The Validator tool suite: filling the gap between conventional software-in-the-loop and hardware-in-the-loop simulation environments

Stefan Resmerita\textsuperscript{1,2}, Patricia Derler\textsuperscript{3}, Wolfgang Pree\textsuperscript{1,2}, Kenneth Butts\textsuperscript{4}

\textsuperscript{1} C. Doppler Laboratory \textit{Embedded Software Systems}, Univ. Salzburg, Austria
\textsuperscript{2} Chrona.com, Salzburg, Austria and Torrance, California
\textsuperscript{3} Electrical Engineering and Computer Sciences (EECS) department, University of California, Berkeley
\textsuperscript{4} Toyota Technical Center, Ann Arbor, Michigan

Abstract

Software-In-The-Loop (SIL) and Hardware-In-The-Loop (HIL) simulation environments represent established methods and tools for testing real-time embedded systems before they are deployed in products. Nevertheless, some key features that we regard as crucial for improving the quality of embedded systems are missing in both conventional SIL and HIL approaches. What we have called the Validator, a cutting-edge simulation environment for verification and validation, fills that gap. The Validator offers the advantages of SIL simulations in terms of cost and simulation speed as well as the advantages of HIL simulations in terms of accuracy. The new key feature of the Validator is advanced debugging of embedded systems that deals with timing behavior as if the software would be executed on a target platform. The Validator also offers regression testing at a level of granularity that leads to a solid comparison between several versions of a system, for example, a legacy system and its reengineered or enhanced artefact. The Validator was applied to large industrial legacy systems for simulating the computational part in closed-loop with continuous-time plant models in MATLAB\textsuperscript{\textregistered}/Simulink\textsuperscript{\textregistered}. For that purpose, the Validator allows co-simulation with other simulation tools based on time synchronization protocols. This chapter
presents the Validator mainly from the user perspective. For a detailed description of the underlying concepts and how the Validator is implemented, we refer to [1].

1 Solid system verification and validation needs an improved simulation support

An embedded system operates in a physical environment with which it interacts through sensors and actuators. An important class of real-time embedded systems is represented by control systems. In this case, the embedded software consists of a set of (controller) tasks and the physical system under control is referred to as the plant. Figure 1 sketches the typical architecture of an embedded control system.

![Figure 1 Typical architecture of an embedded system.](image)

Simulation is an approach for testing embedded systems before they are deployed in real-world operation. In simulation, the plant is represented by a software model executed on a host computer, typically a PC. In a Hardware-In-the-Loop (HIL) simulation, the entire embedded system (embedded software consisting of the controller tasks executing on the target platform) is operated in closed loop with the plant model, which is executed in real time on a dedicated computer [20]. Since the embedded software is executed in real time on the target platform, HIL simulations can be used to verify the real-time properties of the embedded system. Thus, any
difference between the behavior of the embedded system in a HIL simulation and the corresponding behavior in the real world is due to the abstractions made in plant modeling.

In a Software-In-the-Loop (SIL) simulation, the embedded software consisting of the controller tasks is executed on a host computer other than the target platform, in closed loop with the plant model. Both simulations (of the controller and the plant model) are typically executed on the same host computer. The SIL model of an embedded system contains the embedded software and an abstraction of the target platform. This abstraction determines how close the software execution in the SIL simulation is to the HIL simulation, provided that the same plant model is used. It ranges from a minimal representation of the target platform that enables only testing of functional (transformational or processing) properties of the software, to full-fledged hardware simulators (called instruction set simulators, ISS) which lead to system behavior close to a HIL...
simulation, while offering better observability of software executions. Pure functional simulations are fast, but do not allow the testing of timing properties of the embedded system. ISS can be employed for timing analysis [19], but they are extremely slow and expensive. Figure 2a summarizes the characteristics of conventional SIL, ISS-based SIL and HIL simulations.]

**Figure 2b** Validator: advanced SIL simulation in between conventional SIL and HIL simulations

Figure 2b\(^1\) summarizes the features as well as the advantages and disadvantages of state-of-the-art SIL and HIL simulations in comparison with a Validator simulation. Note that the Validator replaces the ISS-based simulation.

Figure 3 refines the comparison between the Validator and SIL/HIL approaches. The Validator unifies characteristics of both a SIL and a HIL simulation. The Validator has the flavor of a SIL simulation as it does not require a target platform for executing the embedded software. On the other hand, the Validator separates the simulation of plant and controller tasks as in a HIL simulation.

