# Integration of Legacy Components in MechatronicUML Architectures*

Christian Brenner, Stefan Henkler, Martin Hirsch, and Claudia Priesterjahn
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[cbr|shenkler|mahirsch|cpr]@uni-paderborn.de

Holger Giese
System Analysis and Modeling Group
Hasso Plattner Insitute
University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
D-14482 Potsdam, Germany
holger.giese@hpi.uni-potsdam.de

## ABSTRACT

One of the main benefits of the component-based development paradigm is its support for reuse which is guided by the interface description of the components. This facilitates the construction of complex functionality by the flexible composition of components. However, the also required verification of the resulting system often becomes intractable in practice as no abstract model of the reused components, which can serve the verification purpose, is available for the integrated legacy components. In this paper, we present the integration of legacy components in a MECHATRONIC UML model by an incremental synthesis of the communication behavior of the embedded legacy components combined with compositional verification.

## 1. INTRODUCTION

One of the main benefits of the component-based development paradigm is its support for reuse which is guided by the interface description of the components (cf. [10, 2]). In general, the proper composition of independent developed components in the software architecture of embedded real-time systems requires means for a sufficient verification of the integration step either by testing or formal verification. However, the overwhelming complexity of the interaction of distributed real-time components usually excludes that testing alone can provide the required coverage when integrating legacy components.

Thus formal verification techniques seem to be a valuable alternative. However, the required verification of the resulting system often becomes intractable as no abstract model of the reused components which can serve the verification purpose is available for legacy components.

A number of techniques which either use a black-box approach and automata learning [8] or a white-box approach which extracts the models from the code [9, 1, 7] exists. However, these approaches did not consider the specific context for efficiently synthesizing the relevant behavior of the legacy component, which is of paramount importance for embedded systems. Further, these approaches are not capable of finding conflicts in early learning steps.

*This work was partly 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.

In this paper we present a tool support for the incremental synthesis of communication behavior for embedded legacy components by combining compositional verification and model-based testing techniques based on [4, 6]. For the exploration of the component's behavior a formal model of the component's environment is applied. The environment model is employed to derive known environment behavior which is then used to systematically synthesize the relevant behavior of the legacy component as well as a formal model describing its communication behavior. While this formal model is not a valid encoding of all possible behavior of the legacy component, it is in fact a valid representation of its communication behavior for the context relevant for its embedding.

In the next section we present the incremental synthesis approach. Afterwards we describe the implementation as well as the evaluation. We finish the paper with the conclusion and future work.

## 2. INTEGRATION OF LEGACY COMPONENTS

Given a concrete context and a concrete component implementation with hidden internal details (legacy component), the basic question we want to check is whether a given property $\phi$ as well as deadlock freedom ($\neg\delta$) holds. We are in particular interested in a guarantee that both properties hold or a counterexample witnessing that they do not hold. However, usually the legacy component cannot be employed to traverse the whole state space as the state space of the complete system is too large to directly address this question. Before we answer the question, we discuss in the next section the prerequisites our approach. Afterward, we present an overview of our approach and discuss in more detail the relevant technique in Section 2.3.

### 2.1 Prerequisites

The approach presented in this paper will only work, if certain prerequisites and constraints can be fulfilled by the legacy component. The component must have neither nondeterministic nor pseudo-nondeterministic behavior. For example the firing of transitions must not directly depend on variable values or timing constraints since they are currently not explicitly captured. Such conditions will only be valid,

if they are encapsulated in the state information. All transitions must fire within a given timespan after the receipt of a triggering message. That is to prevent that a test run which leads to a deadlock will not terminate. This timespan also applies to $\epsilon$-transitions. In this case the timespan starts at the entry to the transition's start state. To enable the learning of the behavior the definitions of the component's interfaces muss be known, a start state must be given and it must be possible to reset to this start state. Further, the state changes (current state) must be observable at the interface.

Our experiences showed us that the prerequisites, besides the state information, are realistic for mechatronic systems, as this are reactive systems. As discussed in the future work we have to extend our black-box approach with additional white-box information, to abandon on the state information at the interface of the legacy component.

