Debugging Models in the Context of Automotive Software Development

Lars Stockmann
Software Engineering Group
Heinz Nixdorf Institute, University of Paderborn
Paderborn, Germany
lars.stockmann@hni.uni-paderborn.de

***Abstract*—Different models are involved in the automotive development process. In the integration phase, AUTOSAR is often the only model description available for a controller. The models that were used to generate the behavioral code (e.g., SIMULINK®) and sometimes even the source code itself are often not available due to IP protection. The controller software is verified using simulation-based testing, which may involve different kinds of environment models and legacy components. When a test fails, developers need to find the cause of the error. Even if the source code is available, source code debugging can be difficult, because the code has often been generated and thus may be confusing. Developers then use signal plotting for known variables, but plots can be ambiguous and lead to false conclusions. Therefore, exploiting the structural and semantic information of the involved models for debugging can be a valuable addition. However, the methods and tooling available are rather limited. Most of the existing approaches only consider live debugging. The intended PhD thesis aims at developing a methodology and tooling for debugging that makes better use of the models and the simulation data. It includes the application of techniques like slicing and the use of model hierarchies. A case study is planned in an industry context.**

***Index Terms*—AUTOSAR, automotive, debugging.**

I. Introduction to Automotive Software Development

Finding the cause for erroneous behavior in software is time-consuming. In 1990, Boris Beizer stated that developers typically spend half of their time testing and debugging [4]. Discussions at Programmers Stack Exchange[1] suggest that it still holds today. Although, there are no evident numbers in the automotive domain, I held interviews with developers working for vehicle manufacturers (Original Equipment Manufacturer, short OEM) and suppliers, which reveal that the effort even increases. This is because in recent years, more and more functionality has been introduced into cars. These functionalities range from engine controllers to decrease fuel consumption, over safety and advanced driver assistance systems (ADAS) to convenience and comfort features.

Traditionally, the OEM has bought these functionalities from suppliers as entire sub systems. However, the traditional one functionality per ECU paradigm is becoming obsolete. Many functionalities (i.e., software) are already distributed on multiple ECUs [22]. This can only be achieved through a separation of concerns. In the automotive software domain this has led to the development of AUTOSAR[2], which is a global standard [13].

The AUTOSAR methodology supports model-based development. This means that developers can use tools like dSPACE SystemDesk®[3], to model system architectures. Here, developers use diagrams and configuration dialogs. The tool then uses the supplied information to generate code. This is convenient and avoids coding errors, but also has the consequence that the developer is no longer familiar with the code involved. Furthermore, AUTOSAR has become huge, with about 20,000 pages of documentation. Its complexity has most recently been found to be the most mentioned drawback among developers [18].

II. Motivation and Problem Statement

The OEM has to ensure the correct and safe operation of the ECUs. Therefore, a plethora of tests is conducted. In a model-based development context, many of these are based on closed-loop simulation where the controller is connected to a simulation environment. It receives the controller's output and calculates the respective input forming a *feedback loop*. When a test fails, developers have to find out what went wrong.

A. Dealing with Errors Today

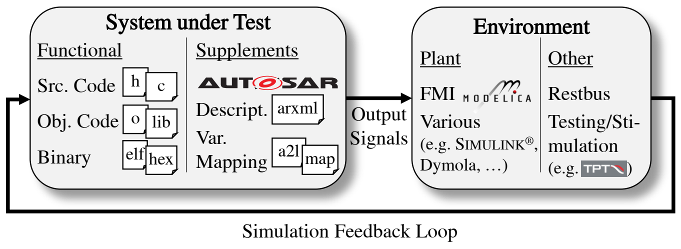
The typical process starts with a documentation of the apparent undesired behavior. This is called a problem/bug/defect or test incident report [12]. The first task of a developer is to reproduce the problem in a controlled environment. Afterwards, she starts tracking down the issue to its root cause, before it finally can be corrected. A common systematic approach is to observe the behavior of the system, induce a hypothesis about a possible cause and finally refute or verify it by testing predictions of the behavior. Zeller names this the scientific method [25]. The actual fix for a problem may range from a compilation with different settings to an additional requirement for a component that is provided by a supplier.

