Interactive Synthesis from Parametrized Real-Time Scenarios*

Holger Giese, Stefan Henkler, Martin Hirsch†, and Florian Klein‡
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[hg|shenkler|mahirsch|fklein]@uni-paderborn.de

ABSTRACT

As technical systems keep growing more complex and sophisticated, designing software for the safety-critical coordination between their components becomes increasingly difficult. Verifying and correcting these components already represents a significant part of the development process both with respect to time and cost. Scenario-based synthesis has been put forward as an approach to accelerate the transition from requirements to a correct, verified model. In [8], we have presented a synthesis technique for deriving pattern behavior from a set of timed scenarios with parametrized time constraints. The derived patterns can then be verified using our technique for the compositional formal verification of Mechatronic UML models as introduced in [10]. In this paper, we argue that the practical relevance of a synthesis technique predominantly depends rather on its ability to identify and point to specification errors than the complexity of the scenarios it could, in theory, process, provided with a correct specification. By means of a case study, we introduce the different types of specification errors that may arise during synthesis. Using our tools for modeling, synthesis, and verification, we then show how we can identify and resolve these errors in the successive phases of an interactive development process.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification

Keywords

Scenario-based synthesis, Diagnosis, Model Checking, Patterns

1. INTRODUCTION

Technical systems are becoming more capable, but also complex and sophisticated. Often, the improvements are due to the use of software or, increasingly, the interaction within a network of software components. As these systems are often safety-critical, software that controls essential functions needs to conform to a high quality standard that can only be achieved by means of formal verification. Designing and verifying the coordination mechanisms is both time-consuming and costly and represents a significant part of the development process.

Scenario-based synthesis has been proposed as an efficient way to derive correct component behavior. Starting from scenarios as an intuitive way to formalize behavioral requirements is supposed to facilitate arriving at statecharts that represent the actually desired behavior. This is especially true for real-time behavior, where specifying time constraints for a specific scenario is usually much more straight-forward than for the resulting automaton.

Consequently, there is a variety of specification techniques and corresponding synthesis algorithms. In [8], we have presented a synthesis technique for deriving real-time coordination patterns from a subset of UML sequence diagrams with time annotations. Its characteristic feature is that the upper bounds of time constraints may be parametrized. For the synthesized pattern, these parameters can then be set to arbitrary values, within the confines computed by the synthesis algorithm. Once a pattern has been supplied with concrete values, it can be applied to a component structure. Using the compositional formal verification technique for Mechatronic UML models that was introduced in [10], we can then verify the overall system by verifying the individual patterns and components.

However, intuitive as a scenario-based approach may be, synthesis remains a tricky business, especially when time constraints come into play. Even if current synthesis algorithms are capable of handling the required number and complexity of scenarios to arrive at a behavioral specification for a complex technical system, the synthesized result is only as good as its input. It is by no means certain that the designer is capable of supplying the necessary scenarios that correctly and consistently represent exactly the desired behavior. To do so on the first attempt is nearly impossible, even more so for practical applications where the designer is not an ex-

†supported by the University of Paderborn.
‡supported by the International Graduate School of Dynamic Intelligent Systems.
*This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

SCESM'06, May 27, 2006, Shanghai, China.

pert on synthesis. Besides, there is not even a guarantee that such a specification exists at all. While it is always possible to synthesize a model for a given set of unconstrained scenarios, mandatory behavior as present, for example, in live sequence charts (LSC) or time constraints may introduce inherent contradictions that make synthesis impossible.

In the domain of formal verification, a similar problem exists. Nonetheless, model checking has been remarkably successful in practice. This is largely due to one of the most compelling features of model checking techniques, the ability to generate concrete counterexamples that explicitly and intuitively demonstrate exactly what can go wrong. It is hard to imagine that a technique that took a complex temporal logic specification and then simply answered 'yes' or 'no' would have gained widespread popularity. In this vein, it is therefore equally crucial that a synthesis algorithm provides sufficient feedback that enables the quick localization and removal of specification errors in an iterative process.

In this paper, we present different types of specification errors that may arise when synthesizing component behavior from parametrized real-time scenarios. Using a case study from the RailCab[1] domain, we run through the phases of scenario modeling, synthesis and formal verification and point out which error types may become apparent at each stage. For each error type, we provide an example and show how it was identified and reported by the tools we have developed. Ultimately, we arrive at the desired verified component behavior.

