Legacy Component Integration by the Fujaba Real-Time Tool Suite*

Stefan Henkler, Jan Meyer,
Wilhelm Schäfer, and Markus von Detten
Software Engineering Group, s-lab (software quality lab), Heinz Nixdorf Institute
Warburger Str. 100
Paderborn, Germany
shenkler,janny,wilhelm,mvdetten@upb.de

Ulrich Nickel
Hella KGaA Hueck & Co.
Rixbecker Str. 75
Lippstadt, Germany
Ulrich.Nickel@hella.com

ABSTRACT

We present a Tool Suite which supports the (re-)construction of a behavioral model of a legacy component based on a learning approach by exploiting knowledge of known models of the existing component environment. This in turn enables to check whether the legacy component can be integrated correctly into its environment.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Verification

Keywords

Legacy System, Model-Driven Engineering, Integration, Formal Verification, Safety-Critical Systems

1. INTRODUCTION

The software of complex embedded systems, like automotive and aerospace systems, is usually a network of components. We assume that the behavior of a single component basically consists of the communication behavior on the one hand and the controller behavior, i.e. controlled feedback loops taking sensor input and producing actuator control signals on the other hand. Communication behavior enables to exploit knowledge of other components in order to enhance the functionality and to adapt the behavior of a single component when beneficial. Adapting the behavior might require reconfigurations of controllers in the form of mode management and control algorithms under hard real-time constraints.

As many of these systems are used in a safety critical environment, high quality software is absolutely necessary. To fulfill this requirement, model-driven engineering has become the means to construct reliable software. These techniques enable the developer to use simulation and formal mathematical methods to verify critical system properties.

However, in domains like the automotive industry the development of new functions is an exception rather than the regular case. In many cases, components exist and have to be reused (for cost efficiency reasons) where no model or only a corrupted model exists.

In addition, with the development of new standards like e.g. AUTOSAR[1], the topic "software as a product" is promoted. This means that a functional network consists of components from different producers. These supplied or bought components must be integrated into the electronical control units by the automobile manufacturers (OEM) or the suppliers. For the integration it is necessary that the component meets the specification. Today this can only be checked during the integration test phase. But the integration has to be done and checked in an earlier development phase. Therefore, the integration of legacy components into a model-driven engineering approach is obvious. This requires the existence of an adequate model of the legacy system.

The main goal of our integration approach is thus to produce a model of the communication behavior of a single legacy component based on a real-time statechart. Real-time statecharts are an abstraction of timed automata supporting more comprehensive specifications. In addition, the approach also allows to reconstruct the transfer functions of the embedded controllers. Finally, the produced model is used to check whether a correct integration with its environment, namely the newly built components is possible.

In this paper, we present the $FRiTS^{Cab}$ Tool Suite[2]. The tool applies an iterative learning approach to (re-)construct the real-time statechart specifying the communication behavior of a single component. Iterative learning speeds up the (re-)construction, because it checks after each step of the learning algorithm whether the (re-)constructed statechart of the legacy component communicates correctly with the given statechart(s) of the newly built component(s).

*This work was developed in the course of the Collaborative Research Center 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

ICSE '10, May 2-8 2010, Cape Town, South Africa

[1] www.autosar.org

[2] $FRiTS^{Cab}$: Fujaba Re-EngIneering Tool Suite for Mechatronic Systems. FRiTS: Fujaba Re-EngIneering Tool Suite, Cab: short form of RailCab (http://www-nbp.uni-paderborn.de/index.php?id=2&L=1), an example of an advanced mechatronic system. $FRiTS^{Cab}$ is part of the Fujaba Real-Time Tool Suite (http://www.fujaba.de/projects/real-time.html)

Correct communication is based on predefined safety and bounded liveness properties which have to hold for that particular communication. If not, the algorithm can backtrack right away and try another possibility.

Furthermore, we support the identification of controller behavior by the integration of classical system identification approaches to describe the in- and outgoing controller behavior by transfer functions. If all transfer functions are known, we are also able to identify reconfigurations.

In the next section we introduce our legacy checking approach. Thereafter, we give an overview of the related work and finish with a conclusion and future work. As an example of an embedded system, we use a wiper control system to demonstrate the practical relevance of our approach.

2. LEGACY CHECKING

In our modeling approach called MECHATRONIC UML (e. g. [5]), the architecture is given by components (see Figure 1, e. g. WiperCoordination and Wiper), their ports and the connections between ports. This model is formally described by an adaptation of the UML 2.0 component model. Communication between components is defined by so-called *coordination patterns* (WiperCoordination). A coordination pattern describes the communication between two components and consists of communication partners, called *roles* (coordination and wiper), i. e. it can be considered as a particular type of protocol specification. The communication behavior of a role is specified by a real-time statechart. Figure 2 shows a cutout of the coordinator role behavior.

Real-time statecharts are an extension of UML state machines which support more powerful concepts for the specification of real-time behavior. They are semantically based on timed automata such that a formal analysis is possible using the model checker UPPAAL[3].

Integrating a legacy component like (Wiper) without an available statechart specification of the communication behavior requires to derive such a model from the component interface and possibly also from the available source code. In addition, such a model must fulfill all liveness and safety properties like e. g. coordination.off implies wiper.off which are required for communication between the legacy component and the communicating component(s) behavior, in this example the coordinator role behavior. In this example, basically "one half" of the communication protocol is known and it has to be checked whether the communication behavior of the legacy component fits to the "other half" as defined by the (WiperCoordination) pattern.

