

Synthesis of Timed Behavior From Scenarios in the Fujaba Real-Time Tool Suite *

Stefan Henkler, Joel Greenyer, Martin Hirsch, Wilhelm Schäfer,
Kahtan Alhawash, Tobias Eckardt, Christian Heinzemann, Renate Löffler
Software Engineering Group
University of Paderborn, Germany
{shenkler,jgreen,mahirsch,wilhelm,alhawash,tobie,chris227,renate}@uni-paderborn.de

Andreas Seibel and Holger Giese
System Analysis and Modeling Group
Hasso Plattner Institute
University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
D-14482 Potsdam, Germany
{andreas.seibel,holger.giese}@hpi.uni-potsdam.de

Abstract

Based on a well-defined component architecture the tool supports the synthesis of so-called real-time statecharts from timed sequence diagrams. The two step synthesis process addresses the existing scalability problems by a proper decomposition and allows the user to define particular restrictions on the resulting statecharts.

1. Introduction

The current and even more so next generation of so-called embedded or mechatronic systems will behave more intelligently than today's systems by building communities of autonomous components (or agents) that exploit local and global networking to enhance their functionality [10].

Typical examples of this type of systems are advanced transportation systems like driver assistance systems in cars, which can inform the driver about accidents and also take appropriate actions in case of an emergency, or as another example, cooperative illumination of roads. In either case, cars communicate with each other extensively. Another example of this type of system is the Paderborn RailCab system, which is used to illustrate the approach described here.

*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.

The complexity of these systems as well as the inability to test them sufficiently enough to guarantee crucial safety properties demands for a model driven approach. Our approach, called MECHATRONIC UML, which is implemented by the FUJABA Real-time Tool Suite, is based on specifying components by a refined UML 2.0 component model. Refinement concerns in particular the detailed definition of ports and connectors which specify a peer to peer communication between components by using real-time statecharts¹. Each single port-to-port connection is defined by a single statechart that specifies the roles of the two ports and the corresponding connector properties. Such a port-to-port communication is called a coordination pattern in our approach [4]. Further refinement of the component model concerns a proper integration between the discrete parts specifying the component interaction and continuous control specifying individual controllers by differential equations.

While our tool demonstration at ICSE 07 [2] presented the latter part and especially illustrated real-time simulation of components in a 3D environment, this paper and the corresponding tool is about specifying the coordination patterns and the individual component communication behavior. The precise definition of the coordination patterns is synthesized from a scenario-based specification. This specification especially supports the definition of time constraints as it is required for the communication of hard real-time systems. The synthesis algorithm generates a real-time statechart specifying a single coordination pattern from a

¹Real-time statecharts are an extension of timed automata

number of given scenarios. In a second synthesis step, a complete component behavior concerning its interaction with other components is automatically derived from the statecharts generated by the first step. These two synthesis steps (see Figure 1) and especially their corresponding tool support are the subject of this paper. Please note that the definition of coordination patterns is also the key to a compositional formal verification approach which is however beyond the scope of this paper and has been described elsewhere [4, 3].

Key original contributions of the two synthesis steps are (1) the use of parameterized timing constraints in the definition of the sequence diagrams which are input to the first synthesis step and (2) the not necessarily only parallel composition of the different statecharts in the second step, i.e. the algorithm accepts additional so-called state restrictions as input which may exclude certain (timed) state sequences from the parallel product of the statecharts (roles). The algorithm however checks whether those restrictions do not violate the specifications of the sequence diagrams from which the statecharts have been derived, i.e. restrictions do not change the observable behavior of the roles.

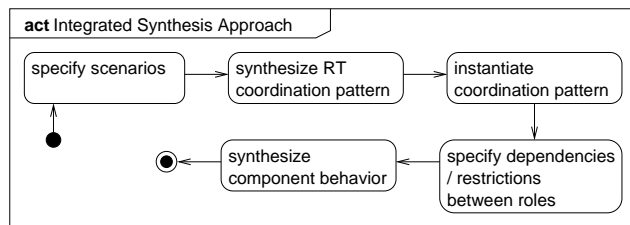


Figure 1. Overview synthesis process

Next, we explain the parametrized timed scenario synthesis approach. Afterward, we explain the synthesis approach of the component behavior. Finally, we compare our approach with related work and we conclude with final remarks.