In Section 2, we provide a more detailed description of our modeling approach (2.1), synthesis technique (2.2) and verification technique (2.3) and present an overview of related work. In Section 3, we introduce the case study and begin with modeling the relevant scenarios (3.1). We then generate parametrized statecharts and handle the problems arising from overlaying the different scenarios (3.2). After fixing the free parameters, the resulting statecharts are then verified (3.3). Section 4 provides a conclusion and an outlook on future work.

2. APPROACH

The central idea of our approach is to compose complex software systems from coordination patterns describing component interactions. Patterns specify ports, with interconnecting channels, a statechart specifying the required role protocol for each port, and pattern constraints expressing a set of desired properties. They thus comprise structure, behavior, and requirements.

A component is then created by composing and refining a set of roles. The patterns then imply valid ways of connecting these components into the overall system.

Our synthesis technique allows us to replace the manual specification of the pattern role protocols by synthesis, making it both faster and less error-prone. Note that, as the individual role protocols are usually of fairly limited size, the central problem in this context is not identifying states and transitions, which are often fairly obvious, but finding local time constraints and network delays that, in conjunction, yield the desired complex system behavior. In practice, specify timing information such as worst-case execution times (WCET), deadlines, or timeouts in advance is difficult. By starting from parametrized scenarios, we can later set the parametrized constraints in a meaningful context and analyze the trade-offs between different alternative parameterizations by varying the constraints, in a way that is compatible with the overall specification.

Once the parameters have been fixed, the concrete pattern can be formally verified to ensure that the synthesis result respects all the pattern constraints that are imposed by the pattern. Verification by means of model checking is feasible for individual patterns, as only a limited number of components is involved. Each component is additionally restricted to a fixed number of role behaviors, with limited side effects. They are joined in parallel composition, but as they are usually not independent of each other, appropriate composite behavior needs to be derived by means of refinement. Once the local behavior is consistent, however, and all ports are connected in a syntactically correct manner, the verified properties are carried over to the overall system.

In the following subsections, we briefly introduce the concepts and notations that are relevant to our application example. Please consult [8] for an in depth discussion.

2.1 Modeling

When designing patterns, we abstract from the components and only represent them by the roles they perform in the pattern. The roles' interfaces are provided by named ports, which can be connected by connectors modeling asynchronous or synchronous communication channels (see Figure 1). The behavior of ports and asynchronous channels is specified by statecharts. Pattern constraints may be specified using temporal logic.

Because of their modular nature and the abstraction by way of roles, patterns promote reuse. Once verified, patterns can be applied to similar cases with a compatible component structure. Patterns that are still parametrized are even more flexible, but anew require a parameterization and verification step, as verification is not possible for the generic case.[2]

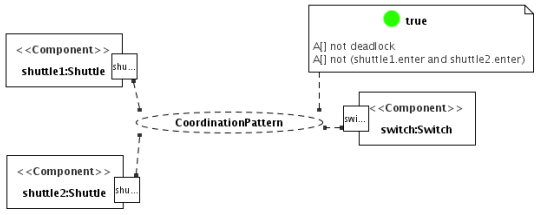

Figure 1: A coordination pattern

Once the structure of the pattern has been defined, we can move on to role behavior. Instead of specifying the statecharts directly, we opt for identifying the desired sequence of events for several representative cases, which is more natural to most designers than reasoning in states.

We use a restricted subset of UML 2.0 sequence diagrams [17, p. 435] to specify parametrized timed scenarios. UML 2.0 sequence diagrams allow specifying durations for message transfers and lower and upper bounds for the time passed between two points on a lifeline. We can also observe the current time and reference this measurement in constraints. Upper bounds may be arbitrary sums of observations, constants and parameters, whereas lower bounds may only consist of observations and constants. This restriction is currently required by our synthesis technique.

Distinguishing optional and required behavior is another important aspect of scenarios. Triggers have first been proposed as a technique for expressing conditional behavior for live sequence charts

[1] http://www-nbp.upb.de/en/index.html

[2] The problem of emptiness for parametrized timed automata with more than 2 parameters has been proven to be undecidable in [2] by showing that the halting problem for 2-counter machines can be encoded. The same problem can be reduced to the synthesis problem which excludes time-stopping deadlocks for via scenarios with related state labels.

(LSC) [11] and also appear in triggered message sequence charts (TMSCs) [19]. We use the assert block introduced by UML 2.0 sequence diagrams [17, p. 444] to describe conditional behavior of parametrized timed scenarios. Such blocks indicate a mandatory sequence of behavior that needs to be executed once the preceding steps have been observed and the block has been entered. For dealing with cases where several assertions clash, we also assign priorities to scenarios.

