Verification of embedded control systems by simulation and program execution control

Stefan Resmerita and Wolfgang Pree

Abstract— **This paper describes features and applications of a simulation framework for software-in-the-loop testing of real-time embedded control applications. The Validator tool performs time-functional simulation of control software and its execution platform in closed-loop with Simulink plant models. It is based on a discrete event simulator which executes the application software on a host platform and simulates the passage of time according to execution times of application code and communication times pertaining to a given embedded target. The Validator also simulates functional behavior of operating system services and hardware components at a level of abstraction that enables capturing significant timing aspects without executing detailed hardware models. We present use cases where the Validator is employed for evaluating integration of new control functions in existing systems, model refinement, and regression testing of automotive control software.**

I. Introduction

State-of-the-art validation of real-time properties of embedded software is performed by extensive testing involving hardware in the loop. Simulation is mainly used for testing functional properties of applications represented as software or as higher level executable models. The costs of hardware in the loop testing, plus the increased complexity of distributed embedded applications make the case for shifting the main load of real-time testing towards software in the loop setups. Clearly, to simulate the real-time behavior of an embedded application, one needs to simulate also the functionality and timing of the execution platform (hardware and operating system), sensors, actuators, and the physical plant under control. An important challenge in this case is finding the right level of abstraction, which determines the modeling effort, the properties that can be tested as well as the efficiency of the simulation.

This paper describes how the Validator tool suite [1] is employed for simulation-based verification of timing-related properties of control software. The Validator is a software-in-the-loop simulator, where the application code is executed on a host computer and the passage of time is simulated according to execution and communication times related to a given execution platform. This timing information can be obtained by static techniques involving program analysis or by methods based on measurements. The Validator also simulates functional behavior of operating system services and hardware components at a level of abstraction that enables capturing significant timing aspects without executing detailed hardware models. The tool achieves co-simulation with Simulink plant models based on time synchronization protocols.

In this paper, we present aspects of the underlying simulation technology of the Validator, with particular focus on the implementation of the state store/restore feature. Then we describe illustrative use cases for the Validator, centered on verification of real time properties of automotive control systems.

Stefan Resmerita and Wolfgang Pree are with the Software & Systems Research Center, University of Salburg, Jakob-Haringer-Str. 2, A-5020 Salzburg, Austria. E-mail: firstname.lastname@cs.uni-salzburg.at.

II. The Validator Tool Suite

A. Simulation Technology

The top level components of a Validator model are actors in the Ptolemy II sense [2]. Ptolemy II is a software framework for modeling, simulation, and design of concurrent, real-time, embedded systems. In Ptolemy II, the behavior of an actor is defined by implementing a programming interface, which can be used by a simulation director to execute the actor according to specific rules.

During simulation, an actor experiences a number of iterations, where an iteration generally consists of three successive actions: *prefire*, *fire* and *postfire*. Each action is represented by a method in the actor interface. The main functionality of an actor is encoded in the fire method, where the actor reads inputs and produces outputs. The actor interface also contains methods for initialization of the model, called *preinitialize* and *initialize*, which are called before the first iteration of the simulation, as well as a finalize method *wrapup*, executed after the last iteration.

The Validator is implemented in C according to the actor-based design principles, and employing a discrete-event simulation director. The main components of a Validator model are depicted in Fig. 1 and described next. Application actors represent software functions mapped to platform tasks and interrupt service routines (ISRs). Operating system actors perform the functions of an OSEK operating system. Hardware actors model functionality and timing of common hardware parts such as interrupt controllers, timers, bus controllers, hardware sensors and hardware actuators. The plant actor represents the physical system under control, including sensors and actuators. The actor implements a protocol for handshake time synchronization with Simulink. For continuous-time Simulink models, this has the effect of zero-order holders for the interface signals.