2. Coordination Behavior Synthesis

We use a restricted subset of UML 2.0 sequence diagrams [8, p. 435] to specify parametrized timed scenarios. UML 2.0 sequence diagrams allow us to specify durations for message transfers and lower and upper bounds for the time passed between two points on a lifeline. Upper bounds may be arbitrary sums of constants and parameters whereas lower bounds may only consist of constants (in contrast to UML 2.0 sequence diagrams).

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 (LSC) [6] and also appear in triggered

message sequence charts (TMSCs) [11]. We use the `assert` block introduced by UML 2.0 sequence diagrams [8, p. 444] to describe the 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 are in conflict, we 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 [8, p. 442]. The labels allow us to use self-documenting state names in the generated statecharts and to make overlaying the different scenarios easier and much more efficient.

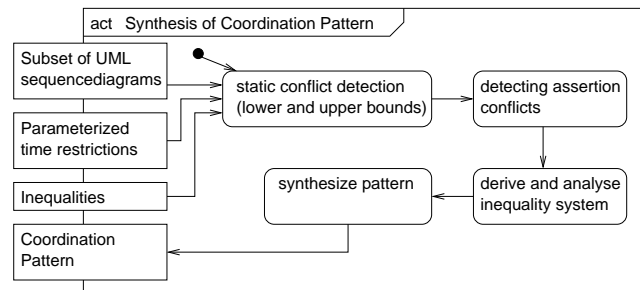


Figure 2. Overview synthesis of coordination pattern

The synthesis algorithm (see Figure 2) first checks whether the specified time constraints are 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. Secondly, time constraints are checked across different scenarios and the algorithm checks `assert` blocks for correctness as they can lead to conflicts when two scenarios require mutually exclusive behavior.

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 valid values, respectively ranges, for each parameter. If a configuration yielding a consistent specification exists, the scenarios can be transformed and combined in order to obtain role behavior in form of a parametrized real-time statechart. These can then be turned into regular real-time statechart by selecting parameter values from the computed ranges.

The “two step” approach sketched above addresses the usually intractable synthesis problem, at least in its most general form. Like other approaches [1] which also consider the parametrized case as well as mandatory behavior and not centralized behavior, the first step takes only lo-

cal context into account. A second verification step checks whether the synthesis result is correct after valid parameter values have been selected.

In practice, specifying timing information such as worst-case execution times (WCET), deadlines, or timeouts early during the requirements definition phase is difficult. By starting from parametrized scenarios, we can set the parametrized constraints in a meaningful context (the requirement of a particular application) and identify the maximal possible timing constraint within the predefined ranges. This enables to specify more general timing constraints for supporting different target platforms instead of specifying absolute values for the timeouts.

This synthesis approach avoids scalability problems because, firstly, the synthesis is supported by providing details about the states involved in the communication behavior. Secondly, the synthesis problem for real-time patterns remains tractable because the single patterns limit the number of interacting roles so that only a moderate number of scenarios has to be synthesized by one run of the algorithm. In the next section we explain how restrictions can be considered when combining the synthesized patterns in a component architecture.

3. Component Behavior Synthesis

After the synthesis of the coordination patterns and their constituent parts, namely their roles, in a second step the different roles have to be mapped (manually) to the ports of a given component architecture. Our tool guarantees that connected ports in the component diagram are assigned the corresponding roles of a coordination pattern (see the example in the appendix for further details).

After each port of a component has been assigned a role (of a corresponding pattern), the overall component behavior is synthesized automatically. In a first step, our algorithm simply computes the parallel composition of all roles or, more technically speaking, of all statecharts.

However, the user may specify restrictions on sequences of states executions paths which should not occur in the synthesized component behavior, i.e. which restrict the component behavior [7]. These restrictions are given in the form of Boolean expressions or in the form of timed automata. The first possibility is used when a restriction does not include any timing constraints.

Our algorithm checks whether the given restrictions are not in conflict with the statechart behavior as synthesized in the first step from the specified sequence diagrams. If so, the user is informed and asked to resolve this conflict. If not, the statechart defining the component behavior is synthesized by changing the parallel composition of the role automata with respect to the defined state restrictions.