[1]Discussion on Programmers Stack Exchange: programmers.stackexchange.com/questions/93302/spending-too-much-time-debugging

[2]AUTOSAR website: www.autosar.org

[3]dSPACE SystemDesk® Product Information: www.dspace.com/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm

Figure 1. Simulation artifacts Overview



While at implementation level (unit level), where the behavioral code is devised, developers use their domain-specific language (DSL) tools (e.g., SIMULINK®[4] and dSPACE TargetLink®[5]) directly to test their hypothesis, in the integration phase, developers usually do not use the modeling environments (e.g., dSPACE SystemDesk®). Instead, they rely on independent tools for signal plotting and (source) code debugging. The former is an adaption of the traditional way to find electrical defects, i.e., measure analog signals. Developers also use it in tools like SIMULINK® and in the context of hardware-in-the-loop (HIL) tests.

Plotting allows the developer to reason from a whole range of signals. A weakness of this representation is that it can be ambiguous for periodically executing systems. For example, a continuous zero ("flat-line") could mean that something writes the actual value "0" one time and then never changes it, or it may write the value "0" multiple times, or it never writes anything, in which case the plot shows a default value "0".

The other method, which involves analyzing the runtime behavior of the code, is a well studied field and different techniques to track down the actual problem have been presented in the past decades. Probably the most used is the step-by-step execution and variable inspection, also called "live" or "interactive" debugging. One major drawback of this technique is that it relies on pausing the execution. As many tests are conducted in real-time and the erroneous behavior may be timing related, this may not help to reproduce the problem or it may simply not be possible. Furthermore, it requires a great knowledge of the code base to decide where to start. As stated above, developers in the integration phase might not have this knowledge, as a large portion of the code has been generated by tools out of models. Some code from a supplier might even not be available due to the protection of intellectual property.

B. Heterogeneous Simulation Systems

This brings us to another problem: Due to the distributed development process, software artifacts may come in different forms and flavors. This means that one cannot assume that all information required for debugging is available. Figure 1 shows an overview of relevant artifacts in a simulation. The controller software that forms the system under test (SUT) is depicted on the left. Note that it may actually be a composite of several software components and modules. It can be available as C code or – e.g., to protect intellectual property – as a platform dependent binary. It is accompanied by an AUTOSAR description, which contains the application software architecture together with the configuration of the ECU (communication behavior, scheduling ...). The behavioral SIMULINK® or dSPACE TargetLink® models that were used to generate the code are often not accessible in the integration phase. AUTOSAR itself does not provide means to model component behavior. Note that in the field of ADAS, there may not even be a model, but instead a lot of hand-written code involved. In the future, a language like MECHATRONICUML [3] could be a valuable alternative here and also for describing the interaction of the components on integration level. It follows the principle by Frank et al. that a "reflective operator" is needed, which monitors and governs the actual controller [8]. MECHATRONICUML also features compositional and timed model checking, which enables finding errors at design time.

The right side in Figure 1 shows the test environment. It is one major source for unexpected/undesired behavior. Artifacts here may originate from different models as well. They are either directly compiled and executed by the simulation technology or wrapped in an Functional Mock-up Interface (FMI) container (see [5]).

C. Problem Summary

To summarize the major problems, we may conclude that

1) the common methods to find the cause for errors, which are stepping through the source code and signal plotting, have shortcomings that cannot easily be addressed
2) automotive simulation systems are very heterogeneous and may not supply all behavioral information.

To tackle these problems, the proposed thesis aims at enhancing the debugging process by providing a methodology and tooling to isolate faults in simulation systems involving AUTOSAR software. This means that developers shall be able to get faster to the actual problem. The solution shall exploit the structural and semantic information of the participating models, which today is predominantly used solely for code generation and static analysis. Yet, it shall be based on the vast experience with source code debugging that has been gained over the last 40 years.