## 2.2 Sketch of the Proposed Approach

Given a MECHATRONIC UML architecture which embeds a legacy component and behavioral models for all other components building the context of the legacy component, the basic question of correct legacy component integration is whether for the composition of the legacy component and its context all anomalies such as deadlocks are excluded or all additionally required properties hold. However, it is usually very expensive and risky to reverse-engineer an abstract model of the legacy component to verify whether the integration will work.

To overcome this problem we suggest employing some learning strategy via testing to derive a series of more detailed abstract models for the legacy component. The specific feature of our approach will be that we exploit the present abstract model of the context to only test relevant parts of the legacy component behavior. The approach depends only to a minimal extent on reverse engineering results.

We start with synthesizing a model of the legacy component behavior based on known structural interface description. As shown in [4] we use a safe over approximation. Then, we check whether the context plus the model of legacy behavior exhibit any undesired behavior taking generic correctness criteria or additional required properties into account. If not, we use the resulting counterexample trace to test the legacy component. If the trace can be realized with the legacy component, a real error has been found. If not, we first enrich the trace with additional information using deterministic replay [3] and then merge the enriched trace into the model of the legacy component behavior. We repeat the checks until either a real error has been found or all relevant cases have been covered.

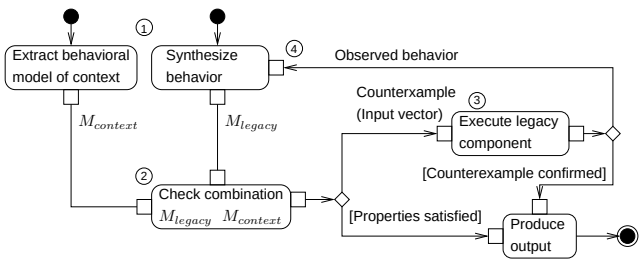


**Figure 1: Sketch of the approach**

Figure 1 illustrates our process with a summary of the overall approach. 1) Initially, we synthesize an initial behavior model for the legacy component based on known structural interface description and derive a behavioral model of the context from the existing MECHATRONIC UML models. 2) We check the combination of the two behavioral models and either get a) a counterexample or b) the checked properties are guaranteed. In the latter case we are done. 3) If we have a counterexample, we use this as test input for the legacy component. Deterministic replay enables us to enrich the observable behavior with state information by monitoring. If the tested faulty run is confirmed, we have found a real counterexample. If not, we can use the new observed behavior to refine the previously employed behavior model of the legacy component. We repeat steps 2) to 4) until one of the described exits occurs.

The approach can be extended to multiple legacy components, by using the parallel combination of multiple behavioral models. The iterative synthesis will then improve all these models in parallel. While theoretically possible, we can currently provide no experience whether such a parallel learning is beneficial and useful for multiple legacy components. Our expectation that it depends on the degree in which the known context restricts their interaction which determines which benefits our approach may show also for this more advanced integration problems.

## 2.3 Chaotic Closure

For our approach it is necessary that the model checker takes into account every behavior which is possible according to our current knowledge about the system. To accomplish this, the already known parts of the system are extended with chaotic behavior, resulting in a new model called *chaotic closure*. The latter is then, in combination with a model of the context, subject to model checking. Namely, 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 real system: It always models at least all of the system's behavior, but not all of the modeled behavior has to be possible in the system.

For modeling chaotic behavior a *chaotic automaton*, a non-deterministic finite automaton consisting of two states, can be used: The state $s_\delta$ with no outgoing transitions represents the case of the system 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 system are enabled and all outputs can occur. This is modeled by one self-transition and one transition to $s_\delta$ 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 system must be known. Both states of the chaotic automaton are initial states.

The chaotic closure is a combination of the synthesized model with the chaotic automaton for the system, mapping all unknown behavior to a chaotic one. Figure 2 shows as an example the Chaotic Closures (on the right side) for a trivial first conjectured behavior model and for a slightly more advanced one.

