Model-Level Debugging of Embedded Real-Time Systems

Wolfgang Haberl*, Markus Herrmannsdörrer†, Jan Birke‡ and Uwe Baumgarten§
Institut für Informatik, Technische Universität München
Garching b. München, Germany
Email: {haberl*, herrmana, birke‡, baumgar}@in.tum.de

Abstract—Model-driven development has become the state-of-the-art approach for designing embedded real-time systems. Due to their high level of abstraction, models are easier to understand and verify, thus leading to less faulty systems. But even when combined with automatic code generation, there is still the risk of unintended behavior. This may, for example, arise from real sensor inputs which differ from the characteristics assumed in the model. Consequently, debugging techniques still play an important role, even in model-driven development processes. However, debugging a system on the embedded target platform is tedious because of the limited user interface. In this paper, we present an approach for capturing runtime data on the target platform and mapping them back to the model. Debugging can then be performed at model-level by visualizing actual input data, like feedback from the target platform’s environment. Using a case study, we demonstrate a realization of our approach.

I. INTRODUCTION

Today, more than 90% of all processors are part of embedded systems [1], [2]. These are integrated in laundry machines, medical systems, cars, and aircrafts, just to name a few. In this paper, we focus on distributed embedded systems, consisting of a multitude of computing nodes, interconnected by various bus systems. Such systems contain or constitute life-critical electronic resources. Faults of any kind thus may be fatal. Even if not fatal, they bare large warranty costs for the designers and integrators of the embedded system.

Model-driven development (MDD) is a well established means to tackle the complexity involved in designing such distributed embedded systems. The COMponent LAnguage (COLA) [3] was built to tackle the complexity of large scale systems. COLA has a concise syntax, which caters for easy understandability, and is defined by a rigorous formal semantics. Consequently, formal verification can be applied—where possible in reasonable time—to guarantee conformance of the models to the requirements, and automatic generation can be used to synthesize code from the models. These characteristics make COLA well-suited for development of mission-critical real-time systems.

The complexity of a designed real-time system not only necessitates proper abstractions, but also calls for an automatic transformation of the model into executable code. The tools built around COLA not only enable the generation of code representing the modeled functionality, but also accomplish the configuration of the distributed target platform. This automated translation reduces the number of faults and guarantees reproducible results, thus improving overall quality.

Problem. But even such a generated system may produce failures, due to faults contained in the system specification or because of limited knowledge about the exact environmental conditions. Consequently, faults become evident not until executing the system on the target platform, and are hard to debug regarding its embedded nature. To get a grip on such faults calls for a technically mature debugging concept, even in a MDD process. While code generation for data-flow models is well known [4], we are not aware of any approach facilitating the debugging of a generated system therefrom at model-level. Regarding safety requirements for generated code, as well as its readability, we consider this form of debugging a must.

Contribution. In this paper, we present an approach for debugging a system at model-level. The presented approach allows to capture actual runtime data for inputs and outputs of real-time tasks executed on the target platform, as well as their internal state. This information can then be mapped back to the corresponding data-flow model. Thus debugging can be carried out at model-level, avoiding any unintended influence on the generated application code by manual patching. To limit the amount of data that has to be captured during execution, it is possible to specify the software components which shall be debugged in the system model. Our approach is facilitated by the fact that COLA features an integrated development concept—including modeling of software and hardware as well as tools for code generation and a matching execution platform scheme—which other development tools do not offer yet in this seamlessness.

Outline. We motivate and outline our approach for model-level debugging in Section II. In Section III, we give a brief overview of the COLA language concepts on which the approach is based. We describe the realization of our approach—including code generation, data recording and simulation—in Section IV. In Section V, we present our experiences while using model-level debugging in a case study. We provide an overview of related work in Section VI, before we conclude in Section VII.
II. The Approach in a Nutshell

The Challenge. In current MDD practice, systems are designed by means of models. Simulation is used to validate these models against the functional specification. If no more errors show up during simulation, code is generated from the models which is then executed on the target platform. In our experience, however, it often happens that the code does not behave as intended.

One reason is that there might be design faults in the model, which are not found by simulation because of missing or incomplete environment feedback. For example, if an adaptive cruise control system adjusts the current throttle position to increase the speed of a vehicle, the wheel rotation sensors will subsequently deliver higher values due to the increased velocity. But these values are also influenced by the weight of the vehicle, the current wind speed, etc. Including all these parameters in simulation demands a complex environment model, which is time consuming and difficult—if in some cases not impossible—to construct. Therefore, the initial design is usually an approximation of the final system, and only tests on the target platform allow to verify the correct functioning of the design.