In order to facilitate the transition to a state-based model, we explicitly add state labels to the lifelines which represent states or sets of possible states. The labels allow using self-documenting state names in the generated statecharts and make overlaying different scenarios easier and much more efficient. * refers to all states that have been defined outside of the present scenario, + and - allow manipulating state sets.

Figure 2 is an example of such a scenario with state annotations. The parameter *switch_wcet* specifies the acceptable worst case execution time for the switch role.

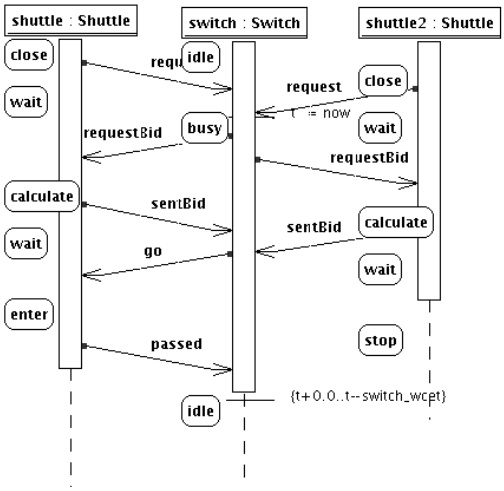

Figure 2: A scenario (UML 2.0 sequence diagram)

In the synthesized pattern, the derived operational description of the roles' real-time behavior is expressed by statecharts. UML 2.0 statecharts only provide rudimentary support for timing constraints (when/after) relative to the time of entry into a state, but cannot express complex timing constraints involving sequences of states. Moreover, their semantics cannot be implemented on a real machine, as they assume transitions to be instantaneous. Finally, event triggers and time guards for transitions are mutually exclusive.

We therefore use real-time statecharts (RTSC) [6] that introduce several extensions. Clocks may be reset when entering or leaving states and when firing transitions. States have time invariants that restrict the permitted clock values. Transitions have time guards that restrict their permitted activation time. Transitions may be urgent or non-urgent. Enabled urgent transitions fire immediately, enabled non-urgent transitions can fire arbitrarily, unless forced by a state invariant or other constraint. Transitions consume time. They may specify a worst case execution time and a deadline. Operations (side-effects) are likewise annotated with WCET and deadlines.

In a parametrized RTSC (PRTSC), clock constraints are not merely constants, but expressions containing parameters and constants. Figure 3 depicts an example of such a statechart.

The Fujaba CASE Tool[3] supports all of the modeling techniques

[3]http://www.fujaba.de

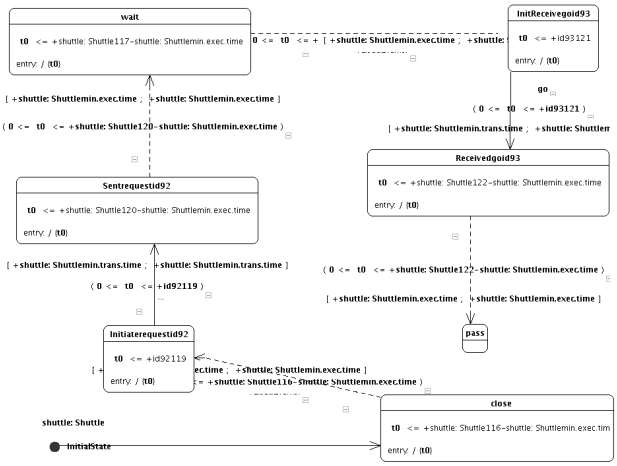

Figure 3: A generated parametrized real-time statechart

introduced above. It provides UML 2.0 sequence diagrams with conditional behavior and parametrized timing constraints. RTSC and real-time pattern are supported by the real-time version of Fujaba. RTSC have a well-defined real-time semantics based on timed automata [12]. Code generation [6] as well as real-time model checking [13] are currently supported for real-time statecharts.

2.2 Synthesis

As mentioned above, a variety of approaches exist which permit using a set of scenarios for the synthesis of operational state-based component behavior (cf. [14, 16, 21, 24]). In the real-time domain, however, the focus is on the timing constraints and their correct transcription.