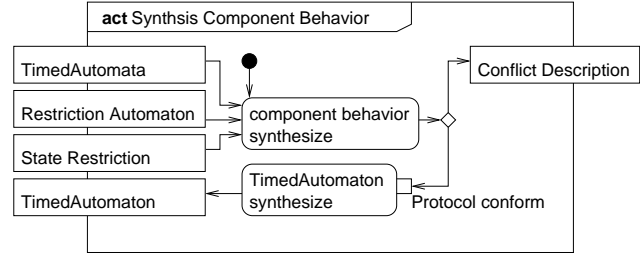


Figure 3. Overview component behavior synthesis

4. Related Work

For synthesizing statecharts from scenarios with timing constraints, only a number of limited approaches exist. The approach proposed in [12] synthesizes only global solutions in form of a single automaton for non-parametrized scenarios, which assumes angelic non-determinism² and does not support progress conditions. The approach of [9] results in a global non-parametrized timed automaton which supports progress, but requires scenario descriptions in form of trees that already introduce some of the required operational behavior. The play-out engine [6] enables the play-out of live sequence charts (LSC) with timers, but also only constructs global behavior for non-parametrized LSCs.

There exist some approaches which facilitate the synthesis of safe real-time component behavior [5]. But no one provides depending / non-orthogonal concerns.

5. Conclusion and Future Work

We have presented a synthesis approach for real-time component systems. We consider the complex real-time coordination behavior as well as the complex component behavior which results from the coordination behavior and the synchronization between the coordination behaviors if required.

In [2] we have shown the specification and simulation of reconfigurations. Based on this idea, we want to extend our synthesis approach to hybrid systems where the reconfigurations of controllers need to be synchronized with respect to the communication behavior between the system components.

Furthermore, the synthesis of the coordination patterns described here already requires detailed knowledge about the states of the role behavior. In particular, the coordination pattern synthesis yields just one solution of parameterized-time role statecharts implementing the desired coordination behavior. Thus, the component behavior synthesis may de-

²cf. [13]

tect conflicts which could be avoided with another possible implementation of the coordination pattern. For this reason, we want to investigate to which degree a more automated synthesis of the role statecharts can be supported while avoiding scalability problems.

References

- [1] 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.
- [2] S. Burmester, H. Giese, S. Henkler, M. Hirsch, M. Tichy, A. Gambuzza, E. MÜch, and H. Vöcking. Tool support for developing advanced mechatronic systems: Integrating the fujaba real-time tool suite with camel-view. In *Proc. of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA*, pages 801–804. IEEE Computer Society Press, May 2007.
- [3] 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, May 2005.
- [4] 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.
- [5] G. Gssler and J. Sifakis. Component-based construction of deadlock-free systems. In *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*, volume 2914/2003 of *Lecture Notes in Computer Science*, pages 420–433. Springer Berlin / Heidelberg, 2003.
- [6] 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 (MAS-COTS 2002)*, Fort Worth, Texas, USA, 2002. (invited paper).
- [7] S. Henkler, A. Seibel, and H. Giese. Synthesis of real-time component behavior. Technical Report tr-ri-08-296, University of Paderborn, Paderborn, Germany, Dezember 2008.
- [8] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. Document ptc/03-08-02.
- [9] 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.
- [10] W. Schäfer and H. Wehrheim. The Challenges of Building Advanced Mechatronic Systems. In *FOSE ’07: 2007 Future of Software Engineering*, pages 72–84. IEEE Computer Society, 2007.
- [11] 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 Software Engineering (FSE-10)*, Charleston, South Carolina, USA, November 2002. ACM Press.
- [12] 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.
- [13] M. Walicki and S. Meldal. Algebraic Approaches to Nondeterminism—an Overview. *ACM Computing Surveys*, 29(1):30–81, March 1997.

A Demonstration

The example stems from the railcab research project at the University of Paderborn. In this project, autonomous shuttles are developed which operate individually and make independent and decentralized operational decisions.

The modular railway system combines sophisticated undercarriages with the advantages of new actuation techniques as employed in the Transrapid³ to increase passenger comfort while still enabling high speed transportation. In contrast to the Transrapid, the existing railway tracks will be reused.