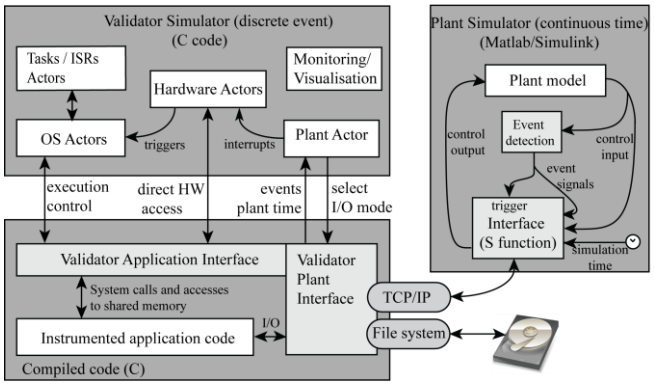

Figure 1. Architecture of the Validator model

To simulate a software application mapped to a target platform, a model with the above structure needs to be built (only models of uni-processor systems are considered here). The modeling process involves the following steps: (1) Execution time analysis of the application code, which is done with existing program analysis tools such as AbsInt's aiT tool [3]. (2) Instrumentation of the code with execution time information. (3) Instrumentation with execution control statements, to enable the simulator engine to control execution of the application code (e.g., to switch execution between application tasks). (4) Generation of the task and ISR actors from the system configuration files (e.g., OIL files in the case of OSEK). (5) Generation of the Validator interface code. These steps are mostly automated by a tool set that achieves a fast modeling process based on information about the hardware/software architecture.

B. Enhanced actor interface for state store/restore

A Validator simulation can be started from a previously saved state. To support this feature, the actor execution interface was enhanced with two more methods: *suspend* and *resume*. The *suspend* method must be executed before saving the program state, as well as at the end of the simulation before the wrapup method. The *resume* method must be executed once after the *initialize* method as well as upon every resumption of execution from a saved program state. The *suspend* method should bring the actor to a state which can be stored and then restored by the *resume* method.

For example, consider an actor with the following behavior: at the beginning of a simulation, the actor should allocate memory for a buffer and open an empty file for writing. At each iteration, the actor stores its inputs in the buffer. When the buffer is full, the actor writes its contents to the file. The file is closed at the end of the simulation. With the classical Ptolemy actor interface, memory allocation and file creation is done in the initialize method. Also, memory de-allocation and closing the file is done in the wrapup method. It would be difficult to save/restore the state of each actor and of the simulation engine using only these methods. In the above example this would require that the initialize method should test if the simulation starts from time zero (then it should create an empty file, which would delete any previous instance of the file), or if the simulation is being resumed from a previously saved state (then it should open the file for appending). Moreover, explicit functions should be added to the discrete event simulation engine to save/restore its state (e.g., its event queue). Using *suspend*/*resume* methods in the actor interface greatly simplifies this task, enabling methods which require no explicit code to save the simulator state. In the above example, the initialize method allocates memory, opens the file for (over)writing and then closes the file. The *resume* method opens the file for appending. The *suspend* method closes the file and the wrapup method de-allocates the memory.

This model provides a clear separation between the states that must be initialized only once (e.g., memory allocation) and the states that must be re-initialized at every resume point. The Validator employs two different mechanisms to store/restore the simulation state:

1) Checkpointing the entire simulation program. The Validator employs the *dmtcp* checkpointer [4] to save the entire program state of the simulator. The checkpointer can later load the saved state and resume the program execution. To save the state and exit, the Validator calls a checkpointer function. When the program is resumed, it continues execution from the return point of that function. Checkpointing is useful for running large number of tests from the same initial condition. Using a checkpointer has some disadvantages such as: a checkpointed program can be resumed only on the same host system, and currently such a program cannot be resumed under a debugger. This has motivated the next approach.

2) The faster-than-real-time simulation achieved by the Validator enables running open-loop simulations from stored inputs as a way to reach a specified initial state. Once the initial state is reached, the Validator connects to the plant simulator and continues with a closed-loop simulation. This approach offers the advantages of complete portability and ability to debug the entire simulation (both the open-loop and the closed-loop parts).