Several proposals to check timed system models against scenario descriptions with time have been made (cf. [15]). For actively synthesizing such models from scenarios with timing constraints, there are only very restricted approaches. The approach proposed in [20] synthesizes only global solutions in form of a single automaton for non-parametrized scenarios, which assumes angelic non-determinism[4] and does not support progress conditions. The approach of [18] results in a global non-parametrized timed automata which supports progress, but requires scenario descriptions in form of trees that already introduce some of the required operational behavior. The play-out engine [11] enables the play-out of live sequence charts (LSC) with timers, but also only constructs global behavior for non-parametrized LSCs.

Addressing the general problem of scenario-based synthesis for parametrized real-time systems would lead to scalability problems. The synthesis problem for real-time patterns remains tractable because the patterns limit the number of interacting roles so that only a moderate number of scenarios has to be considered together.

While generating a statechart from a single scenario is trivial and overlaying several scenarios is at least facilitated by the state labels, the additional timed constraints complicate matters. The time constraints need to be chosen in such a way that the resulting specification is both consistent within a single scenario, e.g., that a deadline may not be earlier than the sum of the lower bounds of all required operations, and across scenarios. Assert blocks can also lead to conflicts when two scenarios require mutually exclusive behavior.

As the synthesis problem in its most general form is intractable

[4]cf. [23]

for the parametrized case as well as mandatory behavior and not centralized behavior, we proposed (like other approaches which are also intractable due to mandatory behavior and not centralized behavior [5]) to separate the synthesis task into a feasible first step which addresses each process only taking the local context into account and a verification step which checks whether the synthesis result is indeed correct. In the case of parametrized systems this second step is in fact only possible after values for the parameters have been chosen.

In order to solve the synthesis problem, the parametrized time constraints are transformed into a system of linear inequalities. These are then passed to a constraint solver (either a free proprietary implementation of the simplex algorithm or the – much more efficient – commercial CPLEX library), which determines feasible values, respectively ranges, for each parameter. If a configuration yielding a consistent specification exists, the scenarios can be transformed and integrated in order to obtain role behaviors. The resulting PRTSC can then be turned into regular RTSC by choosing parameter values from the computed ranges.

The synthesis process is integrated into the Fujaba Tool Suite as a plugin. Beside the sequence diagrams that serve as input and the resulting PRTSC and RTSC, the plugin also provides access to model parameters (user-defined or computed, see Figure 4) and graphical representations of the time constraint graphs used internally as an intermediate format.

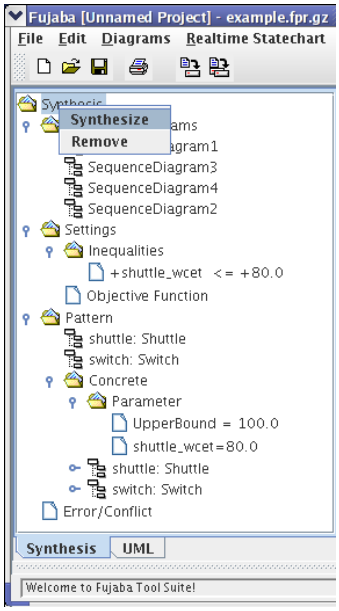

Figure 4: The Fujaba synthesis navigator

2.3 Verification

Support for the verification of real-time patterns is provided directly from within the Fujaba Tool Suite by means of another plugin. Pattern constraints, such as deadlock freedom or safety properties, can be expressed using temporal logic specifications expressed in the restricted TCTL [1] (computational tree logic) variant ATCTL, which only contains *always path* operators. Once a constraint has been added and verified, it is automatically updated whenever the component is modified and displayed as an annotation (see Figure 1)[7].

The actual verification is performed by the model checker UPPAAL [4]. As the formal semantics of RTSC are based on Timed Automata, it is possible to directly transform the role and channel protocols of a pattern into the required input format of the model checker, which is based on the same formalism. In case the constraint is not fulfilled, UPPAAL generates a counterexample in the form of an interactive, graphical execution trace that allows inspecting the exact sequence of transitions that has led to the violation.

3. APPLICATION

We now apply our approach to the design of a case study in order to illustrate the procedure and the possible error types that may arise at different stages.

As our case study, we use a concrete example from our work in the collaborative research center 614 of the German National Science Foundation (DFG), titled *Self-optimizing Concepts and Structures in mechanical Engineering*. As an example application, we are exploring ways of enhancing the intelligent shuttles of the RailCab research project by adding self-optimizing behavior.

In accordance with the agent paradigm, the shuttles are acting autonomously and making independent and decentralized decisions, aimed at maximizing their own profit. They are equipped with a wireless communicating network (WLAN). The WLAN can be used for shuttle-to-shuttle communication or a shuttle-to-track communication. While the track system is generally passive, tracks may provide simple services to agents.