One particular goal is to reduce the energy consumption due to air resistance by coordinating the autonomously operating shuttles in such a way that they build convoys whenever possible. Such convoys are built on-demand and require a small distance between the different shuttles such that a high reduction of energy consumption is achieved. Coordination between speed control units of the shuttles becomes a safety-critical aspect and results in a number of hard real-time constraints, which have to be addressed when building the control software of the shuttles.

In this demonstration, we will first show how the role protocols are specified by real-time sequence diagrams (Section A.1.1). Then we will present the real-time statecharts which are obtained by the scenario synthesis (Section A.1.2). Thereafter, we will show how the statecharts are used in the real-time coordination pattern and how these patterns are instantiated (Section A.2.1). After that, a constraint restricting the behavior of a component will be shown (Section A.2.2). Finally, we will present the result of the component behavior synthesis based on the real-time statecharts and the constraint (Section A.2.3). All presented tools are part of the Fujaba Real-time Tool Suite.

A.1 Coordination Behavior Synthesis

A.1.1 Modeling real-time scenarios

In the first step, the role behaviors of the patterns are specified by sequence diagrams. In this example, we present two sequence diagrams: one describing the building of a convoy and one describing the break-up of a convoy.

The sequence diagram in Figure 4 shows the building of a convoy. Initially, both shuttles must be in state `noConvoy`. Then the shuttle traveling behind proposes the convoy by sending the message `convoyProposal` to the leading shuttle and waiting for an answer. The leading shuttle changes to state `Processing`. If the leading shuttle considers the convoy to be useful (`convoyUseful` evaluates to `true`), the assert block has to be executed by the semantics described in Section 2. The leading shuttle has to send `startConvoy`.

Thereafter, both shuttles are in state `convoy`. The parameter `maxLeftTimeToStartConvoy` models an upper bound for the duration between the proposal and the start of the convoy which might be unknown in this early phase. The parameters used instead of hard upper bound for supporting early phases have to be replaced by concrete values later on as described in Section 2.

The sequence diagram shown in Figure 6 specifies the break-up of a convoy. Initially, both shuttles have to be in state `convoy`. The leading shuttle proposes the break-up by sending `breakConvoyProposal`. After sending the message, the shuttle waits for the answer in state `wait`. The follow-up shuttle changes to state `Processing` and then evaluates `isConvoyUseful` to false. Then it has to confirm the break-up with `breakConvoy()`.

The sequence diagrams in Figures 5 and 7 describe how the according shuttle leaves the state `Processing` if it does not get an answer from the other shuttle within 1000 ms. The shuttle then has to return to state `noConvoy` respectively `convoy`.

A.1.2 Synthesizing real-time statecharts

After specifying the role behaviors using sequence diagrams, the synthesis is used to generate real-time statecharts. Before we can use the synthesis, we have to assign values to the parameters. As described in Section 2, valid parameter values can be computed using a constraint solver, but as the example contains no further constraints on the value of the parameter, we just set the parameter `maxLeftToSend` to 1000. The synthesis generates one statechart for each participating role, namely one for `frontRole` and one for `rearRole` in this case. The statecharts resulting from the different scenarios are merged such that identical states are identified and occur once only. During the synthesis, a consistency and timing analysis is performed on the sequence diagrams which yields no problems in this example.

The real-time statechart for `frontRole` is shown in Figure 10. It describes the behavior of the shuttle driving in front. The statechart contains the states `noConvoy`, `convoy`, and `wait` annotated in the sequence diagram. Additionally, the synthesis algorithm generates a state before and after every message sent or received. The front-role goes e.g. from state `noConvoy` to state `convoy` by receiving the message `convoyProposal` serving as a trigger, and by sending `startConvoy` modeled as a raised event. The intermediate states have empty transitions only. This part of the statechart originates from the `buildConvoy` scenario where the rest originates from the `breakConvoy` scenario. There, the `frontRole` goes from `convoy` to `wait` by sending message `breakConvoy` and finally back to `noConvoy` by receiving message `breakConvoy`.

The real-time statechart for `rearRole` is created analo-

³<http://www.transrapid.de/cgi-tdb/en/basics.prg>

gously. This statechart is very much the same as for frontRole beside that all raised events here are triggers in frontRole and all triggers here are raised events in frontRole.