A Chaotic Closure is constructed as follows: First, the chaotic automaton for the input- and output-alphabets of

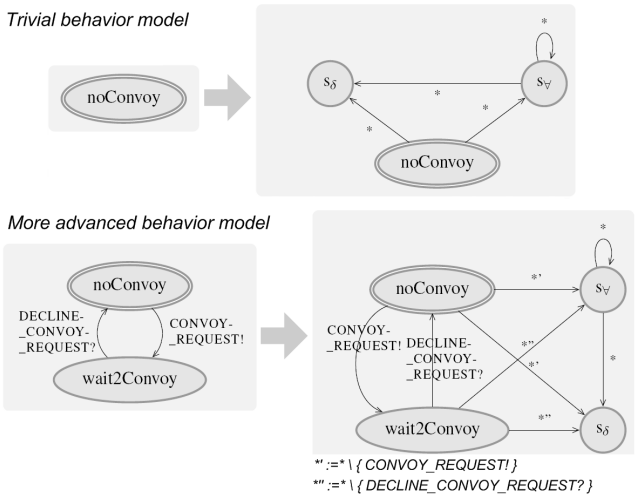


**Figure 2: Example for a chaotic closure**

the system 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 $s_\delta$ state. Contrary to the synthesized behavior, the chaotic closure constructed for it is non-deterministic.

The explicit deadlock state $s_\delta$ in the chaotic hull 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 system 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 system is integrated into, and therefore no time needs to be wasted with testing it.

# 3. IMPLEMENTATION AND EVALUATION

As the aim of our synthesis approach is to allow to safely integrate legacy components into existing Mechatronic UML contexts, it has been implemented in a way that it is on the one hand compatible to models created with the Fujaba Real-Time Tool Suite and on the other hand does not rely on a specific testing framework. The latter is important because it depends on the legacy component, which testing framework can be used. To accomplish this, the behavior synthesis step, being the core of the approach, was implemented in Java as a command-line tool. For the verification step, the model checker verifyta of the integrated tool environment UPPAAL has been chosen.

In the synthesis tool automata are saved as *Extended Hierarchical Timed Automata* (ExHTA), which have the same semantics as Fujaba Real-Time Statecharts (RTSCs), in a tool-independent XML-format. This has several advantages: As RTSCs can be exported into this format, it is possible to use a context which has been modeled in Fujaba using Mechatronic UML. Also the Uppaal Plugin provides a way to convert ExHTA to Timed Automata which can be validated using verifyta. It is possible to enable support for other model checkers as well, by implementing additional mappings to the formats they are using. Finally, the model synthesized by our approach also is saved in ExHTA. This should simplify implementing a method for loading it as a RTSC in Fujaba[1].

The first execution of the synthesis tool creates the first trivial behavior model, it constructs the corresponding chaotic closure by using the system's i/o-interface and it combines it with the model of the context. The subsequent execution of the (slightly modified) Uppaal Plugin converts this combination from ExHTA to the Uppaal XML-format. Then verifyta is used for model checking it against the properties defined in a certain CTL dialect. The resulting counterexample (if any) is then used on the one hand by a testing framework to execute it as a test case, on the other hand it is used by the synthesis tool for comparing it against the trace resulting from those tests. If trace and counterexample conform, a message will be issued by the tool. Otherwise the trace is used for learning new behavior (and so on).

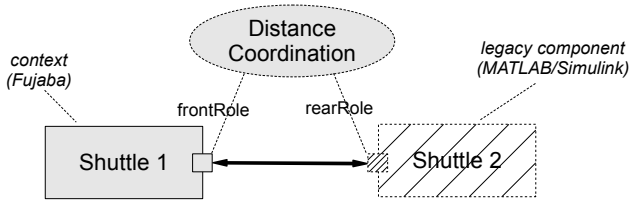


**Figure 3: Component diagram for the scenario used for evaluation of our approach**

Using the synthesis tool, our approach has been evaluated within a MATLAB simulation for the case of a MATLAB/Simulink legacy component being integrated into an existing Mechatronic UML system model. Figure 3 shows a component diagram picturing the scenario considered for this. The components in this diagram are two Rail Cab Shuttles, safety critical mechatronic systems, communicating with each other within a collaboration pattern. One of these Shuttles, the one that made the decisions, was the context in our approach. The other one, which was a reactive system, was the system to be integrated.