In this paper, we consider two shuttles arriving on different tracks which are joined at a (passive) switch. As both shuttles are approaching the same location, a crash at the switch is a hazard, depending on their positions and velocities. In order to exclude accidents, there needs to be some form of coordination w.r.t. the passage of the switch. As depending on the shuttles' objective function and current situation, the shuttles might either be interested in going first (e.g. in order to stay ahead of a slower shuttle) or actually yielding (e.g. for forming a convoy), their behavior is difficult to predict.

A first idea for solving the problem is based on direct shuttle-to-shuttle coordination. One shuttle is selected by means of some criteria and then tells the other shuttle what to do. A solution that is more suited for a multi-agent system would be to let the shuttles negotiate about the right of way at the switch. For example, the shuttles might bid for the right of way. However, as both shuttles have an interest in the matter, the situation would be asymmetric, as the shuttle hosting the auction might cheat.

A better solution would therefore be to use the switch as an arbitrator. Each agent sends a bid to the switch in a one-shot Vickrey auction [22], i.e. an auction where the highest bidder wins, but needs to pay the bid of the second highest bidder. As Vickrey auctions promote truthful behavior, i.e. announcing one's true valuation is the dominant strategy, this would allow the agents to quickly arrive at a mutually acceptable result.

3.1 Scenarios

We now attempt to describe this solution using only a small number of representative runs. First of all, however, we need to model the participants of these runs, i.e. the switch and up to two shuttles, and their structure. Each of them is modeled as a component; the wireless link between them is modeled as an asynchronous channel connecting these components. We furthermore add the requirements that the system should be deadlock free and that only one shuttle at a time will enter the switch, i.e. that collisions are avoided. Together, these components, connectors and requirements form the *SwitchAuctionPattern*, as already presented in Figure 1.[5]

[5]Note that the instantiation of the patterns can be specified and

We can then proceed with describing the different role behaviors of the pattern by means of scenarios. We model for different scenarios, specified as UML sequence diagrams (cf. Figure 5, 6, 7 and 2). The cases we identified as relevant are (a) a single shuttle successfully passing the switch, (b) one shuttle (or more) requesting passage, but not receiving a timely reply from the switch, (c) two shuttles meeting at the switch, successfully carrying out an auction, and passing the switch in the determined order (d) two shuttles bidding, the auction winner receiving the permission to enter the switch, whereas the losing shuttle times out while waiting. Scenarios (a) and (c) thus specify desired behavior, whereas scenarios (b) and (d) describe likely deviations from the ideal case.

Figure 5 shows scenario (a) which models the communication between one shuttle and the switch. First shuttle sends a request message to switch, which replies to shuttle with a go message. In this scenario, no negotiation is shown as there is just one shuttle. After receiving the go message the shuttle changes to the state enter. When leaving the switch, the shuttle finally sends a pass message back to the switch and enters the state pass. The time span between state close and pass is specified by a timing constraint. In state close the time observation t is reset and until shuttle is in state pass, at most t+pass_wcet time units can elapse. To describe that the switch has to turn to busy right after receiving the request message, a timing constraint t+busy_wcet is associated with state busy.

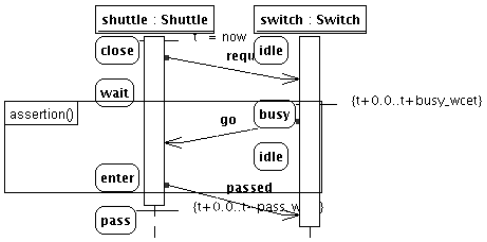

Figure 5: Scenario (a) - Passing a working switch

Like scenario (a), scenario (b) in Figure 6 contains no negotiation. Unlike in the first scenario, the switch does not reply to the shuttle. The shuttle has to assume that the switch has failed and stops, as passing the switch without permission would risk a collision. Switching to the stop state is brought about by means of a timing constraint of the wait state that has been defined a priori.