The timing constraints specified in the input sequence diagrams are synthesized into the statecharts in form of clock resets and time guards with respect to the results of the constraint solver.

A.2 Component Behavior Synthesis

A.2.1 Instantiating real-time coordination pattern

The synthesized statecharts are used as state-based behavior models for the roles of the real-time coordination pattern `Convoy`. This pattern can now be instantiated in a component diagram. The roles of the pattern are assigned to ports by dragging and dropping the role onto the port. The component diagram shown in Figure 11 uses only one pattern as just the behavior attached to a role matters. In a deployed system, component instances will not execute the pattern with themselves but with preceding or following shuttles.

A.2.2 Specifying constraints

The behavior of the component is the sum of all behaviors attached to its ports in form of statecharts. In the example, we want to allow convoys of two shuttles only. Therefore, we specify a constraint that `frontRole` and `rearRole` must not be in state `CONVOY` at the same time. The restriction attached to the component is shown in Figure 11. In the constraint, `frontRole` refers to all elements of the `frontRole` statechart and `rearRole` refers to all elements of the `rearRole` statechart. The overall expression `frontRole.convoy` refers to the state `convoy` of the `frontRole` statechart. So, the constraint that `frontRole` and `rearRole` must not be in a `convoy` is modeled such that the component must not be in states `frontRole.convoy` and `rearRole.convoy` at the same time.

The constraints are specified for a component and not for a pattern as the constraints are used to synchronize behavior of multiple pattern instantiations in one component. It is possible to use one set of restrictions for all components as restrictions have no effect if the restricted state combinations do not occur in a component.

A.2.3 Synthesizing component behavior

The last step is to synthesize the component behavior based on the statecharts assigned to the ports of the component and the specified restrictions. Hence the statecharts of the ports (Section A.1.2) and the specified restriction (Section A.2.2) are the input for the synthesis algorithm. The input statecharts (Figure 9 and Figure 10) are simplified for this part of the example by removing trivial transitions and states in order to reduce the statespace of the parallel composition.

In a first step the synthesis algorithm computes the parallel product of the input automata. In a second step of the algorithm the state restrictions are applied resulting in the removal of state `frontRole.convoy_rearRole.convoy` and all related transitions. Accordingly, if the shuttle is in state `frontRole.processing_rearRole.convoy` it is not possible to switch to `frontRole.convoy_rearRole.convoy`. Instead the `rearRole` part of the shuttle has to leave `convoy` first, before the `frontRole` is able to switch to `convoy` again.

In the third step the synthesis algorithm checks if the observable behavior of each input automaton is still present in the restricted parallel composed automaton. As this is the case in our example, the result is a restricted component behavior which is still conform to the behavior specifications of each instantiated pattern role in this component. Because of lack of space the final parallel composed statechart is not shown here.