The *suspend*/*resume* methods of most actors are independent of the save/restore mechanism, which is visible only to the simulation director and to the plant actor. The related part of the director is given in Fig. 2 (in pseudo-code).

```
if (suspend_requested OR stop_requested){
  call suspend() on all actors;
  if (stop_requested){
    break from the main event loop;
          // this will jump to the wrapup phase
    }
  else { //suspend was requested, stop was not
    if (checkpointing_required){
      call checkpointing function;
      // program will resume at this point
      }
    call resume() on all actors;
    }
  }
} // continue with processing the event queue
```

Figure 2. Implementation of suspend/resume in the DE director

The plant actor connects to a plant simulator in the *resume* method, through an interface implemented by a connector object (shown by a rounded rectangle in Fig. 1). If needed, the connector is configured to save all its data traffic (application inputs and outputs) to the file system. The *fire* method uses the connector to send and receive plant signal values until the connection is closed by the plant simulator or by reaching the end of the file with stored inputs. In this case, the plant actor asks for simulation to be suspended. The suspend method checks if the simulation needs to be (possibly later) resumed. If so, it replaces the current plant connector with a TCP/IP connector and returns. Otherwise, it requests a simulation stop. The first connector object is created and configured in the initialize method (at simulation time zero). It can be either a TCP/IP one or a file-based connector. This design allows for combinations between the two state save/restore mechanisms. For example, one can use an input file to reach a specified initial state and then one can save that state by checkpointing.

I. USE CASES

To demonstrate significant use cases for the Validator, we have assembled a demo application consisting of several common automotive controllers. This system is described next.

A. Example of an Automotive Control System

The vehicle control system used for illustrating some of the Validator features includes controllers for: engine idle speed (ISC), spark angle (SAC), fuel rate (FRC), automatic gear shift (GSC), cruise (CRC), and main throttle (THC). Most of these controllers, as well as the models of the corresponding vehicle and engine parts, have been adapted from individual Matlab/Simulink demonstration models [5]. Also, their triggering conditions have been preserved. The closed-loop Simulink model is shown in Fig. 3.

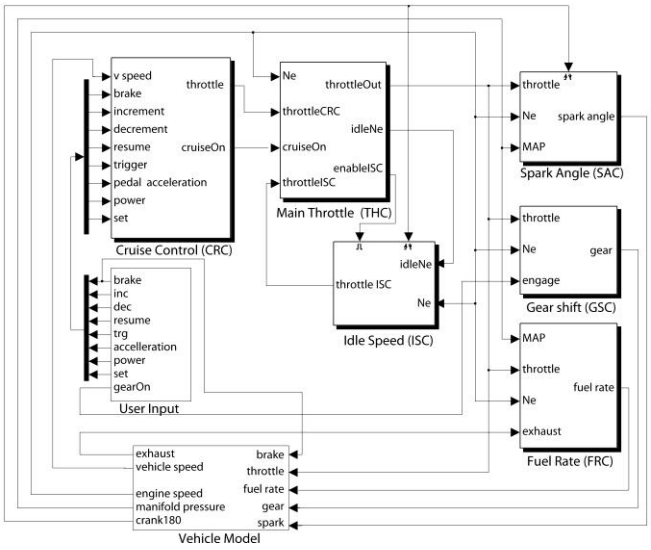

Figure 3. A vehicle control system used for illustrating Validator applications

The vehicle subsystem is a continuous-time model. The controllers are discrete-time, with sampling periods as follows: GSC – 0.04s, CRC – 0.01s, THC – 0.005s, FRC – 0.01s. The ISC and SAC are triggered every 180 degrees of crank angle. The THC generates the throttle signal, which is a control output representing the opening angle of the throttle valve. Note that the THC can disable the operation of the ISC (e.g., when setting the throttle according to the cruise control or acceleration pedal). Such control systems are typically developed by control engineers.