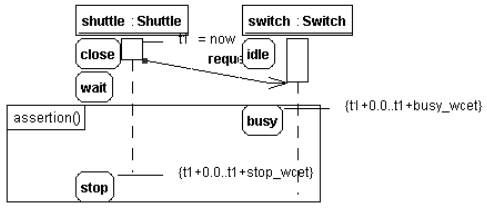

Figure 6: Scenario (b) - Switch failure

The preceding scenarios have only considered a shuttle-to-switch communication. In Figure 7 and 2, we consider an auction-based negotiation. Figure 7 shows the conflict-free scenario (c), basically the ideal execution of the auction. After receiving two concurrent requests, the switch asks each of them to place a bid. Both shuttles then send bids to the switch. The switch now acts as an arbitrator and decides on the winning bid. The switch then sends the corresponding shuttle a go message. After the shuttle has passed the switch and confirms this with a passed message, the switch sends a go message to shuttle2, which is then free to pass the switch as well.

Scenario (d) in Figure 2 (see Section 2.1) is identical to scenario (c) in Figure 7 up to the point where the switch is supposed to send the second go message to the losing bidder. As the switch does not send a go message to shuttle2, the second shuttle times out and, after a predefined time, switches to the stop state.

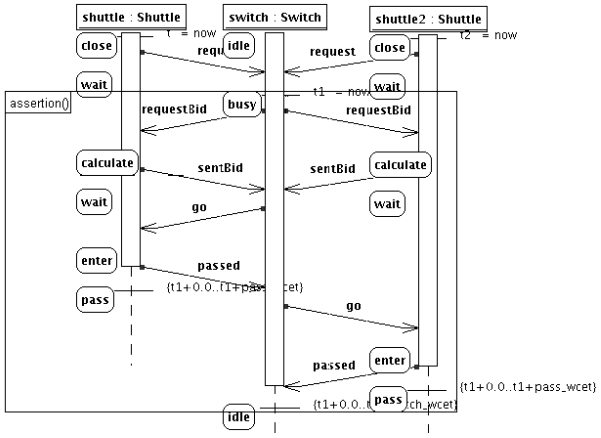

Figure 7: Scenario (c) - A successful auction

3.2 Synthesis

In general, the scenarios can contain several errors which make a synthesis impossible. This may be at first syntactical errors such as multiple definitions of the same time observation or time constraints where the lower bound is larger than the upper bound. Another class of errors is detected when we check that for the given set of scenarios solutions are possible which guarantee consistency and locality (cf. [8]). When we go one step further and merge the different scenarios into a statechart for each role, in addition conflicts between alternative transitions of the same state which result from different scenarios have to be excluded.

Checking Consistency and Locality The simplest type of error concerns individual states or side effects. In scenario (a) (Figure 5), time constraints are specified for the time spans between state close and state busy and between state close and state pass. The upper bound for the former time span is determined by the parameter busy_wcet $<= 25$, the latter is restricted by pass_wcet $<= 30$. This implies that the time span between busy and pass may be as short as 5 time units. Together, these constraints lead to a contradiction (caused by the communication delay of the message go), which is immediately flagged by the tool when performing the analysis. The designer may relax one of the constraints in order to resolve the conflict. In our case, we use the tool to compute the admissible MIN and MAX values for the variables. As a result, we find that busy_wcet has to be at least 25 and pass_wcet has to be at least 33.

Excluding Conflicts Finally, conflicts arise when two scenarios with the same priority contain assert blocks that assert different transitions or activities for the same state. In the case study, the assert blocks in scenarios (a) and (c) require different behavior in state busy. In this case, the tool opens a dialog, and highlights the exact state where the conflict arises and points to the corresponding assert blocks (cf. Figure 8).

The simplest way to resolve this type of conflict is to adjust the priorities of the relevant sequence diagrams, which will then allow

verified using a related extension of Fujaba [3].

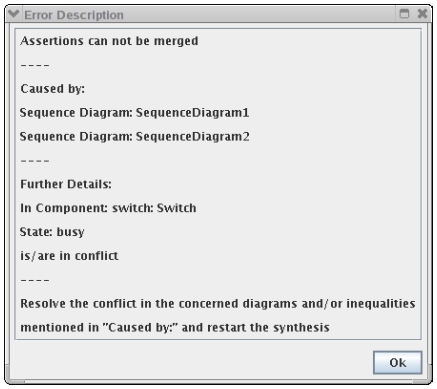

Figure 8: Assertion conflict

the algorithm to decide which one of several conflicting behavioral options should take precedence. In our case, this does not lead to the desired behavior, because the state busy serves for two different tasks. Firstly, in state busy all incoming requests are collected. Secondly, the decision for the auction is performed. To resolve the assertion conflict, the tasks are decoupled by splitting busy in the states assert and decide (cf. Figure 9).