B Screen dumps

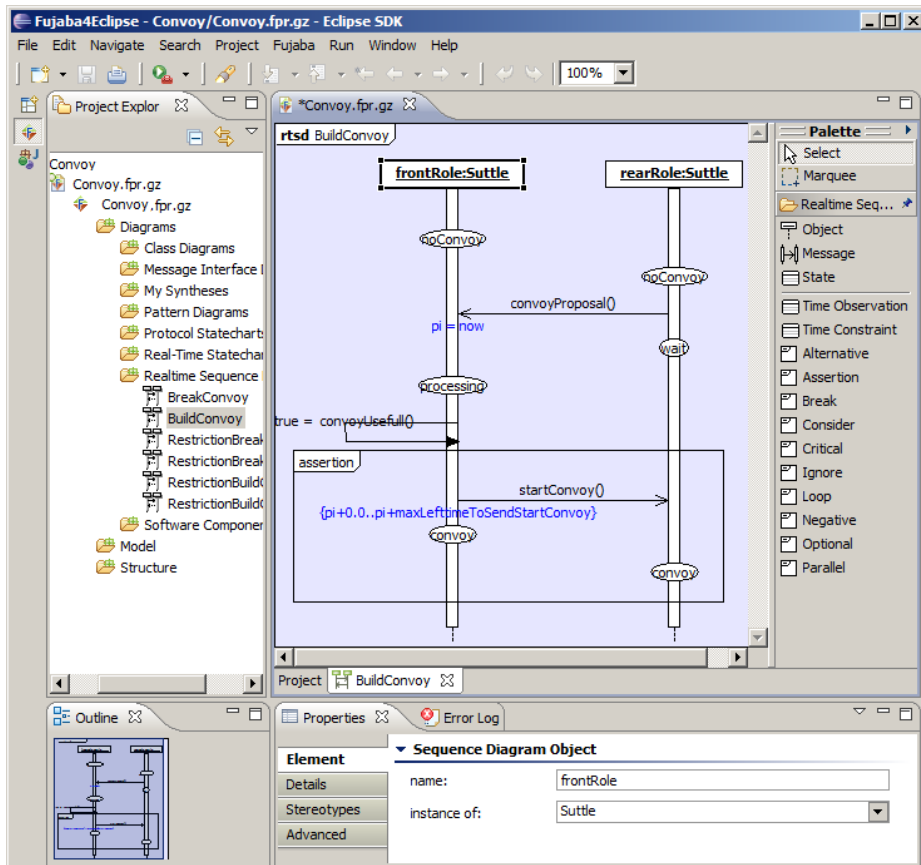


Figure 4. Scenario for building convoy

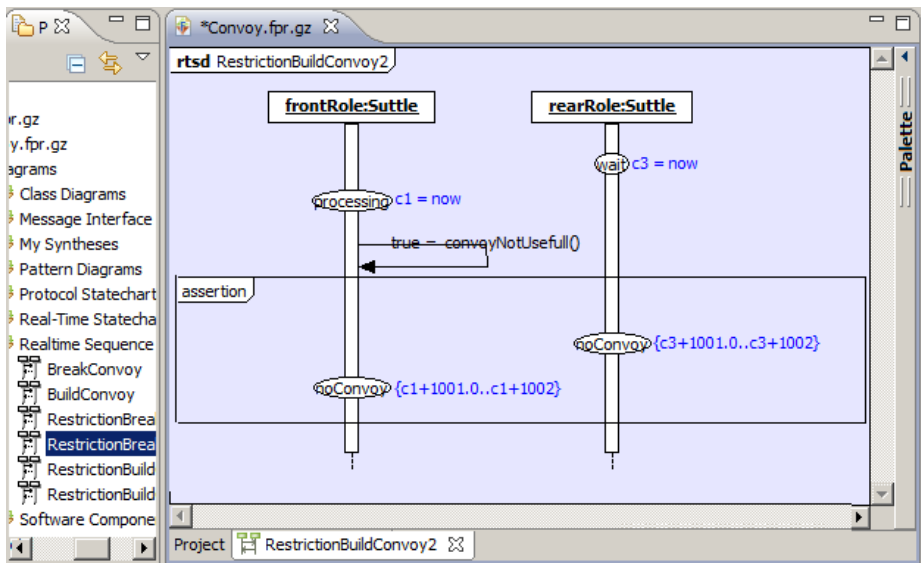


Figure 5. Scenario for not building convoy