An implementation on an embedded computational platform is normally performed by software engineers and entails many design decisions, some of which may affect control performance. In this section we describe several uses cases illustrating how the Validator can be employed in evaluating such decisions. In these examples, various implementation versions are tested against the Simulink controller models in the classical block-box regression testing setup, where different versions of the same system are executed (simulated) with the same inputs and their outputs are compared. For a given test input, we consider that a software version A is closer to the original Simulink controller model than another version B if the outputs of A are closer (trajectory-wise) to the model's outputs than the outputs of B. While the Validator can be used in more elaborated white-box testing setups (where internal state variables can be recorded and compared) [6], black-box testing was sufficient for the purpose of illustrating the Validator applications in this paper.

B. Integration of Software Functions

When a new control function is added to an existing software system, its control performance may be affected by the platform specific configuration and by the place in the software where the function is added. More precisely, errors may occur due to the way a concurrent model component is placed into a specific sequence of executions, which is influenced by execution times and prioritization of tasks on the platform. To illustrate this situation, let us consider an implementation of the above control system on an OSEK operating system with three time-triggered tasks: task_1ms, task_5ms, and task_10ms, as well as one task triggered by the 180° crank angle event, called task_CA180. Assume that FRC is included in task_1ms, GSC is included in task_10ms (being executed every 4th invocation of the task), THC and ISC are called from task_5ms, and SAC is included in task_CA180. Also, the OSEK tasks are prioritized according to the rate monotonic policy for the periodic tasks, where longer periods have lower priority. The priority of task_CA180 is highest. Given this configuration, a decision has to be made regarding the inclusion of the CRC function. As its sampling period is 10ms, the most natural choice is the 10ms task. Let us call this configuration (A). The behavior of the control software in this case is tested against the reference model as shown in Fig. 4. Here, the Simulink model is executed in parallel with the Validator model of the software and platform. The "Controller Software" subsystem is linked to an S-function that connects to a Validator interface as shown in Fig. 1.

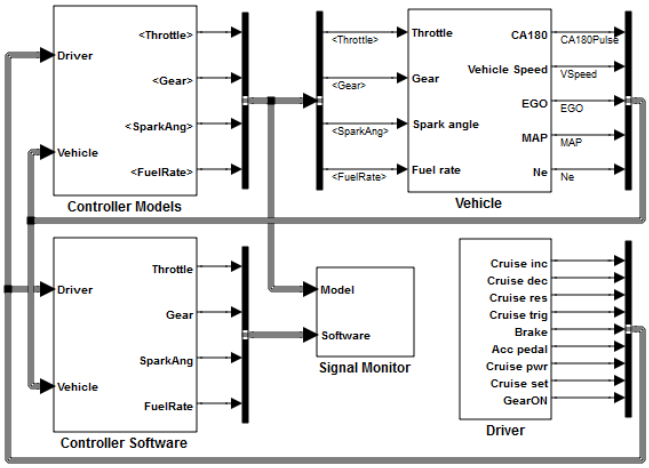

Figure 4. Simulink model for testing integration of software functions

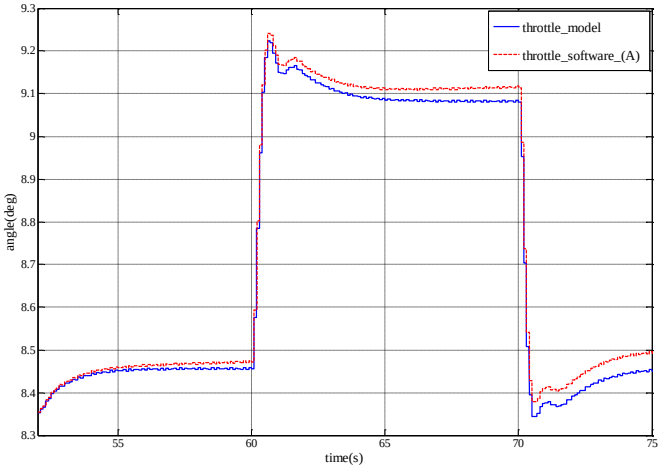

Figure 5. Throttle signals from a simulation of the model in Fig. 4

In one of the tested scenarios, the vehicle is accelerated until time 50 (all times in this example are expressed in seconds), when the acceleration pedal is released and the cruise controller is set. At time 60 the vehicle enters an uphill section of the road, which levels again at time 70.