III. RELATED WORK

The idea to use models in the debugging process is not new. In 1997 Balzer concluded that there is a need to "recreate instrumentation, debugging, and monitoring capabilities we have long enjoyed at the program level" [2]. But concrete descriptions on how this can be achieved are scarce. Advanced techniques, such as algorithmic debugging and slicing are often not considered (see Zellers' excellent compendium on why programs fail [25]). Many approaches are based on live debugging.

One example is the tool suite by LieberLieber. In a recent development, they coupled[6] the AUTOSAR Engineer with

[4]SIMULINK® Product Information: mathworks.com/products/simulink/

[5]dSPACE TargetLink® Product Information: www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm

[6]Announcement of cooperation between LieberLieber and Lauterbach: http://www.lieberlieber.com/en/lieberlieber-cooperation-with-lauterbach/

the TRACE32®-In-Circuit-Debugger. It allows the user to set hardware break points in the model. I definitely share ideas with their approach. One is to use the AUTOSAR architecture diagram to present runtime data of the controller, which is retrieved using a debugger. However, as mentioned earlier, they do not consider other debugging techniques. Also, they only consider the debugging of one artifact. The developer still has to keep the whole system in mind and has to comprehend the relationship between different artifacts.

An earlier solution for live debugging is the UML Target Debugger by Willert Software [21]. It is comparable to the approach by LieberLieber in that the runtime information is also retrieved directly from the embedded hardware. It is then presented in their UML state and activity diagrams.

Another AUTOSAR related solution has been given by Elektrobit in 2009. Their tooling provides tracing functionality for AUTOSAR basic software [20]. They use a visualization based on plotting (e.g., for task active times). However, they do not offer a debugging solution for the application layer.

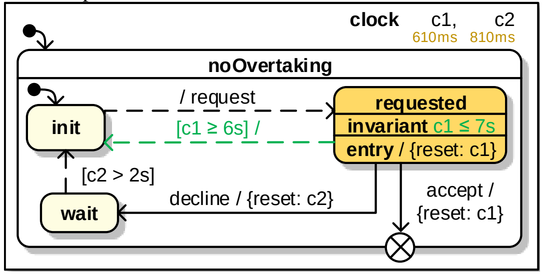
There are also solutions in the scientific literature. Haberl et al. show a promising approach that addresses some of the aforementioned problems. They map runtime data for inputs and outputs of real-time tasks, as well as internal states back to the corresponding data-flow model [11]. However, they require the system (including the units) to be modeled using their own component language (COLA), which supplies different layers of abstraction and instruments the generated code with meta-information. In our case, the model of the units may not be accessible.

Another interesting approach is AToMPM [17] that supplies a domain-specific modeling and debugging environment. It shows how step-by-step execution of arbitrary models can be realized. Here, the user specifies the runtime behavior of the models using a set of rules. An integrated simulator executes the model and the current state is visualized. Earlier approaches are based on UML, such as the UML simulator by Kirshin et al. that allows live debugging using a "Generic Model Execution Engine" [14]. Their environment features an animated diagram that reflects the current runtime state. The approach by Mayerhofer ("fUML model execution engine") [19] builds on top of that. In contrast to what we require, these approaches execute (i.e., interpret) the model directly. The full behavioral specification is available (as model). No code generation is involved. However, especially the work of Mannadiar et al. [17] tackles the foundations that I need in my work.

IV. Proposed Solution

As stated before, the goal is to make better use of AUTOSAR and the other different models in the debugging process. We do not have the full behavioral specification of every artifact. However, AUTOSAR compliance guarantees certain symbols to be exported/known to the integrator even if no source code is available. Thus, we have at least some structural and behavior related information, like component communication interface and runnable/task triggering. Furthermore, we assume to have the interface description of environment models, extensive simulation data and may be even some (possibly rough) knowledge about a components' behavior.