Another source for unmeant system behavior may be sensors and actuators. A sensor will eventually deliver values different from assumptions made in the model, due to digitization errors. Unexpected, intermittent jumps in the delivered values are a prominent example of this problem. Similarly, actuators might react differently or less accurate than anticipated to rapidly changing real-world values or ambient conditions.

Classical Debugging. As a consequence, today, developers start to debug the generated code. However, debugging at platform-level is tedious due to several reasons.

In embedded systems development, there are very limited possibilities for communicating debug information. Textual error messages—as oftentimes used with desktop systems—typically cannot be employed due to the lack of a display. Thus, debug messages are signaled by a blinking LED or other low-level communication mechanisms, making debug-
system model, as well as the corresponding values captured at platform-level, can be loaded in the simulator. These real-world values are then used for simulation. Compared to simulation based on fictitious values, a simulation based on data acquired at platform-level exhibits realistic system behavior. Hence, realistic debugging at model-level is made possible.

This approach solves the aforementioned difficulties. Thanks to the import of captured data into the simulator, a convenient display of runtime data is achieved. All values are shown next to the corresponding model elements, allowing quick and easy identification of design faults as well as their elimination. Changes are completely performed at model-level, thus not affecting code integrity. Finally, the acquired input values can be reused for future simulations of new designs, without the need of a hardware platform available from the very beginning.

III. OVERVIEW OF COLA

COLA—the COmponent LAnguage—is a modeling language for the design of distributed, safety-critical real-time systems. COLA provides constructs to model a system along different layers of abstraction, helping to reduce the complexity involved in the design of large-scale embedded systems. Each abstraction layer focuses on a certain aspect and adds new detail to the information defined on the higher layers.

COLA defines three abstraction layers. The feature architecture formalizes the requirements by means of so-called features which define the behavior observable at the system boundary [9]. The logical architecture maps the features onto a software architecture, and refines them to obtain a platform-independent model of the overall system functionality. The technical architecture partitions the modeled functionality into distributable entities, and maps them onto the target hardware platform. In this paper, we focus on the two lower abstraction layers and the transition between them. Figure 2 illustrates the core modeling constructs of these two layers and their relations by means of screenshots from the COLA editor.

Like Lustre, COLA is based on the synchronous data-flow paradigm and has a well-defined semantics [3]. It is assumed that operations start at the same instant of time and are performed simultaneously with respect to data dependencies. The computation of the system over time can be subdivided into discrete steps, called ticks, and the execution is performed in a stepwise manner over the discrete uniform time-base.

Logical Architecture. The logical architecture defines the functional behavior of a system. Therefore, the system can be successively decomposed into or composed of modular units. The behavior of a unit is hidden behind the unit’s interface which is defined by a number of typed input and output ports. Figure 2 depicts COLA’s graphical representation of a number of units: interfaces are represented as rounded rectangles, and ports as triangles. The root unit of a system does not have any unconnected input or output ports, but communicates with the environment using so-called sources and sinks, representing sensor and actuator interaction. COLA defines three kinds of units: networks, automata, and blocks.

A system can be decomposed into smaller units, establishing data-flow networks. These units are called sub-units of the network, and are connected by so-called channels. A channel connects an output port with one or more suitably typed input ports [10]. Due to COLA’s synchronous semantics, communication over channels takes no time. The channels thus determine the data-flow dependencies between the sub-units, and induce a causal order of execution. Feedback circles of channels connecting an output port of a unit to an input port of the same unit have to contain a delay block which defers propagation by one time interval. A delay thus serves as a data storage, as it retains values from one tick to the next. Figure 2 depicts COLA’s graphical representation of a network: sub-units are represented by rounded rectangles, and channels by lines connecting the sub-units.

A system can be decomposed according to its control-flow by means of an automaton. An automaton consists of a number of states of which exactly one is activated at a time. The behavior of the automaton is defined by the active state. Each state is again implemented by a sub-unit. Transitions determine how the activated state of an automaton may change over time. The transition guard is again implemented by a unit, whose evaluation is based on the inputs of the automaton. Our semantics requires to check for possible transitions before evaluating the activated state’s behavior. Figure 2 depicts COLA’s graphical representation of an automaton: states are represented by ellipses, and transitions by arrows.

To finish decomposition, COLA provides a number of basic building blocks like e.g. arithmetic or boolean operators. These blocks execute their respective operation based on the values present at the input ports and emit the according result at their output port.

In the course of computation, a unit may act differently depending on its history. Such units are considered stateful. In COLA, only delays and automata retain information of previous computations, whereas all other kinds of units are stateless. Note that a unit containing a stateful element becomes stateful as well.