A snapshot of the throttle output signal is displayed in Fig. 5. One can observe a deviation of the CRC throttle output from the reference signal. This difference does not occur when the software system is simulated with classical software-in-the-loop configuration, i.e., with no execution times for the software tasks. It follows that its cause must be related to execution times. Several options for placing the CRC function in the OSEK tasks have been tested. As the controller is not active before time 50, its placement has no impact on the system state before that time. Consequently, most of the tests were done by starting closed-loop simulations from the state of the system at time 50, using stored inputs to initialized the program state as described in Section II.B. These inputs were obtained by simulating the system with option (A) until time 50. This took about 110 seconds on a quad-core host PC with Windows 7 (64-bit), 1.73GHz, and 4GB of RAM, running Matlab R2010a. The Validator application was run on a Ubuntu10.10 virtual machine on the same host system. The initialization phase in subsequent simulations took about 3 seconds.

A configuration (B) where no significant throttle deviation occurs is the one where the CRC function is included at the beginning of the 5ms task (and called every second invocation of the task). Note that the CRC is executed at different moments in time and with different priorities in the two configurations. This can be seen from the task execution signals in Fig. 6, where integer numbers on the y axis represent the tasks in their inactive states: 2 for task_10ms, 3 and 4 correspond to task_5ms and task_1ms, respectively. A value of 0.4 added to the task number indicates a preempted state, and an added value of 0.6 shows that the task is in execution. One can see that the difference in execution moments of the two cases is relatively small (less than 1ms).

C. *Refinement of software and plant model*

The Validator enables verification of properties related to sensing and actuating by enabling one to compare the behavior of a system where signals from the plant model are directly connected to software variables, with the behavior of the system containing more refined models of platform and/or plant which may include sensor/actuator components. For example, the setup in Fig. 7 has been employed to evaluate the more realistic case where the engine speed and position are computed in the application software from a pulse signal generated by a teethed crank wheel model. The control output in this case is compared with the more ideal situation where the engine speed (Ne) and crank position (CA180) are passed directly to the software, as well as with the reference model-based behavior. Thus, two Validator instances run as concurrent processes on the host computer with two versions V1 and V2 of the application software, connected with two corresponding Simulink S-functions as shown in Fig. 7.