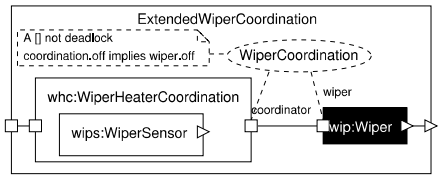

Figure 1: Component Architecture

Overview As a prerequisite of our approach a legacy component has to provide a proper interface definition, i. e. all incoming and outgoing events used for communication, as well as all signals used and produced by the controllers, and all relevant information for the execution of the legacy component (e. g. like the period).

[3]www.uppaal.com

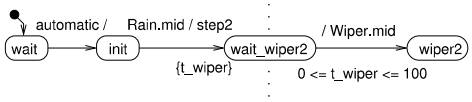

Figure 2: Cutout context behavior

Our approach includes three different algorithms which support (re-)construction of the statechart, if 1) the legacy component has additional interface operations to retrieve the current state during execution [4] or, 2) this is not the case. For case two we distinguish in turn between a) a black-box (no source code is available) or b) a white-box (source code is available) approach. We call the corresponding algorithms in case of 1) gray-box-checking, in case of 2a) black-box-checking, and in case of 2b) white-box-checking.

The correctness of the integration, i. e. component communication, can either be shown 1) by a formal verification of the legacy component communication with the context (e.g. coordination) or 2) by a verification whether the (re-)constructed legacy component (e.g. Wiper) behavior is a correct refinement of the abstract role behavior (e. g. wiper) given by the coordination pattern. It has to be shown that $coordination||learned\ \ wiper \models coordination.off \ \ implies \ \ learned \ \ wiper.off \wedge \neg\delta \ \ (no; deadlock)$ for 1) and $learned\ \ wiper \leq wiper$ for 2).

A refinement could also be shown by a parallel composition if a test automata is derived from the role behavior (e. .g the wiper role) [9]. A test automata is derived by building the complement of the abstract model (wiper). Additionally, the behavior which is not fulfilled by the automata is taken into account by an extra failure state. The analysis should show that this error state is not reachable.

2.1 Gray-Box-Checking

Our approach extends the current knowledge of the legacy component with chaotic behavior, resulting in a new model called *chaotic closure* [7, 4]. Initially, we assume current knowledge to be a single state automata, identifying that the legacy component is in a quiescent state (see Figure 3). For all so far unknown behavior it is assumed that on the one hand *any* possible interaction may occur but on the other hand a deadlock is possible at any time as well. Therefore, the chaotic closure is an over approximation of the legacy component: it always models at least all of the legacy components' behavior, but not all of the modeled behavior has to be possible in the legacy component. The chaotic closure is then in combination with a model of the context subject to model checking by taking safety and bounded liveness properties into account (step 1) and 2) in Figure 3). If we have a counterexample, we use this as test input for the legacy component (step 3)). If the tested faulty run is confirmed, we have found a real counterexample. If not, we use the new observed behavior to refine the previously employed behavior model of the legacy component (step 4)). We repeat the checks until either a real counterexample has been found or all relevant cases have been covered.

For modeling chaotic behavior a *chaotic automaton*, a nondeterministic finite automaton consisting of two states is used: The state s_δ with no outgoing transitions represents the case of the legacy component being in a deadlock, neither receiving nor sending any messages. The state $s_\forall$ on the contrary represents the case where all inputs being possible for the legacy component are enabled and all outputs can occur. This is modeled by one self-

[4]For example, AUTOSAR components provide this information

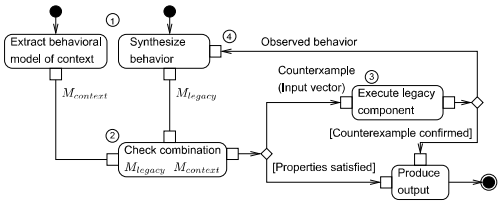

Figure 3: Counterexample guided learning

transition and one transition to s_δ for each possible input (with no output) and each possible output (with no input). For creating these transitions the input- and output-alphabets of the legacy component must be known. Both states of the chaotic automaton are initial states.

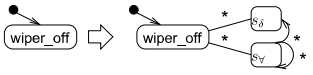

Figure 4: Example for a chaotic closure

The chaotic closure is a combination of a synthesized model with the chaotic automaton for the legacy component, mapping all unknown behavior to a chaotic one. Figure 4 shows as an example of a chaotic closure (on the right side) for a trivial first conjectured behavior model.

A chaotic closure is constructed as follows: First, the chaotic automaton for the input- and output-alphabets of the legacy component is constructed. Then, the states and transitions of the chaotic automaton are added to the incomplete automaton modeling the behavior that has been learned until now. For every combination of a state and an incoming or outgoing event for which a transition neither has been defined nor excluded, a new transition is created from that state to both the $s_\forall$ and the s_δ state. Contrary to the synthesized behavior, the chaotic closure constructed for it is non-deterministic.

The explicit deadlock state s_δ in the chaotic closure makes sure that as long as there still is behavior left to learn, the model checker will be able to find a deadlock. The result is that in every iteration of our approach at least one new transition is learned. However, this only applies to behavior of the legacy component which can be reached in combination with the model of the context. Any other behavior is not considered to be relevant in the context the legacy component is integrated into, and therefore no time needs to be wasted with testing it.