Figure 2. Example runtime visualization of a Real-Time State Chart (cf. [6])



A. General Methodology

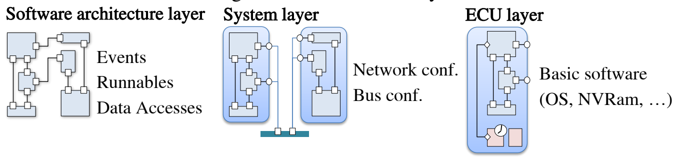
To use source code debugging techniques in our context, we require the following:

(1) a model runtime state mapping
(2) a value to model entity mapping
(3) opt.: a user interface providing a graphical representation of the runtime state, which suits the DSL

The *model runtime state mapping* (1) is the bridge between model runtime states and events to their respective code sequences. It provides entry points for break points or complex break conditions. It is somewhat similar to what is required to generate the code for a certain model entity. A break condition (e.g., a target value for a model entity) must be formulated using a proper constraint language. Referring to such values requires (2) *a value to model entity mapping*. It defines how the content of variables and runtime states of the program manifest in the model. As an example consider an AUTOSAR architecture and an event that is described by a condition like "The state of port $\mathcal{P}$ is changed". It maps to all functions in the AUTOSAR Run-Time Environment (RTE), which write a data element on port $\mathcal{P}$ (assuming sender/receiver communication). If data element $\mathcal{D}$ on port $\mathcal{P}$ is written, the respective RTE function is called. After the event has taken place, the actual value that has been handed over to the call is associated with respective data element $\mathcal{D}$.

A suitable graphical user interface (3) that visualizes the value to model entity mapping can improve usability, though technically a pure textual representation is sufficient. Note that special care must be taken for very big models. Further note that a properly profiled/extended UML model like the A Real-Time State Chart (RTSC) already supplies a suitable graphical syntax that could be used as a basis for the visualization. For RTSCs like those presented by Dziwok in [6] no solutions have been shown yet, but inspired by the approaches presented in the last section, it could look like Figure 2. Here, one can see that the state 'requested' is active (being highlighted) and the invariant 'c1 <= 7s' holds. The green highlighted transition to state 'init' is ready to fire, because the clock 'c1' is now greater than 6s.

Figure 3. AUTOSAR layers



But what can we do with components for which no model exists? To be able to include them in the process of model-level debugging, we need at least to model their interfaces. There is a scheme described by Giese et al. to embed black box components into MECHATRONICUML. They use a testing based learning strategy for reverse engineering [10]. This approach is probably not feasible for all participants in a simulation. However, instead of a learning mechanism, the developer herself (as oracle) could specify constraints or states as a result of what she expects from that component. Even if this does not make a complete specification, it may be enough to reason about an error.

B. *Advanced Techniques*

The basis for our methodology is that the developer should start debugging at the most abstract level available and then subsequently "zoom in", when the origin for an error on this level is identified. For example, one would start analyzing the runtime regarding the components' interaction (e.g., using MECHATRONICUML to see if no protocols are violated). Note that this is only possible if the respective debug symbols and descriptions are available. After that, one would advance to the AUTOSAR level, given that a mapping between those levels exists. AUTOSAR itself supplies a hierarchy (depicted in Figure 3). Unfortunately, each level requires its own mappings as described in the general methodology (sect. IV-A).

Using hierarchies in a live debugging context would enable a 'divide and conquer' like approach, which is one of Agans "golden" debugging rules [1]. They could also be useful for techniques other than the step-by-step method, even when those have been found impractical at the code level (e.g., *algorithmic debugging* [23]).