Technical Architecture. The technical architecture maps the system functionality as specified by the logical architecture onto a hardware platform. As is depicted in Figure 2, the technical architecture consists of hardware architecture, cluster architecture, and allocation.
The hardware architecture describes the structure and the properties of the target hardware platform. The main construct of the hardware architecture is the electronic control unit (ECU) which forms a computing node of the distributed platform and can be composed of processors, sensors, and actuators. Sensors and actuators provide access to the environment, whereas processors execute the system functionality. ECUs can be connected via a bus which provides a communication mechanism. The characteristics of each hardware element—like e.g. the resources provided by a processor—can be determined by a number of properties, for example speed, storage capacity, etc. Figure 2 depicts COLA’s graphical representation of a hardware architecture.

The cluster architecture partitions the system functionality, defined in the logical architecture, into distributable entities. A distributable entity is called a cluster and encompasses part of the system functionality. The clusters are intended to be distributed on the ECUs of the hardware architecture. The cluster architecture can be specified manually by the developer, or can be derived automatically from the logical architecture based on the characteristics—for example available processing speed, memory, etc.—specified in the hardware architecture. In the latter case, the derivation of the cluster architecture can be performed based on some optimization criterion.

The allocation establishes the relationship between the cluster architecture and the hardware architecture. The allocation therefore maps each cluster onto an ECU. It must accommodate the resources provided by the ECUs—like computing power, memory, etc.—with the resource requirements of the clusters. To estimate the resource requirements of a cluster, its worst-case execution time (WCET) needs to be calculated. A suitable allocation can then be generated automatically based on both, the hardware and the cluster architecture.

From the technical architecture, we can automatically generate code which can be deployed onto a platform as specified by the hardware architecture. The generated code interfaces with our middleware which provides an abstraction from the real hardware distribution.

IV. REALIZATION

Figure 3 illustrates the different steps of our realization of model-level debugging.

Starting with the COLA model, an executable system is derived by means of code generation. The generated functional code includes meta-information about its relation to model elements, as we will describe below. During system execution, the actual input and output data of the clusters which shall be debugged are captured. We refer to this activity as tracing. The gathered data are in binary form, and thus data extraction has to take place, before the data can be mapped back to model elements. The resulting trace is the model-level representation of the data captured during system execution. Together with the original model, this
trace is used to allow for a realistic simulation at model-level. We will detail each of these steps in the following.

**Code Generation.** To retain the quality of the modeled system down to a concrete platform, the tool chain for COLA includes tools for automatic generation of C code. We presented the basic mode of operation of the generator tools in previous papers [5], [6]. So we will focus on the relevant points necessary for our tracing approach here.

Besides the application code generated for the modeled application, our code generators produce configuration files for the middleware instance on each ECU of the system. At system startup the middleware reads information about the clusters and their ports deployed to the respective ECU from this configuration file. If a cluster is enabled for tracing in the model editor, it is marked accordingly in the configuration file. The middleware then records a trace for this cluster during execution.

While mapping data captured for ports back to the model is easy using their unique middleware address, the decoding of the clusters’ internal states, which are composed of automata states and delays, is more complex. The whole state of a cluster is stored as a struct using a single middleware address. For each sub-unit of a network or state of an automaton, the cluster’s struct comprehends a nested struct for keeping the sub-state of the contained model element. To facilitate a mapping of struct members to model elements, the C code is annotated during code generation using unique identifiers taken from the modeling framework.

The middle part of Figure 4 shows an excerpt from such a struct definition for a cluster named atm_emergency_stop. The top-level unit of the cluster is an automaton. The actual state of the automaton is stored at runtime using an integer variable named atm_state. The automaton’s states are modeled as sub-units named state_parking, state_sdc_active and state_normal. Each of the sub-units’ states is stored in a nested struct. To facilitate a mapping of data back to units in the model, the states’ unique identifiers used by the modeling framework are inserted as a comment, next to each variable definition. In the example, these identifiers can be found at each end of line.

**System Execution.** The distribution of clusters—tasks at runtime—over the different ECUs of the platform is irrelevant during code generation. Rather all ports at the cluster boundary, which imply communication during execution, are translated into read and write calls to our middleware. The middleware handles inter-cluster communication as well as access to the underlying hardware transparently, using numerical identifiers for addressing. In addition, the tasks use the middleware for loading and storing their internal state, at the beginning and end of their execution, respectively. Thus our middleware is able to serve at runtime—besides its primary duty as a mediator for communication—as a monitor for exchanged information and the tasks’ internal states.