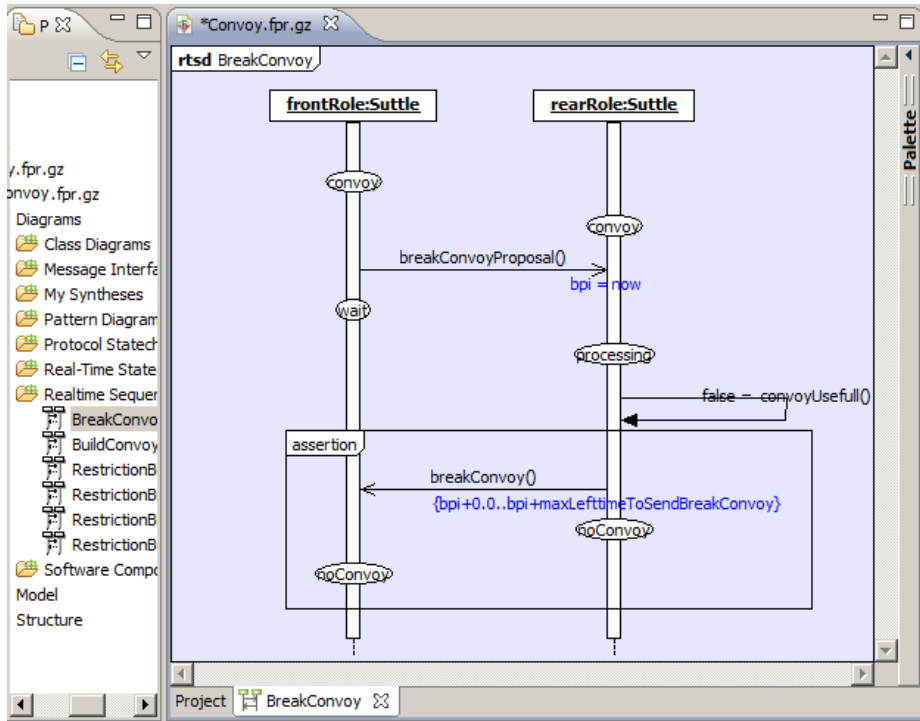


Figure 6. Scenario for breaking convoy

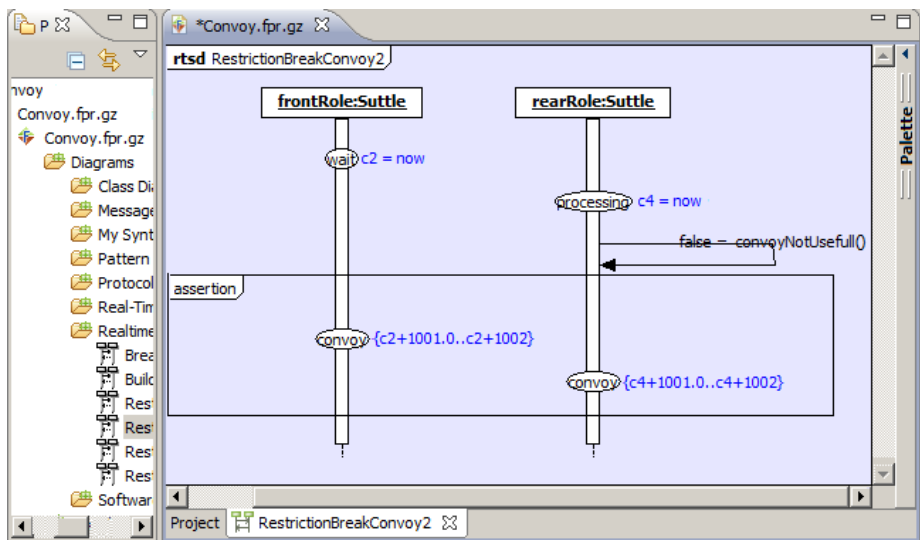


Figure 7. Scenario for not break convoy