Another technique is to analyze the program and data flow w.r.t. dependencies and then use this information to synthesize a dependency graph. Thus, only the relevant parts of a program need to be analyzed. This technique is called *program slicing* and dates back to a PhD thesis by Weiser [24]. It complements the scientific method as an experienced developer will always reason about the relevant parts of the program, ignoring those that have no influence. Static program slicing can be enhanced with runtime information (i.e., execution history) that reduces the nodes of a dependency graph to those that really have been reached (dynamic program slicing as in [16]). This seems like a promising approach in our context. The execution history can be derived from the simulation data. Given a suitable query language, this would allow reasoning about event chains that happened in this simulation. In a way, this is comparable to formal verification, only that there is no state explosion problem, because all states are already known.

V. PRELIMINARY WORK

In the past I have analyzed automotive simulation systems regarding the involved artifacts. This has been done through interviews with developers of different OEMs and past projects. In these project we developed an AUTOSAR controller from the ground up [7]. Furthermore, I conducted an extensive literature study of the different debugging techniques that have been used during the last decades. This study was focused, but not limited to what is currently used in the automotive domain. An overview of the findings have been presented in the previous sections.

We also implemented a first prototype for debugging an AUTOSAR architecture. This prototype is able to present runtime data (e.g., values of data elements) and allows stepping. The data is retrieved using a source code debugger.

VI. EXPECTED CONTRIBUTIONS

This PhD thesis aims at delivering a methodology and tooling for model level debugging of automotive simulation systems. The focus lies on AUTOSAR, but also the other artifacts involved in a test will be considered. Therefore, the following questions shall be answered:

(1) How can we acquire the required simulation data with the least possible impact on the SUT?
(2) How can we effectively formulate conditions to describe the runtime state of a model, specifically AUTOSAR?
(3) How can we apply advanced techniques (e.g., program slicing, hierarchies)?
(4) How can we integrate knowledge/assumptions about component behavior (e.g., black box components)?

Regarding (1), note that until now, we assumed for simplicity that the required simulation data is somehow available. However, it is not trivial to get the whole history, which is required for slicing techniques and further analysis. In [9] it is stated that recording the programs input and output as well as the complete sequence of intermediate states is not feasible. It can be argued that this depends on the model. The more abstract a model is, the less data has to be recorded.

Regarding question (2) and (4), note further that a component behavior can be regarded as the sum of all its defined runtime states. Thus, those two questions are related to each other, in that a language that can describe a condition in a simulation system may also be suitable to describe (incomplete) component behavior.

VII. PLAN FOR EVALUATION AND VALIDATION

I plan to prototypically implement the different debugging techniques on the model level. With the help of an industrial partner, I would like to evaluate the methodology and tooling in the industry on real problems. Interviews or a query would then yield qualitative feedback from industry partners. An alternative (or addition) is a case study following the guidelines of Kitchenham et al. [15]. For this study, a working simulation system is artificially infected. Ideally, one would then have two groups finding the infection. One group would rely solely on source code debugging and signal plotting, while the other could additionally use our new approach. However, it is hard to find developers of the same skill level. Thus, such evaluation would be difficult.

VIII. Current Status

One current problem is the acquisition of relevant simulation data (1st question in the last section). It is required for full replay and slicing. The goal is minimal impact on the system under test. The usual methods involve instrumentation, which may modify the code in such a way that tests become meaningless. Here, several approaches are evaluated, one is the integration of a debugger, others involve bypassing strategies and instruction set simulation.

Another aspect is the formal specification of (incomplete) black box component behavior and (break) conditions (2nd and 5th question in the last section). Constraint languages (e.g., OCL, [T]CTL), but also the mentioned RTSCs and commercially available solutions are evaluated.

Furthermore, we have just started to transform MechatronicUML models to AUTOSAR, which has benefits beyond the scope of this PhD thesis. For my thesis it will be interesting to see if debugging beyond (AUTOSAR) model boundaries is feasible.

References

[1] D. J. Agans. *Debugging: the 9 indispensable rules for finding even the most elusive software and hardware problems*. American Management Assoc., Inc., New York, NY, USA, 2002.