2.2 Black-Box-Checking

Compared to the gray-box-checking approach, 1) we first have to learn a candidate of the legacy component based on Angluin's algorithm [1][5], which is a minimal automaton, 2) the candidate is in combination with a model of the context subject to model checking by taking safety and bounded liveness properties into account (see Section 2.1), 3) if the check is successful, it is proven if the candidate is equivalent with the legacy component (conformance tests by Vasilevskii and Chow [11, 2]). Otherwise the counterexample is input for step 1), and 4) if the candidate is equivalent, the black-box-checking is finished.

To learn a candidate for step 1), we extend Angluin's algorithm for the domain of embedded systems [4, 6]. That means especially that we have to take into account in- and outgoing messages and time.

To support in- and outgoing messages, we extend Angluin's algorithm, which supports DFA, by mealy machines similar to [8]. That means, the simple tracing of accepted words by the algorithm of Angluin is extended to capture output sequences.

In principle, a legacy component of the considered domain has to react deterministically. The reaction just in time is of paramount importance. Hence, these systems are implemented periodically. A timely periodic execution is a prerequisite for a correct functionality. The specification of the period of the legacy component is mandatory for the integration of such components. Our learning approach exploits this fact to check timing constraints which are discretized by a (multiple of a) period of a legacy component. This is enabled by adding specific empty words to the learning algorithm which has the meaning of waiting for an answer for a specific time. We know the maximum waiting period for an event as the upper bound is given by the legacy component.

Besides the required in- and outgoing messages and time, we extend Angluin's algorithm additionally by taken the context into account to learn and check iteratively the relevant behavior of the integration. This is similar to the counterexample guided learning presented in Section 2.1. The main difference is that a successful model checking must be additionally confirmed by equivalence checks of the learned automata and the legacy component to be sure that the learning is finished.

If we know or estimate well the upper bound of the number of states for the equivalence checks, the approach ensures either to find a conflict in the integration or the integration is correct (with respect to the constraints).

2.3 White-Box-Checking

To show the correctness of the integration for the white-box case, we use existing source code model checker to take profit of its abstraction capabilities. Compared to the gray- and black-box approach, we are not able to learn and check iteratively the legacy behavior as typically the internal model of a source code model checker is not visible.

The main task for the white-box case is to generate source code of the context behavior for a source code model checker in a proper manner to support a compositional verification with the legacy component. Furthermore, we support a run-time framework which implements the execution semantics of real-time statecharts. The generated source code, the framework source code, the legacy component source code, and constraints are the input of a source code model checker (see Figure 5).

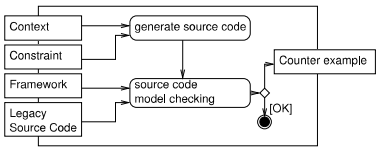

Figure 5: White-Box-Checking Overview

In a first step, we have to map the (abstract) model to source code to enable the required check (with the legacy component) [6]. The mapping has to preserve the execution semantics of the model. Hence, one possible (deterministic) path of the model has to be mapped to source code.

As the considered kind of systems are reactive once the generated system is executed periodically (that is also the case for the legacy component). The WCET (worst case execution time) of a

[5]Angluin's algorithm [1] is a popular and efficient approach for learning a DFA of a black-box.

transition has to satisfy the specified deadlines. Within a period a task can be executed nondeterministically by the scheduler. Furthermore, the periods of the legacy system and the context can be of different length as well as the sending and receiving of events could be at some point during the period (see Figure 6).

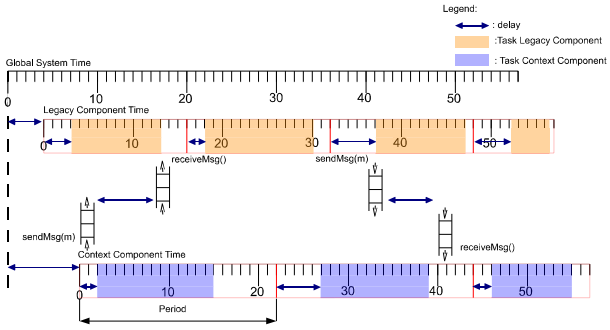

Figure 6: Overview Scheduling

Our Tool Suite supports a C code generation which considers the discussed requirements. Based on the generated code and a runtime framework which encapsulates the communication calls as well as a simulated time, we can start the proof with a source code model checker. Conceptually the BLAST model checker[6] fits the best as this model checker supports a good abstraction based on lazy abstraction. But, in our evaluation BLAST had a lot of problems with bounded arrays which are required for the communication. The bounded model checker CBMC[7] supports the most C constructs and also supports the required bounded arrays. Hence, we use CBMC for our evaluation. To support nondeterministic constructs of the scheduler, e. g. the execution of a task can be arbitrary within a period, we use the concept of non-deterministic variables to specify an arbitrary execution within a specified time interval.

2.4 System Identification

System identification [10] is the approach which enables the identification of continuous behavior. This is done by simulation. We can simulate each path of the learned behavior and for each state we can identify the controller behavior. The input of the system identification is a specified test trajectory or a realistic run in its environment. Based on the input and output behavior the transfer function of the controller is identified for linear systems. If the transfer functions are known, we can identify reconfigurations by different transfer functions. This approach supports the engineer in integrating embedded (controller) components in a system model in early development phases. Typically, the engineers test the legacy components (controllers) only in hardware-in-the-loop scenarios or the real environment later in the development process. We have integrated the MATLAB System Identification Toolbox[8] in our tool suite [6].