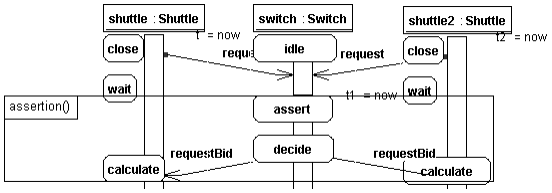

Figure 9: Solution for resolving assertion conflict

3.3 Verification

Using the range constraints that have been computed by the constraint solver as guidelines, we can now set the remaining parameters of the pattern to concrete values that preserve the behavior as specified by the scenarios.

Once valid parameters for the real-time statecharts have been set, we can use the model checking features of the Fujaba Real-Time Tool Suite in order to ensure that the synthesis result is free from deadlocks or time stopping deadlocks and fulfills all pattern constraints. As verifying these properties requires the exploration of the whole state space, we need other techniques and algorithms that go beyond the available analysis algorithms to cover this problem. Model Checking is one efficient technique for verifying such constraints. The model checking plugin of the Fujaba Real-Time Tool Suite makes it possible to verify the behavior of a component structure or pattern described by real-time statecharts, provided that all parameters are bound, by mapping the Real-Time Statecharts to timed automata, which can then be verified by the model checker UPPAAL. The benefit of integrating model checking directly into our approach is that it allows us to easily obtain counterexamples when a property is not fulfilled.

In our case study, we are mainly interested in the question whether the pattern constraint *A[] not (shuttle1.enter and shuttle2.enter)* (cf. 1) is fulfilled or not. The pattern constraint ensures that only one shuttle enters the switch at the same time, thus avoiding collisions. To begin with, the pattern, consisting of the three pattern-roles *shuttle1*, *shuttle2* and *switch* are mapped to timed automata. The pattern constraint is already specified in ATCTL so that no transformation is necessary. The actual verification is performed by the verifier engine of the model checker UPPAAL. In case of the pattern constraint, the model checker returns *false* and an error trace (cf. Fig. 11). Using the graphical front-end for UPPAAL, we can then analyze the error trace; the process of mapping the error trace back to UML 2.0 sequence diagrams is not yet automated. During our manual inspection, we find that shuttle1 and shuttle2 are in state enter at the same time because they have simultaneously received a go message. This error was caused by the parameterization we have chosen for scenario (c) (see Figure 7). Contrary to what was intended, the chosen parameters have failed to ensure the proper order of the go messages and thus made the run depicted by the scenario in Figure 10 valid behavior. After adjusting the constraint parameters of scenario (c) so as to ensure that the second go message occurs after the first passed message, we repeat the verification, this time successfully meeting all constraints.

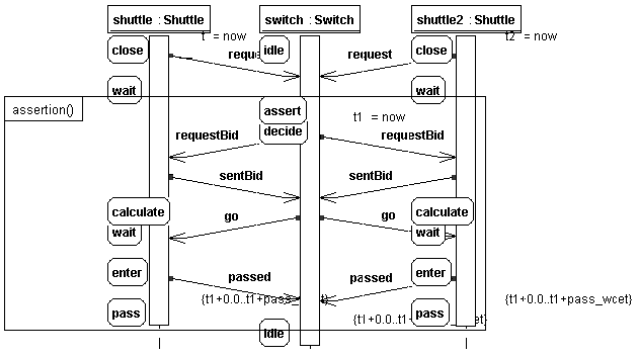

Figure 10: The resulting concrete instance of scenario (c)

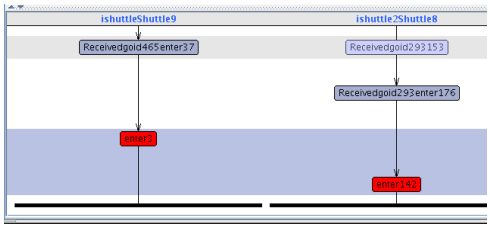

Figure 11: The error trace as presented by UPPAAL

4. CONCLUSION AND FUTURE WORK

We have shown how during the scenario-based synthesis from real-time scenarios, different types of specification errors may arise. Firstly, time constraints in the different scenarios which refer to the same side effects and states of the operational behavior can be in conflict. When synthesizing the operational behavior, the different transitions of the same state in different scenarios might secondly be in contradiction. We provide tool support for diagnosing such conflicts, as well as the ability to assign priorities to different scenarios in order to interactively resolve such conflicts. Finally, the generated operational behavior is not necessarily free of deadlocks and may also not realize the abstract properties identified at the outset. We cope with this by supporting the model checking of model instances with concrete values for each parameter.