If tracing is enabled for a cluster, its communication is logged to a file. As control systems are executed cyclically, this file is expanded during each invocation of the corresponding task. Since memory in embedded devices is limited and should not be exhausted, the storage format for the trace has to be as compact as possible. Also, no complicated transformations on the data should be carried out to save processing resources. So the proposed storage format is very simple and contains the minimum of necessary information. This includes a header specifying the traced cluster together with its ports and the corresponding addresses, as well as the actual data in binary format. Additionally, the number of previous task activations is stored at the beginning of each invocation.

**Data Extraction.** For decoding the trace file, it is important to know about the structure of the file. Otherwise, the binary data cannot be transformed into the original values. Therefore, a header is written at the beginning of the file. The header is in character representation and defines the values which are stored in the binary part of the file. For each traced cluster a TASK definition specifies the types used for the state values. After the TASK definition, the DATA definitions for the ports of the cluster follow, defining port names and data types. If more than one cluster is traced, there may be repeated TASK and DATA definitions. The traced data are saved to the trace file in the same binary representation as they are stored in the memory of the system. Thus, architecture specific properties such as endianness and sizes of primitive data types have to be considered during data extraction. Together with the complex data types specified in the header, it is possible to partition the binary trace into its particular values.

For mapping port values from the trace to elements in the model, the corresponding ports can be found via cluster and port names specified in the header. Mapping state values contained in a struct to model elements is more difficult,
since the corresponding elements are distributed along different refinement layers in the model. The values for all these elements are combined in one binary chunk extracted from the trace, namely the state of the cluster. In order to partition the state struct into its members, the application’s source code is considered. Using the struct definition, the binary data can be divided into struct elements. To identify the model unit associated with a struct element, the numerical identifier of the unit is looked up in the annotated C code. In Figure 4, these identifiers can be seen in the form of comments in the C code. Figure 4 also indicates the mapping of actual runtime data in the system to model elements, using the identifiers.

As can be seen in Figure 4, nested structs are annotated with one identifier, while the integer variable `atm_state` is annotated with a list of identifiers separated by semicolons. This list is necessary because the `atm_state` holds the active state of the automaton, which is encoded using an integer value in the C code. Each token of the list contains the code representation of an automaton state and the unique identifier of the corresponding automaton state in the model, separated by a comma. Using this information, the integer value found in the trace can be mapped to an automaton state in the model.

After data extraction is finished, the recorded trace can be loaded into the simulator to retrace the execution at model-level. The meta-model of a trace is clearly defined and is part of the integrated modeling language COLA, which has been presented in Section III. Conceptually, the trace consists of a sequence of execution steps. For each execution step, it logs enough information to be able to repeat the execution in the simulator. First, the valuations of the ports at the cluster’s boundary are stored. This also includes sources and sinks that are contained in the cluster. Whereas the valuations of the input ports are required to retrace the execution, the valuations of the output ports are used to validate the executed code against the model-level simulator. Second, the states of all stateful units that are part of the cluster are saved. Whereas the state during the first step is required to initialize the execution in the simulator, the states of the following steps are also used for validation.

The concrete syntax defined by our modeling language represents traces by means of two-dimensional tables. Figure 5 depicts an example of a trace in its concrete syntax. The table has a row for each execution step, and a column for each port at the cluster’s boundary as well as for each stateful unit contained in the cluster.

**Simulation.** We extended the existing simulator [8] with the functionality to execute a trace recorded on the hardware platform. The simulator then executes the units which are encompassed by the traced cluster. Due to the modular architecture of our simulator, the realization of this functionality could be integrated in a straightforward manner. The architecture of the simulator is modularized into three components: the runtime configuration, the environment interface and the execution strategy.

The runtime configuration describes the state of a cluster during runtime. That means, it decorates the clustered units with information required at runtime, i.e. the valuations of ports and the internal states of stateful units. The initial runtime model can be automatically derived from the structure of the clustered units. The environment interface specifies the behavior of the environment surrounding the cluster.
Therefore, it controls the inputs entering and observes the outputs exiting the ports at the cluster’s boundary. Based on the environment interface provided by the simulator, different kinds of environments can be realized. The execution of a trace is realized as an environment interface that provides the simulator with the information from the trace. The execution strategy performs a stepwise execution of the system. This is done by modifying the runtime configuration depending on the inputs and generating the outputs at the cluster’s boundary. The execution strategy is determined by the semantics of our modeling language.

Figure 5 shows the user interface of our simulator during the execution of a trace. The trace was recorded while testing the functionality of our case study on the hardware platform. A control panel allows the user to start, pause and stop simulation as well as to regulate its speed. The user interface provides views for both the structure and the visualization of the runtime configuration in concrete syntax. An optional environment visualization shows the movement of the car in relation to a wall.