3. RELATED WORK

A number of techniques which either use a black-box approach and automata learning (e. g. [8]) or a white-box approach which extracts the models from the code exist (e. g. [3]). However, these approaches did not consider the specific context and safety requirements for (efficiently) synthesizing the relevant behavior of the legacy component, which is of paramount importance for embedded systems. Furthermore, these approaches are not capable of finding conflicts in early learning steps.

[6]http://www.cs.ucla.edu/ rupak/blast/
[7]http://www.cprover.org/cbmc/
[8]http://www.mathworks.com/products/sysid/

4. CONCLUSION AND FUTURE WORK

In this paper we have presented a tool support for the incremental synthesis of communication behavior for embedded legacy components by combining compositional verification techniques, model based testing and learning techniques. It enables context specific learning with conflict detection in early learning steps. Furthermore, we enable the engineer to identify controller behavior and its reconfiguration, which, in turn, supports an early conflict recognition. The presented legacy checking approaches cover a wide range of techniques required for integrating legacy components with different supported information in a system model. Our evaluations in industry and the RailCab project confirm this statement.

5. REFERENCES

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[2] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.

[3] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[4] H. Giese, S. Henkler, and M. Hirsch. Combining Compositional Formal Verification and Testing for Correct Legacy Component Integration in Mechatronic UML. In R. de Lemos, F. D. Giandomenico, C. Gacek, H. Muccini, and M. Vieira, editors, *Architecting Dependable Systems V*, volume 5135 of *LNCS*, pages 248–273. SPRINGER, 2008.

[5] H. Giese, S. Henkler, M. Hirsch, V. Roubin, and M. Tichy. Modeling Techniques for Software-Intensive Systems. In D. P. F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*. Langston University, OK, 2008.

[6] S. Henkler, M. Breit, C. Brink, M. Böger, C. Brenner, K. Bröker, U. Pohlmann, M. Richtermeier, J. Suck, O. Travkin, and C. Priesterjahn. $frits^{Cab}$: Fujaba re-engineering tool suite for mechatronic systems. In P. V. Gorp, editor, *Proceedings of the 7th International Fujaba Days*, pages 25–29, Eindhoven University of Technology, The Netherlands, November 2009.

[7] S. Henkler and M. Hirsch. Iterative behavior synthesis by combining formal verification and model-based testing. In *Postproc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany*, pages 39–51, 2008.

[8] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *In Proc. 15 Int. Conf. on Computer Aided Verification*, 2003.

[9] H. E. Jensen, K. G. Larsen, and A. Skou. Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT '00: Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–30, London, UK, 2000. Springer-Verlag.

[10] L. Ljung. Perspectives on system identification. In *Proc. 17th IFAC World Congress*, Seoul, Korea, July 2008. (Plenary session).

[11] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics and Systems Analysis*, 9(4):653–665, July 1973.

APPENDIX

A. DEMONSTRATION

We exemplify our tool suite by an application of the automotive industry. We consider the problem of integrating a legacy windshield wiper and windshield heater component in a wiper-heater component. This extends the functions of the wiper and heater by an automatic control based on inputs of sensors for the temperature and the rain intensity.

To remove fast wetness or ice (a film) from the windshield a heating is required. Typically, the maximal possible temperature is limited to 70° Celsius. Conventionally, the maximal heating is heavily reduced to avoid exceeding this temperature. With a temperature sensor and an automatic control of the heating, the cleaning of the windshield can be speed up as the temperature can be precisely adjusted for different situations.

In combination with a rain(-light) sensor and an automatic control of the wiper the cleaning process of the windshield is completely automated and improved in its speed compared to the manual adjustment of the wiper and heater. Especially in critical situations like a spindrift from other cars or changing snow-ratio, the system can react faster than an (appalled) driver. A robust real-time automatic film detection supports the driver and this way significantly increases driving safety and comfort.

For reasons of simplification, we present in more detail in the demo the integration of the wiper component and the automatic control of this component. The implementation of the wiper component is based on a patent [13]. This includes the specification of in- and outgoing events for triggering specific wiper levels and the transfer function of the wiper (and washer) controller. To exemplify the different analysis tools, we consider the wiper component as a gray-, black-, and white-box component.

In this demonstration, we first show how the structure of the component and network is specified by component diagrams and a real-time coordination pattern (Section A.1.1). Furthermore, we will show how we embed legacy components in such a component architecture. Then we will present the behavior of the context specified by real-time statecharts (Section A.1.2). Thereafter, we will show how we iteratively learn and check the coordination behavior of the wiper component by our gray- and black-box-checking tool (Section A.2). After that, we analyze the source code by our white-box-checking tool (Section A.3). Finally, we will present the identification of the controller behavior (Section A.4). All presented tools are part of $FRiTS^{Cab}$, a Tool Suite in the Fujaba Real-time Tool Suite.

A.1 Modeling

A.1.1 Structural Specification

The first step is to specify the architecture of the system, which is done with a component diagram as visualized in Figure 7. The system consists of legacy components Wiper and Heater. A specific attribute for the specification of the legacy component is the destination of the legacy component (the source-code or the executable). The component WiperHeaterCoordination implements the automatic control of the Wiper and Heater. The WiperHeaterCoordination component embeds the RainSensor and TemperatureSensor component (see Figure 8). The continuous output of the sensor is visualized by a triangular port.