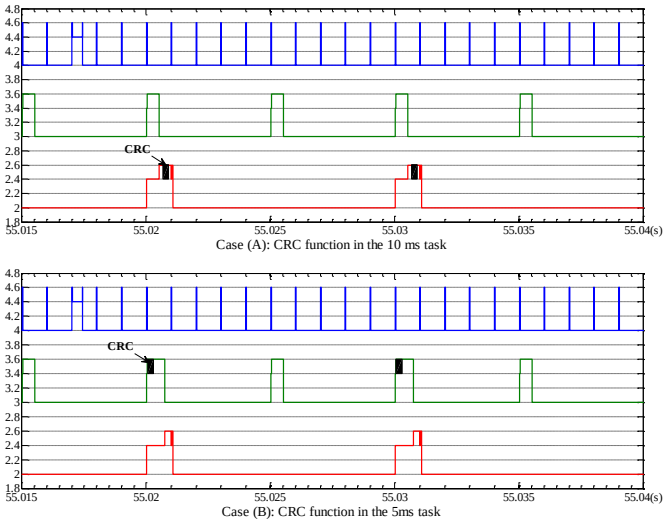

Figure 6. Task execution states and execution of the CRC function

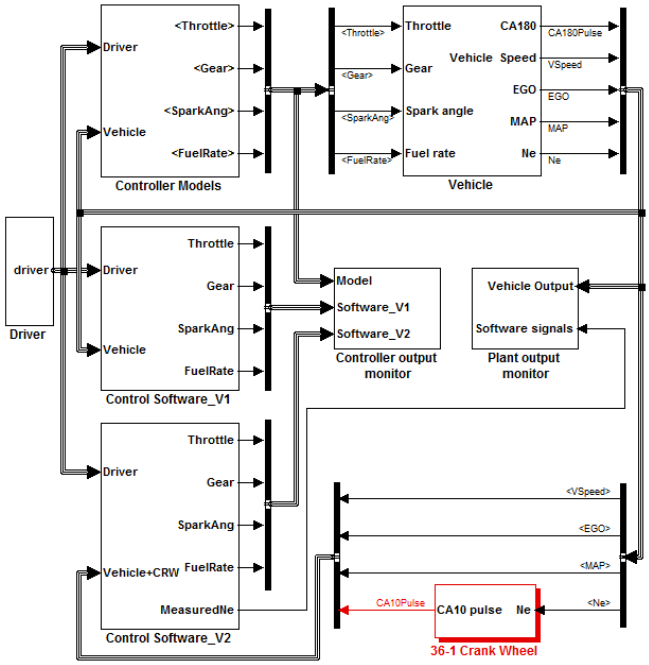

Figure 7. Testing a crank wheel sensor model

In the version V2 of the control system, the engine speed and position are calculated by measuring the time intervals between consecutive 10° crank angle events, which also allows for detecting the missing tooth of the crank wheel. When testing this system, an error was observed in the throttle output at idle speed, as shown in Fig. 8. As the V1 behavior is very close to the model-based signal, the V2 implementation was debugged by executing both V1 and V2 in parallel under the gdb debugger, going step by step in both versions and comparing signal values. The debugging user interface of the Eclipse CDT plugin enables smooth switching between the two parallel sessions, which use essentially the same view. The error was discovered in the timing: in the V2 case, the THC and ISC start operation after the crank synchronization phase, whereas in V1 (and in the model) they start at time zero. Thus, the first integration interval used by the ISC is smaller in the V2 case. By changing the start time of THC and ISC back to zero in V2, one obtains a throttle signals closer to the reference: the signal V2* in Fig. 8. This difference will disappear with the introduction of a corresponding start-up phase in the model.

D. Regression testing for timing properties

The Validator has been employed to verify timing properties of a large industrial legacy software system for engine control, henceforth called the ECS. Its ability to capture software execution times in simulation made it especially suitable in testing timing-related changes in the ECS. To improve the timing predictability of the application, the ECS has been re-engineered to satisfy timing specifications expressed in the Timing Definition Language (TDL) [7].