V. EXPERIENCES

To prove the viability of our approach, we implemented a case study using the concepts and tools described before. The idea was to build a model car, featuring a function also available in real cars. We decided to implement an autonomous parking system based on several distance sensors. Additionally, the system should be controllable manually via a Bluetooth connection to a cell phone, and it should initiate an emergency stop when reaching a given minimum distance to obstacles. The COLA implementation of this functionality consists of 616 units organized in eleven clusters.

The model car which is depicted in Figure 6 was equipped with three Gumstix® electronic control units (ECU) connected by an Ethernet network. Our middleware was employed on top of the network for data exchange and clock synchronization services. Xenomai served as operating system for the ECUs. Distances were measured using two infrared and one supersonic sensor. Bluetooth was used as another input, connected to the cellular phone remote. The model car’s motor and steering, indicator, reversing, and breaking lights were the actuators of the system. The mentioned sensors and actuators were connected to different ECUs, thus posing the need for synchronous communication in the system.

During development of the described system, the tracing of actual data proved to be very helpful. An early version of the modeled system did not behave as expected. Using model-level debugging, we were able to identify several faults in the model. One case, for example, was related to some flipped channels in the model, thus forwarding data of the employed sensors to the wrong inputs. Additionally, the generated values of the used infrared and supersonic sensors were not as constant as assumed in the model. Rather, there where jumps in the measured distance values, even though the actual distance did not change. By tracking actual data in the simulator, we were able to identify the problem and add filters for these input values to the system model. The corresponding parking cluster contained 115 units and was traced for 147 invocations, corresponding to 14.7 seconds. The generated trace file was 45 KByte in size. So the tracing of real-time data showed to be a valuable addition, while producing moderate memory consumption.

VI. RELATED WORK

Similar to Lustre [11], which could be seen as a superset regarding language constructs, COLA uses synchronous data-flow as its model of computation. In contrast to Lustre, COLA provides constructs to model a system along different layers of abstraction, which have been introduced in Section III. These layers further reduce the complexity involved in the design of large scale embedded systems and comprise information about the system not only at a functional level, but from requirements definition down to a concrete hardware platform specification. The abstraction layers are tightly integrated and thus enable a seamless model-driven development process. Without this seamless integration, an automatic deployment of distributed systems including the model-level debugging approach described here would not be possible.

Considering model-driven development, there are commonly used commercial tools like ASCET-SD® and MATLAB/Simulink® for the design of embedded systems. These tools also feature automatic code generation as described by Putty et al. [12], [13], and even simulation at model-level. Still they lack a way for importing actual data from a real test system. Rather, generated code can be executed on a virtual target [14], which does not necessarily behave like a real target platform. Thus, the simulation results may not be as meaningful as the ones used in our approach.

Another commercial solution is Rose RT® from IBM, which features the demanded mapping of actual system behavior back to the system model. However, this solution relies on a network connection between target and host computer for data transfer. This requirement might not be
satisfiable for many embedded targets, demanding for a modified development target platform. Moreover, the debugging server needed on the target has comparatively high impact on the computing resources available.

At platform-level, there is extensive tool support for debugging. A method to replay traces recorded on real-time systems has been presented by Thane et al. [15]. Similar to our approach, data stored during an actual execution of the target system are used for later debugging. In contrast, the debugging is carried out at platform-level.

Another concept aiming at platform-level was presented by Park et al. [16]. Their approach is based on the well-established GNU Debugger (GDB), which allows the use of breakpoints and variable manipulations at runtime, using the program code as basis. Thus, debugging again is not applicable for graphical models. In addition, a resource consuming debugging server running on the target is necessary, comparable to Rose RT.

While the mentioned tools and concepts are well established and have proven their benefit for system development, to the best of our knowledge, none of them implements a mapping of runtime data from the platform-level to a system design at model-level.

VII. CONCLUSIONS

In this paper, we presented an approach for debugging embedded systems at model-level using data captured during execution on an actual target platform. This approach closes the gap between system design and system debugging, which arises in model-driven development. The avoidance of manual changes on the code-level allows for the generation of code which complies to security standards, assuming the use of a certified code generator.

We believe that our approach is applicable to target platforms different from our demonstrator. While this would require the middleware to be adapted to the new target and the characteristics of the employed compiler would have to be considered during data extraction, the basic concept is still applicable.

An interesting research direction for future work is generation of environment models from traces. By using a huge set of traces, it might be possible to derive dependencies between output and input, for example throttle position and measured speed. If these dependencies could be generalized for a specific type of system, they would allow simulation and debugging of new models for similar systems in a virtual environment.

REFERENCES