The network infrastructure is specified by the WiperCoordination pattern (see Figure 9) which specifies the communication protocol. The communication pattern has the roles coordinator and wiper which are instantiated by the WiperHeaterCoordination component (instantiating the coordinator port) and the Wiper component (instantiating the wiper port). The communication between the roles is bidirectional. The behavior of the roles, which belong to the pattern, is specified by real-time statecharts (RTSC) (see Section A.1.2). The WiperHeaterCoordination component refines the coordination role. As the protocol behavior of the Wiper component is not known, the communication is unsafe. We have to show that the Wiper component is a correct refinement of the wiper role.

An ATCTL formula specifies the requirement of the pattern: No deadlock may occur (A[] not deadlock) and the situation, that the coordination is in state off while the wiper is active, may never occur (coordination.off implies wiper.off). As mentioned before, the correctness of these constraints is unknown, as the protocol behavior of the Wiper component is unknown.

A.1.2 Behavioral Specification - Context

In our application, the context is given by the coordination role. The behavior of the role is shown in Figure 10. The coordination role is initialized when the driver/user selects the automatic mode. The application distinguishes between three intervals of values from the rain sensor: Rain.high, Rain.mid, Rain.low, and Rain.off. Based on these signals an event is send to the Wiper. If signal Rain.high is received event step3 is send to the Wiper, if signal Rain.mid is received event step2 is send to the Wiper, and so on. The coordination role then waits 100 time units for an answer of the Wiper. In case of sending the step3 event, the coordination waits 100 time units for the Wiper.high event. Accordingly, in case of sending the step2 event the coordination role waits for the acknowledgment of the Wiper in form of the Wiper.mid event, and so on. If the acknowledgment of the Wiper role is not received within the time-guard $0 \leq t_{wiper} \leq 100$, the coordination role signalizes an error in form of sending the event warning_on. This event could for example trigger an error-lamp of the dashboard and record the error trace. We did not specify this situation in more detail. The user must then restart the system by triggering the automatic mode again. The coordination switches between the different wiper steps (states wiper_0 to wiper_3) if another sensor signal is received. From each state of the coordination role it is possible to trigger the washer. After 5 time units a wash.off event is automatically send to the washer and the coordination role switches back to the last state (because of the history state).

A.2 Gray- and Black-Box-Checking

Now, as the Wiper component is embedded in its (new) system architecture and the context is specified, we are able to apply our legacy checking approaches (see Figure 7). To start the legacy checking, some specific information for the legacy component is required, like the in- and outgoing events and the initialization (-method) of the legacy component. This information can be specified in the legacy component editor.

The execution environment in our demo is a simulated one, which means that time for example is simulated by our verification framework. This is enabled by specific knowledge of the worst case execution time (WCET) of a legacy component's function (which can be computed by standard WCET analysis tools a priori). Our simulated time uses this knowledge to compute the possible time progress. As the time consumption of a function is not fix, we implement the time consumption by non-deterministic variables (see Section 2.3).

As the user interface of the gray- and black-box-checking tools is similar, we present both parts in the demo section of this paper in one section. The main difference is that the gray-box approach

shows a real-counterexample or the learned behavior at the end while the black-box approach could also learn the behavior of the protocol if a real-counterexample is found. A real-counterexample means that the determined counterexample is confirmed by the legacy component.

In the presented example, the gray- and black-box-checking tool detects a real-counterexample after some learning steps. The event sequence step1,low/step2,mid/off,-/- pinpoints a deadlock. The cause is, as shown by the learned behavior (see Figure 12), that the legacy Wiper component does not allow switching from each step back to the off state. Based on the patent description this is not clearly specified. This kind of failure could cause a danger for the environment / the driver as the wiper would not be able to switch to the off state if it is in step2 or step3 and the Wiper off signal is triggered. As the legacy component has shown their quality, we have to adapt the context behavior to enable an integration. In our example, we have to disable to switch directly from each Wiper step to the off state. E. g. the off event is connected through the next lower Wiper steps.

While the gray-box-checking tool found the counterexample within a few seconds after some learning steps, the black-box-checking tool sends about 500 hundred queries to the legacy component to identify a state behavior. This could be very expensive if the period of the legacy component is long. If the period is 10 seconds for example than the learning algorithm based on Angluin (see Section 2.2) needs about 5000 seconds only for the membership queries. Positive is that our implementation detects about 600 cache hits and about 7000 prefix cache hits in the shown scenario. As our approach also takes the context into account, we can learn the relevant behavior much more goal oriented. In cases that we only want to find a counterexample, we can improve/reduce the computation time drastically with respect to the classical Angluin algorithm[9]. However, if also the complete behavior should be learned if a counterexample is found, we have to check the equivalence by the Vasileskii and Chow algorithm. This is one of the most popular conformance test algorithms which can show the equivalence indeed. The runtime however, is exponential with the maximum number of states of the legacy component. As we consider a compositional approach, where each protocol behavior has only a small number of states typically, all this drawbacks have no consequences. In our example the runtime of our black-box-checking approach is about two minutes.

A.3 White-Box-Checking

To start the white-box-checking approach, the ports wiper and coordination have to be selected. Using the context menu (see Figure 7), the dialog for the adjustment of the parameters of the white-box-checking can be called (see Figure 13). The dialog enables the user to adjust different parameters for different source code model checkers, like CBMC and BLAST. These parameters are used as the input for the code generation of the context (coordination role) which is required to enable the model checking of the source code of the legacy component. As presented in Section 2.3, the source code of the context, the source code of the legacy component and a verification framework, which simulates the semantics of RTSCs, are inputs for the source code model checker. The parameters specify for example, the length of the context's period and the legacy component. The code generation of the context is triggered when the start button of the white-box-checking properties dialog is pressed.