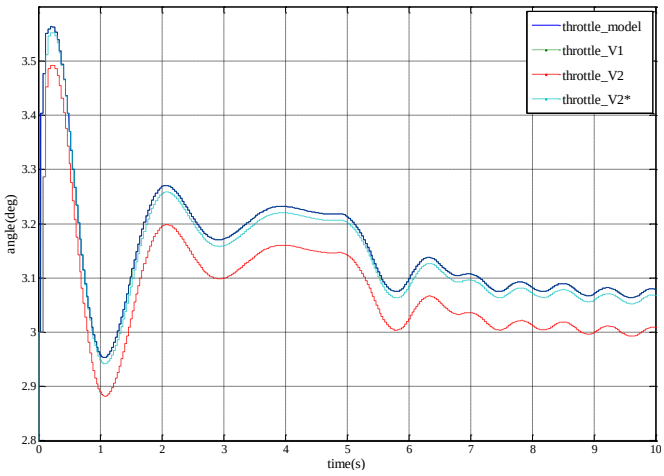

Figure 8. Testing a crank wheel sensor model

TDL is based on the Logical Execution Time model, where a fixed execution time (LET) is associated to an application function, and a runtime system ensures that the LET is observed during the operation of the system. Thus, the LET of a periodically executed function specifies the start and end time instants of every execution, effectively fixing the time interval between the moment when inputs are read and the moment where outputs are made available to the other functions in the software system. When this behavior is imposed on a legacy control software which has been developed based on shortest response time considerations, it is likely to introduce delays in data transfer, leading to possible degradation of control quality. Thus, it is necessary to assess various tradeoffs between additional resources required by a TDL implementation and the resulted behavior of the control system.

Two methods of implementing the TDL requirements have been tested. In variant (A), an important requirement was to ensure that minimal changes are done in the legacy code [8]. Thus, the hardware/OS configurations were not changed and no original source code line was modified. TDL was implemented only by adding functions and variables to the application at the top level. In variant (B), the original code was modified at specific lines down to low level functions and a TDL-dedicated timer with associated interrupt and ISR was added. This implementation had the advantages of enabling significant flexibility in choosing LETs, thereby enabling shorter delays, while requiring more resources and more intrusive changes in the legacy application.

The Validator was employed to test the two versions against each other and the original ECS in a setup similar to the one in Fig. 7 (without the crank wheel model). In Fig. 9, one can see an example of the consequences on the system outputs of the reduced delays achieved by approach B. Within the time window shown in the figure, there are six time instants at which the update in B is closer to the original value. One of them is at time 2.163s. There is also one time instant at which the situation is reversed: at 2.241s.

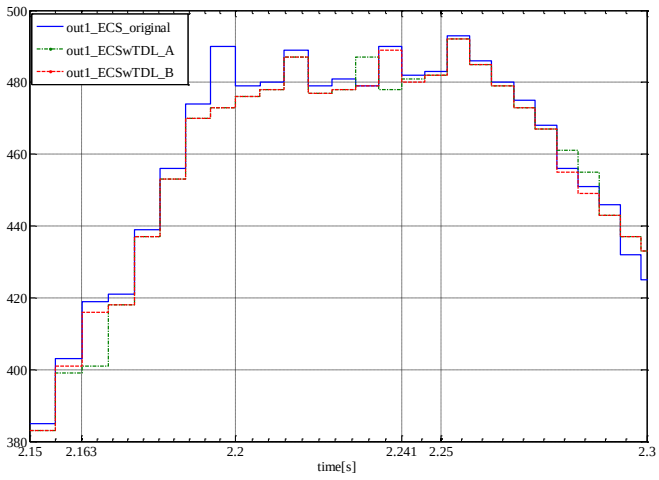

Figure 9. One output signal from three software versions

One can employ the Validator's advanced debugging features to understand the causes of such exceptions. The Validator enables system-level debugging: the hardware and OS models together with the instrumented application code form a C program that can be debugged with any available C debugger. The plant model can be executed in parallel under the Simulink debugger. The Validator allows traversing preemption points during forward and reverse debugging [1]. Note that debugging in the Validator can be live, i.e., the application code can be debugged during closed-loop simulations. This is in contrast to simulation techniques aimed specifically at improving debugging, where the application code is executed according to timed traces recorded from actual executions on the hardware (so-called time machines).

II. Conclusion

The Validator is related to other tools for supporting co-design of control systems and their real-time implementations – see [9] for a survey. The unique place of the Validator in this landscape is given by several key factors, as follows:

(1) It is an independent simulator written in C, based on actor-oriented principles rooted in Ptolemy II. Thus, it can interact with different plant simulation tools such as Matlab/Simulink and Modelica. Moreover, it provides significant flexibility for running simulations. For instance, the save/restore feature presented in Section II.A can be hardly achieved (for large C applications) with similar tools based on Matlab/Simulink such as TrueTime [10] and AIDA [11].