To realize the evaluation, a simple testing framework was implemented within MATLAB/Simulink. The main task of this framework is the execution of the counterexamples provided by the model checker as test cases: It sends a message to the system whenever the context would send one according to the counterexample and it logs all communication and all state changes of the system. Additionally, by calling the synthesis tool and the model checker it is able to execute our approach automatically for this scenario.

The evaluation showed that our approach is able to successfully synthesize a correct model for the given scenario. Also a simple error that had been added to the simulation could be found. However, it turned out that the synthesis needed quite a few steps, especially for learning the complete correct model. This was due to the model checker returning only very short counterexamples which lead to only one transition being learned in each iteration of the approach.

[1]An implementation for this is currently under development.

The total amount of testing necessary was greatly increased by this because to reach a new transition, often much of the already known behavior had to be tested again.

A way to force the model checker to create longer counterexamples is instrumenting the Chaotic Closure and modifying the temporal logic formula used by the model checker. These changes can be used together with certain command line options to make verifyta try to maximize the number of some transitions in the counterexamples. Three different modifications of this kind have been tried: One possibility is to increase the number of self-transitions of $s_\forall$ up to a maximum value. However this has the drawback that verifyta usually uses several iterations of one loop to achieve this. Another option is to make the model checker try to use every self-transition of $s_\forall$ at least once in each counterexample. Finally, the model checker can also be driven to try to let every counterexample contain every transition of the context's model. Each of these possibilities has been tested for our evaluation scenario, in several cases resulting in a much smaller amount of iterations and a smaller total amount of testing steps as well. However the success of each of these modifications is likely to depend heavily on the specific scenario they are used in.

In addition to the evaluation with MATLAB/Simulink we also consider using our approach with the IPANEMA framework. Using our synthesis approach within this framework would have the advantage of being able to use the model of the context on the one hand for exporting it to ExHTA and on the other hand for automatically generating code from it which can be compiled to run on the framework. A testing framework for model based testing [5] already exists which can be adapted to work in conjunction with the synthesis tool. Also a Test Case Generator exists which can be used for converting counterexamples to test cases.

# 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 and model based testing. It enables context specific learning with conflict detection in early learning steps. The employed learning strategy provides options for optimization as shown in the evaluation. The interplay between the formal verification and the test could be improved when a number of counterexamples instead only single one could be derived from the model checker. This is achieved by using specific strategies of the model checker to derive counterexamples.

Next, we want to combine the presented dynamic analysis with a static analysis to extend the applicability of the approach. E.g., the parts of the code which are responsible for the current state or internal variable values and dependencies could be detected by a static analysis and used by the dynamic analysis.

# 5. REFERENCES


[1] 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.

[2] I. Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[3] H. Giese and S. Henkler. Architecture-driven platform independent deterministic replay for distributed hard real-time systems. In *Proceedings of the 2nd International Workshop on The Role of Software Architecture for Testing and Analysis (ROSATEA2006)*, pages 28–38, New York, NY, USA, July 2006. ACM Press.

[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 248–272. SPRINGER, 2008.

[5] H. Giese, S. Henkler, M. Hirsch, and C. Priesterjahn. Model-based testing of mechatronic systems. In L. Geiger, H. Giese, and A. Zündorf, editors, *Proc. of the 5th International Fujaba Days 2007, Kassel, Germany*, pages 1–4, September 2007.

[6] S. Henkler and M. Hirsch. Compositional validation of distributed real time systems. In *OMER4 - Object-oriented Modeling of Embedded Real-Time Systems*, pages 1–6, 2007. accepted.

[7] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 597–607, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[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] D. Lucio, J. Kramer, and S. Uchitel. Model extraction based on context information. In *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems*, LNCS. Springer, 2006.

[10] C. Szyperski. Component Software and the Way Ahead. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, incollection 1, pages 1–20. Cambridge University Press, New York, NY, 2000.