If the verification is successful the correctness is determined (as in typical model checking approaches). If an error is detected a counterexample is presented by a dialog. In our case, the counterexample shows the scenario as presented by the gray- and black-box-checking. Additionally, the counterexample can be animated in the context behavior.

[9]As explained in Section 2.2, we also have to extend the Angluin algorithm to learn the behavior of the considered system.

A.4 System Identification

System identification enables to identify continuous behavior of a system. In our case, it enables to identify controller behavior. In general we would have to identify the behavior of a hybrid system. Our approach however, uses the knowledge of learned state behavior (see Section 2.4). Based on this state behavior, we identify the controller behavior for each state. As the controller behavior of a state could be different for each incoming transition, we simulate all possible situations a state could be switched. For each situation, we start a diagnosis of the legacy controller behavior. This could be some specific test trajectory or, as in our case, a standard diagnosis trajectory supported by the Matlab System Identification Tool Box. We record for each diagnosis the test trajectory and the output of the system. Then, we load this data into our tool suite. Figure 14 depicts the situation where we have loaded two different data (data_controller1 and data_controller2) which are connected to the legacy Wiper component by in- and out-going continuous ports. We start the system identification for each loaded data (by triggering the Matlab System Identification Tool Box). The identified behavior (identificationResult) is presented as a transfer function (see Properties Element). Additionally a fitness function is computed. This shows the correlation of the estimated model output compared to an output data set of the system in order to ensure that the estimated model represents the system dynamics accurately. Next, this transfer functions can be analyzed by engineers to identify specific controllers for example (in our case the WiperController and the WasherController). Finally, we attach the results of the system identification to the corresponding state. State q2 to state q4 for example, embed the WiperController and state q5 embeds the WiperController and WasherController.