(2) The Validator is integrated with the Eclipse IDE, tapping into the power of the Eclipse's CDT (C Development Tool) plugin. This provides a specialized environment familiar to the software developer, for tasks such as testing and debugging. Moreover, this enables rapid development of custom program analysis and instrumentation tools based on CDT, as well as custom graphical user interfaces (GUIs) based on Eclipse's GEF (Graphical Editing Framework) plugin. Such a GUI is already provided in the tool suite. Validator simulations can also be configured and launched from Simulink plant models via a dedicated GUI component.

(3) The Validator specifies a fixed granularity for execution time information, that is, the basic block of source code. The Validator comes with companion tools for automatically extracting this information from other tools such as the AbsInt's aiT [3] (which works at the binary level), and instrumenting the source code accordingly. In contrast, tools such as TrueTime [10] and the Ptolemy Multitasking Domain [12] set this granularity at the task level or require the user to manually specify it.

In the Validator, the simulation accuracy depends on the employed execution time estimates. Obtaining precise execution times of software on today's embedded processors is a notoriously difficult problem and further research is needed to advance along this direction. Nevertheless, using estimates obtained with commercially available tools, the Validator has already proved useful in early evaluation of control software both in model-based development processes and in legacy software re-engineering.

References

[1] S. Resmerita, P.Derler, W.Pree, and K.Butts, "The Validator tool suite: filling the gap between conventional soft-ware-in-the-loop and hardware-in-the-loop simulation environments," in *Real-time Simulation Technologies: Principles, Methodologies, and Applications*, K.Popovici, P.J. Mosterman, Eds., CRC Press, to appear, 2012.

[2] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, *Heterogeneous concurrent modeling and design in Java (Volume 1: Introduction to Ptolemy II)*, EECS Department, University of California, Berkeley, UCB/EECS-2007-7, 2007.

[3] Absint. aiT worst-case execution time analyzers. http://www.absint.com/ait/, 2011.

[4] J. Ansel, K. Aryay, and G. Coopermany, "DMTCP: Transparent checkpointing for cluster computations and the desktop", in *Proc. The 2009 IEEE International Symposium on Parallel & Distributed Processing*, Washington DC, 2009, pp. 1-12.

[5] MathWorks®. Matlab® 7.10 and Simulink® 7.5 (R2010a). http://www.mathworks.com/products/simulink/, 2010.

[6] T. Xie and D. Notkin, "Checking Inside the Black Box: Regression Testing By Comparing Value Spectra", *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, pp. 869-883, October 2005.

[7] J.Templ et. al. "Modeling and simulation of timing behavior with the Timing Definition Language (TDL)", in *Real-time Simulation Technologies: Principles, Methodologies, and Applications*, K.Popovici, P.J. Mosterman, Eds., CRC Press, to appear, 2012.

[8] S. Resmerita, et. al, "Migration of Legacy Software Towards Correct-by-Construction Timing Behavior", in *Proc. Monterey Workshops 2010*, R. Calinescu and E. Jackson,Eds., LNCS 6662, 2011, pp. 55–76.

[9] M. Törngren, D. Henriksson, K.-E. Årzen, A. Cervin and Z. Hanzalek, "Tool supporting the co-design of control systems and their real-time implementation: current status and future directions", in *Proc. 2006 IEEE Conference on Computer Aided Control Systems Design*,2006, pp.1173-1180.

[10] A. Cervin, et. al , "Control loop timing analysis using TrueTime and Jitterbug," in *Proc. 2006 IEEE Computer-Aided Control Systems Design Symposium*, 2006, pp.1194-1199.

[11] O. Redell, J. El-Khoury, and M. Törngren, "The AIDA tool-set for design and implementation analysis of distributed real-time control systems", *Journal of Microprocessors and Microsystems*, 28:4, pp. 163-182, 2004.

[12] J. Liu and E. Lee, "Timed multitasking for real-time embedded software", *IEEE Control Systems Magazine*, 23:65–75, 2002.