The presented results highlight the practical relevance of tool support for diagnosis when a problem is detected. To actually pro-

vide the frequently touted productivity increases in practice, the ability to identify and pinpoint the errors in the specification thus seems to be as crucial as the synthesis algorithm and its complexity themselves. We therefore advocate that the evaluation of synthesis approaches in our field should not merely focus on what can be synthesized but also on whether the provided diagnostic information is sufficient for identifying error causes and arriving at a correct specification for non-trivial examples and thus actually improves productivity.

In the near future, we plan to integrate refinement checks for real time statecharts [9], which will allow us to verify whether manual modifications to synthesized statecharts represent valid refinements of the original behavior.

Acknowledgements

The authors thank the student Sergej Tissen for his implementation of the scenario analysis and synthesis plugin.

5. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, PA, 1990.

[2] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 592–601. ACM Press, 1993.

[3] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. of the* 28^{th} *International Conference on Software Engineering (ICSE), Shanghai, China*. ACM Press, 2006. (accepted).

[4] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[5] Y. Bontemps and P. Heymans. As fast as sound (lightweight formal scenario synthesis and verification). In H. Giese and I. Krüger, editors, *Proc. of the 3rd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools" (SCESM'04)*, pages 27–34, Edinburgh, May 2004. IEE.

[6] S. Burmester and H. Giese. The Fujaba Real-Time Statechart PlugIn. In *Proc. of the Fujaba Days 2003, Kassel, Germany*, October 2003.

[7] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, pages 670–671. ACM Press, May 2005.

[8] S. Burmester, H. Giese, and F. Klein. Synthesis of Parameterized UML Real-Time Patterns from Multiple Parameterized Real-Timed Scenarios. In F. Bordeleau, S. Leue, and T. Systä, editors, *Scenarios: Models, Algorithms and Tools*, volume 3371 of *Lecture Notes in Computer Science*, pages 193–211. Springer Verlag, April 2005.

[9] H. Giese and M. Hirsch. Checking and Automatic Abstraction for Timed and Hybrid Refinement in Mechtronic UML. Technical Report tr-ri-03-266, University of Paderborn, Paderborn, Germany, December 2005. (to appear).

[10] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[11] D. Harel and R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proc. 10th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, Fort Worth, Texas, USA, 2002. (invited paper).

[12] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *Proc. of IEEE Symposium on Logic in Computer Science*. IEEE Computer Press, 1992.

[13] M. Hirsch and H. Giese. Towards the Incremental Model Checking of Complex RealTime UML Models. In *Proc. of the Fujaba Days 2003, Kassel, Germany*, October 2003.

[14] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[15] X. Li and J. Lilius. Timing Analysis of UML Sequence Diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Second International Conference on The Unified Modeling Language Fort Collins, Colorado, USA*, volume 1723 of *Lecture Notes in Computer Science*, October 1999.

[16] E. Mäkinen and T. Systä. MAS - an interactive synthesizer to support behavioral modeling in UML. In *Proceedings of the* 23^{rd} *International Conference on Software Engineering (ICSE 2001), Toronto, Canada*, pages 15–24, May 2001.

[17] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. Document ptc/03-08-02.

[18] A. Salah, R. Dssouli, and G. Lapalme. Implicit integration of scenarios into a reduced timed automaton. *Information and Software Technology*, 45:715–725, August 2003.

[19] B. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. In W. G. Griswold, editor, *Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Softare Engineering (FSE-10)*, Charleston, South Carolina, USA, November 2002. ACM Press.

[20] S. Somé, R. Dssouli, and J. Vaucher. From Scenarios to Timed Automata: Building Specifications from Users Requirements. In *Proceedings of the 1995 Asia Pacific Software Engineering Conference (APSEC '95)*, 1995.

[21] S. Uchitel and J. Kramer. A Workbench for Synthesising Behviour models from Scenarios. In *Proceedings of the* 23^{rd} *International Conference on Software Engineering (ICSE 2001), Toronto, Canada*, pages 188–197, May 2001.

[22] W. Vickrey. Counterspeculation and Competitive Sealed tenders. *Journal of Finance*, 16:8–37, March 1961.

[23] M. Walicki and S. Meldal. Algebraic Approaches to Nondeterminism—an Overview. *ACM Computing Surveys*, 29(1):30–81, March 1997.

[24] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on on Software engineering June 4 - 11, 2000, Limerick Ireland*, 2000.