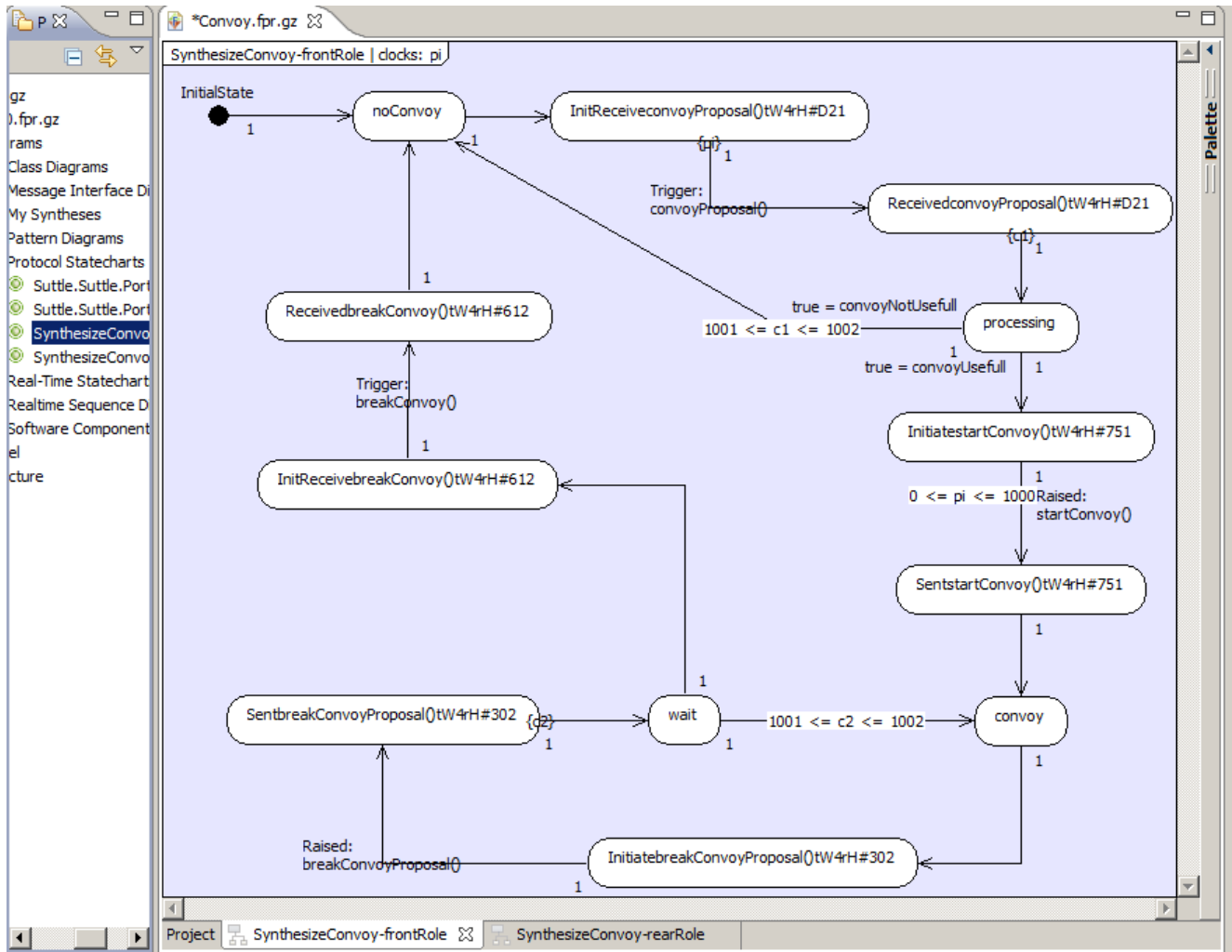


Figure 10. Synthesized behavior for frontRole

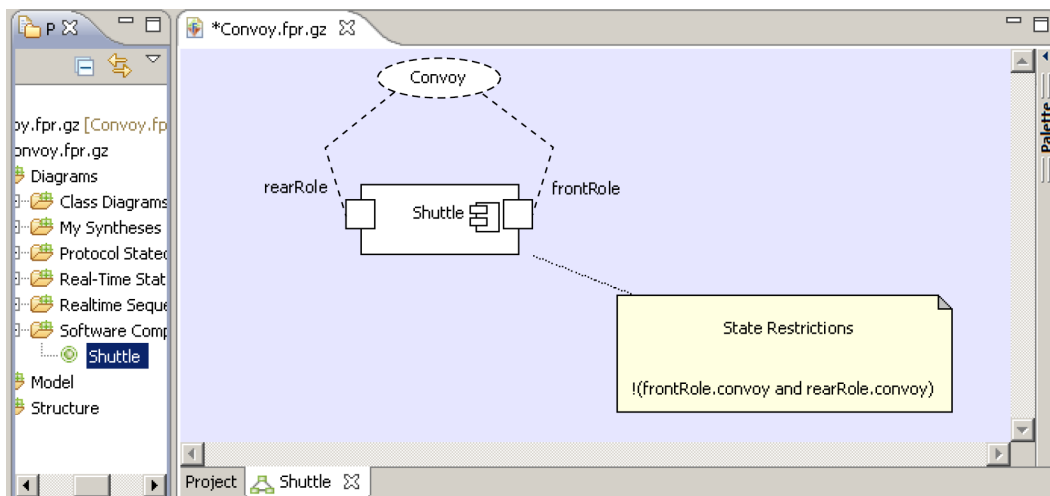


Figure 11. Instantiated pattern in a component diagram