[2] R. Balzer. Instrumenting, monitoring, debugging software architectures. Information Sciences Institute 4676 Admiralty Way Marina Del Rey, CA 90292 USA, 1997.

[3] S. Becker et al. The mechatronicuml method: model-driven software engineering of self-adaptive mechatronic systems. In *Companion proceeding of the 36th ICSE*. Hyderabad, India, 2014.

[4] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 2nd ed., 1990.

[5] T. Blochwitz et al. The functional mockup interface for tool independent exchange of simulation models. In *In proceedings of the 8th international modelica conference*, 2011.

[6] S. Dziwok et al. A tool suite for the model-driven software engineering of cyber-physical systems. In *Proc. of the FSE 2014*. Hong Kong, China, 2014.

[7] E. Farshizadeh et al. Design and Analysis of a Controller from System Design Idea to AUTOSAR Architecture with Basic Software Modules. In *6. tagung simulation und test für die automobilelektronik*. Stuttgart, Germany, May 2014.

[8] U. Frank et al. *Selbstoptimierende Systeme des Maschinenbaus - Definitionen und Konzepte*. J. Gausemeier, editor. Heinz Nixdorf Institut, Universität Paderborn, 2004.

[9] H. Giese et al. Architecture-driven platform independent deterministic replay for distributed hard real-time systems. In *Proc. of the issta 2006 workshop on role of software architecture for testing and analysis*. Portland, Maine, 2006.

[10] H. Giese et al. *Combining Formal Verification and Testing for Correct Legacy Component Integration in Mechatronic UML*. In. *Architecting dependable systems v*. Springer Verlag, 2008, pp. 248–272.

[11] W. Haberl et al. Model-level debugging of embedded real-time systems. In *Computer and information technology (cit), 2010 ieee 10th international conference on*, June 2010, pp. 1887–1894.

[12] Ieee standard for software test documentation. *Ieee std 829-1998*:1–64, Dec. 1998.

[13] F. Kirschke-Biller et al. AUTOSAR – a worldwide standard current developments, roll-out and outlook. In *5th vdi congress baden-baden spezial 2012*. Baden-Baden, Oct. 2011.

[14] A. Kirshin et al. A uml simulator based on a generic model execution engine. In *Proceedings of the MoDELS 2006*. Genoa, Italy, 2006.

[15] B. Kitchenham et al. Case studies for method and tool evaluation. *Ieee software*, 12(4):52–62, 1995.

[16] B. Korel et al. Dynamic slicing of computer programs. *Journal of systems and software*, 13(3):187–195, 1990.

[17] R. Mannadiar et al. Debugging in domain-specific modelling. English. In B. Malloy et al., editors, *Software language engineering*. Vol. 6563, in Lecture Notes in Computer Science, pp. 276–285. Springer Berlin Heidelberg, 2011.

[18] S. Martínez-Fernández et al. A survey on the benefits and drawbacks of AUTOSAR. In *Proc. of the first international workshop on automotive software architecture*. Montréal, QC, Canada, 2015.

[19] T. Mayerhofer. Testing and debugging uml models based on fuml. In *Proceedings of the ICSE 2012*. Zurich, Switzerland, 2012.

[20] J. Olig. Standardized debugging in AUTOSAR. In *6° automotive spin italia workshop*, 2009.

[21] E. Römer et al. Datenblatt - embedded uml target debugger™. W. S. T. GmbH, editor. 2012.

[22] O. Scheickl et al. Distributed development of automotive real-time systems based on function-triggered timing constraints. In *Proc. of the ERTS^2*, May 2010.

[23] E. Y. Shapiro. Algorithmic program debugging. PhD thesis. Yale University, 1982.

[24] M. D. Weiser. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. AAI8007856. PhD thesis. Ann Arbor, MI, USA, 1979.

[25] A. Zeller. *Why programs fail, second edition: a guide to systematic debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd ed., 2009.