B. SCREEN DUMPS

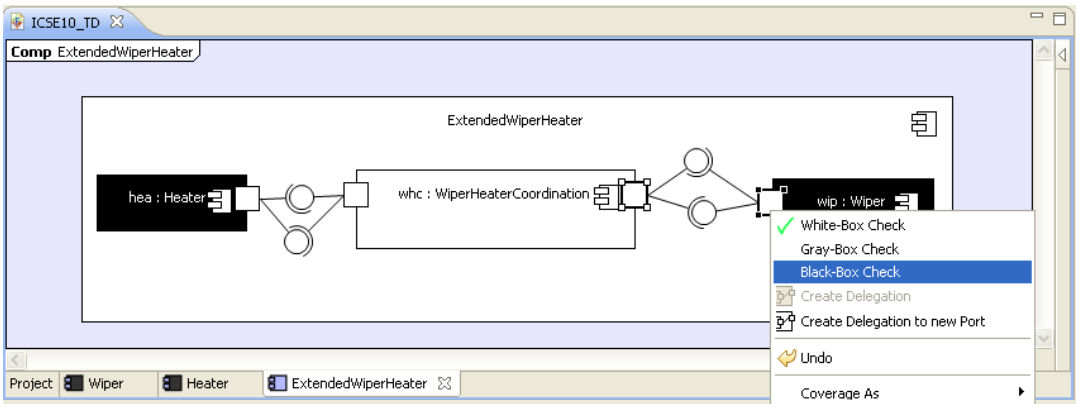

Figure 7: Structure of the ExtendedWiperHeater component

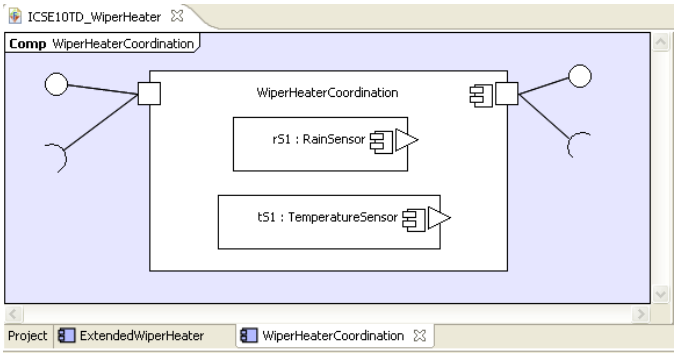

Figure 8: Structure of the WiperHeaterCoordination component

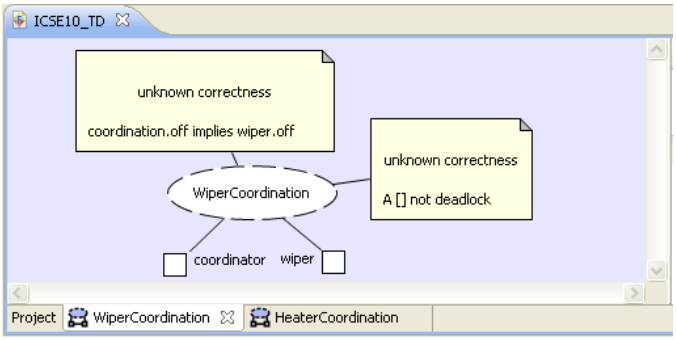

Figure 9: Structure of the WiperCoordination pattern

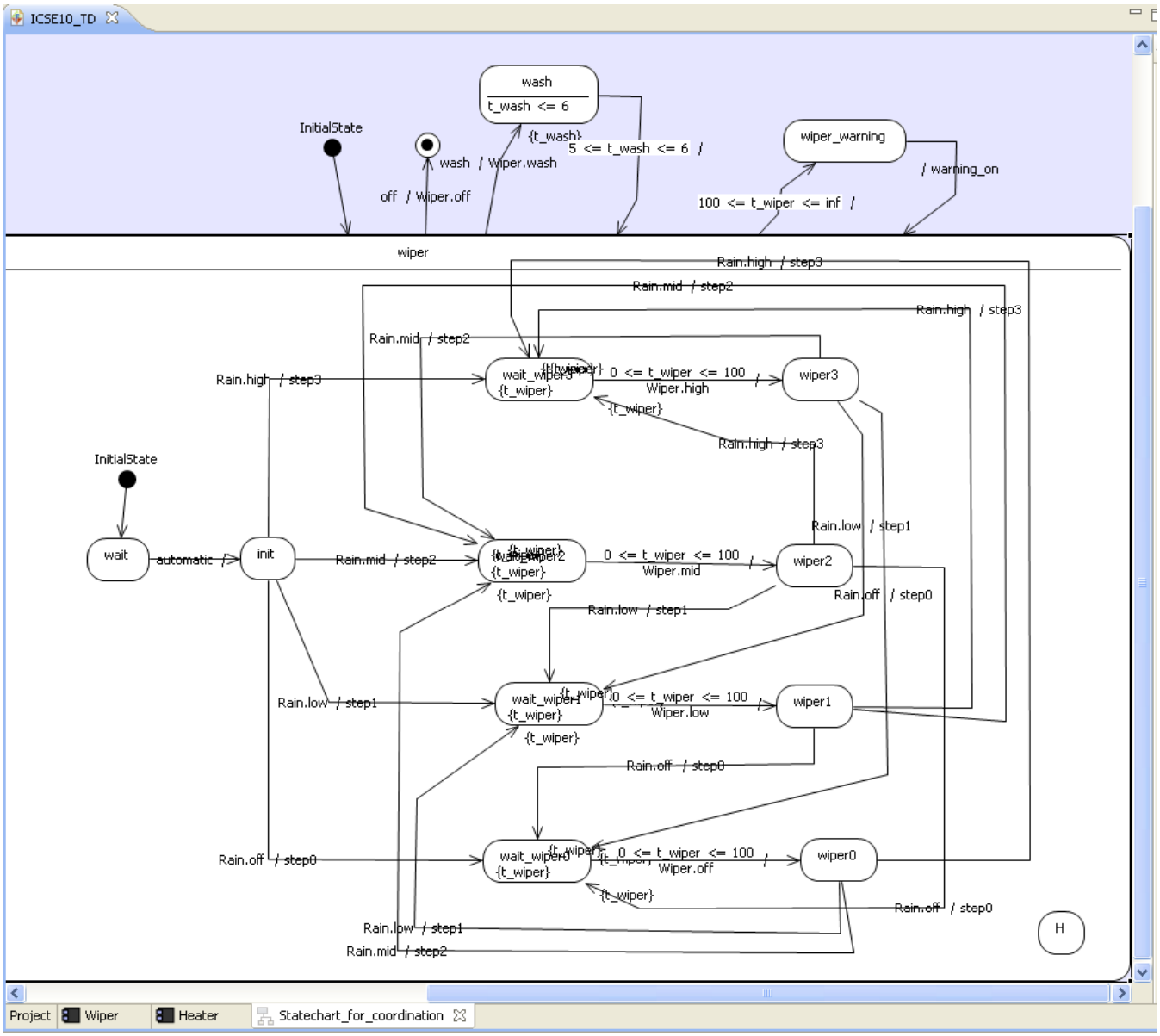

Figure 10: Context behavior of the coordination role

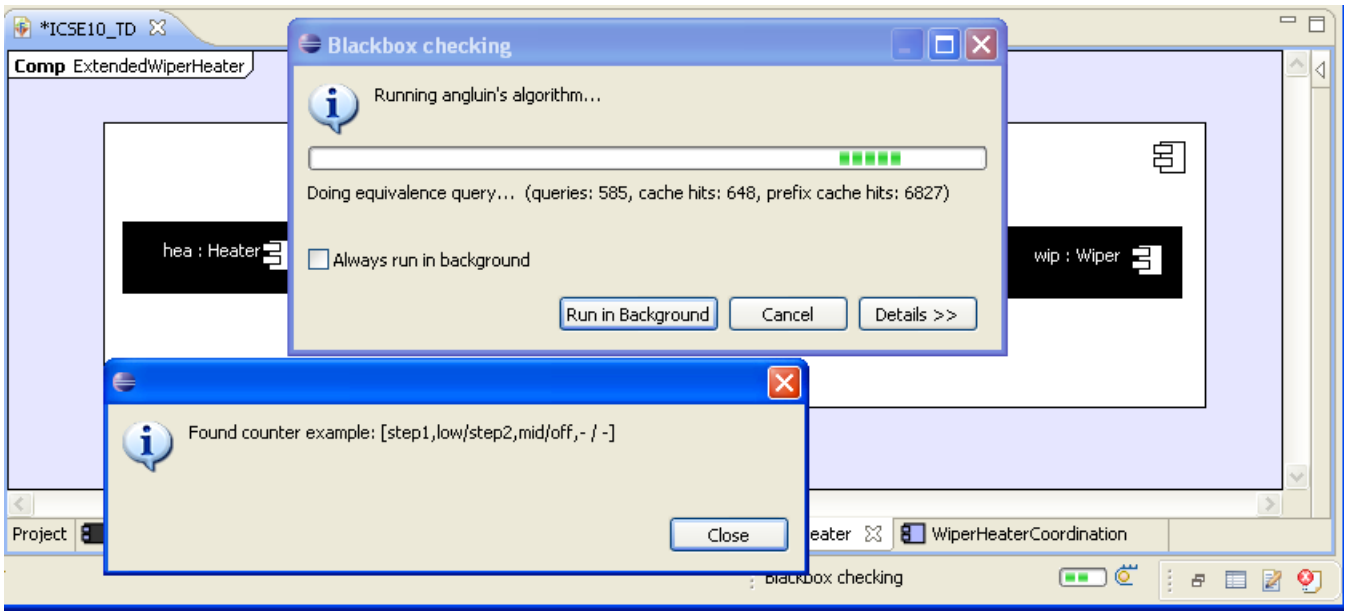

Figure 11: Black-Box-Checking: progress and counterexample

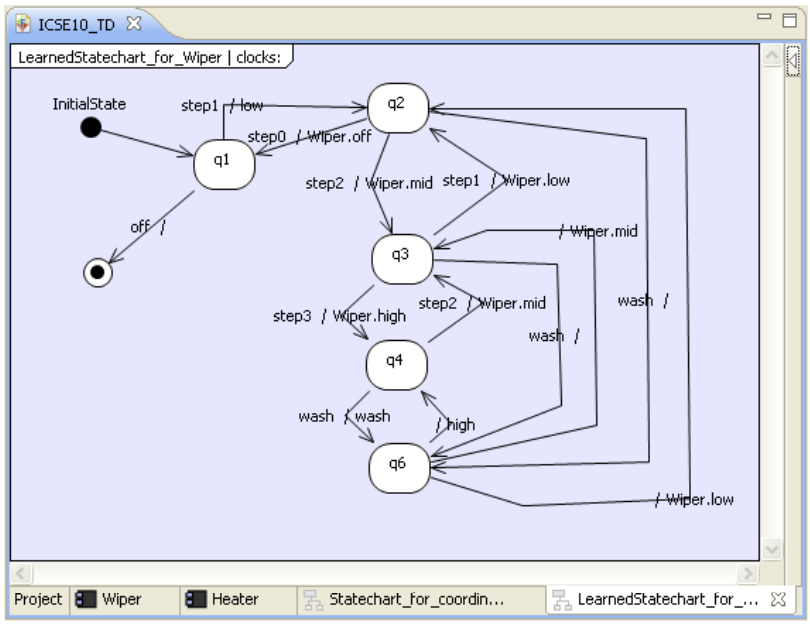

Figure 12: Learned behavior of the Wiper component

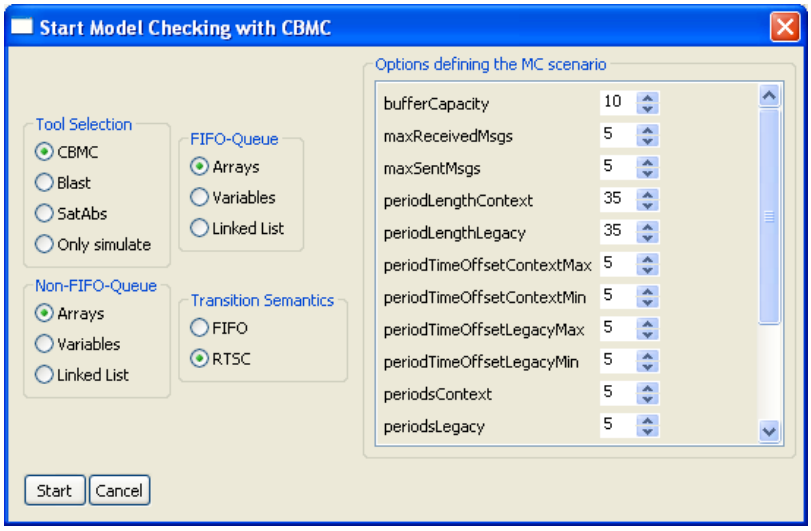

Figure 13: White-Box-Checking properties

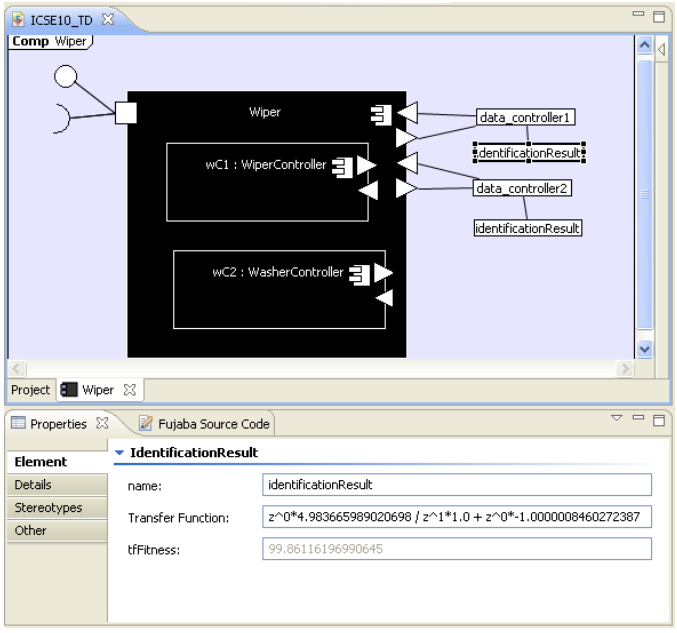

Figure 14: System identification of the Wiper component