\(^1\) Parts of the picture are taken from [http://www.mathworks.com/products/xpctarget/](http://www.mathworks.com/products/xpctarget/) and are courtesy of MathWorks
Real-time behavior in the Validator. An important aspect is the simulation of real-time behavior. A simple example that illustrates in which respect the Validator is better than typical SIL simulation tools is shown in Figure 4. Consider three concurrent tasks, called DynamicsController (DC), MotorController (MC), and ParkingController (PC), that communicate through a shared (global) variable called angle. The code of the tasks is sketched in Figure 4a. The tasks DC and MC are periodic with periods equal to 5ms and 1 ms, respectively. The task PC is event triggered. Assume that they are deployed on a real-time operating system with fixed priority preemptive scheduling where the priorities of the periodic tasks are assigned by a rate monotonic policy. Thus, the MC task has a higher priority than the DC task. Moreover, consider that the PC task has highest priority.
A snapshot of real time behavior of the application is depicted in Figure 4b, which indicates the sequence of accesses to the variable *angle* by the three tasks. Note that task DC is triggered first, at 5ms and then it is preempted by MC at 6 ms, before writing into the variable *angle*. Thus, MC reads from *angle* first. Thereafter DC resumes and writes the angle, then it is preempted by PC, which reads the variable. Figure 4c shows the behavior of the application in a pure functional simulation, where each function is completely executed at the triggering time. In other words, the code is executed in logically zero time. Such a simulation can be obtained in Simulink® [22], for example, by importing the C code as so-called S-function(s) in Simulink. Figure 4d presents a timed-functional simulation, where each task has a specified execution time and sharing of processor time among tasks in the system is also simulated. A timed-functional model includes a scheduling component, to decide when a triggered task obtains access to the processor. When the task is started, its code is still executed in zero time, thus using the inputs available at that moment, however the outputs of the task are made available after the specified execution time has elapsed. Examples of SIL simulation environments that offer task-level timed simulation are the Timed Multitasking Ptolemy domain [2] and TrueTime [3]. Notice that the order in which the three tasks access the variable *angle* is different in the two simulations compared to the real-time case.
On the other hand, a simulation with the Validator would reflect the same order of accesses as in the real-time behavior shown in Figure 4-b. This requires a detailed execution time analysis and a corresponding instrumentation of the embedded software by the Validator support tools as described in the next section.

## 2 Architecture of a simulation with the Validator

Remember that a simulation with the Validator is a closed-loop co-simulation of the plant under control and the controller tasks. Currently, the Validator supports continuous-time plant models in MATLAB® [21] and Simulink. For that purpose a communication interface was implemented as a MATLAB/Simulink S-function (see Figure 5). The underlying protocol for communication between the plant and the simulation with the Validator is TCP/IP. So both simulations can execute in parallel on the same computer or on different cores or, for example, for efficiency reasons, on different computers. We will extend the Validator to support co-simulation with other simulation environments in the future. For that purpose communication interfaces must be
implemented for the particular simulation environment.

The Validator also offers a file reader for processing time-stamped values of input data from recorded signals. This is useful for regression testing as discussed in Section 4.2.

**Figure 5** Architecture of a closed-loop simulation with the Validator

The Validator simulation engine is a discrete event simulation that takes the platform specifications into account. The platform specifications comprising the operating system (OS), the communication bus, hardware timers etc. are plug-ins of the Validator simulation engine. The lower half of Figure 5 sketches this aspect of the architecture of a Validator simulation. The discrete event simulation controls which of the tasks are executed once the control flow gets back to the discrete event simulation from a task execution. For example, based on the scheduling strategy used by the OS, a higher priority task must interrupt one with a lower priority. In such a situation the discrete event simulation will switch the execution to the appropriate task. Dashed arrows express this control flow between the instrumented tasks and the discrete event simulation in Figure 5. In an analogous way the Validator takes care of the appropriate reading
of sensor values and writing of actuator values.

Figure 6

Figure 6 illustrates the discrete event simulation in the Validator in more detail. As the discrete-event simulation of the controller tasks proceeds from what we call a spot (S) within a task to the next spot, the control flow between tasks and the discrete event simulation constantly switches back and forth. In the sample scenario in Figure 6 the discrete-event simulation starts executing sampleTask(). This task executes till it reaches the first spot and returns control to the discrete-event simulation. Based on the platform specification, the discrete-event simulation decides to interrupt sampleTask() and to give control to anotherTask(), for example, because that one was triggered for execution and has higher priority than sampleTask(). As anotherTask() has only one spot at the end, it executes completely and then returns the control back to the discrete-event simulation. Now the discrete-event simulation gives back control to sampleTask(). The discrete-event simulation continues with the execution of sampleTask() also at the other two spots.
Basic features of the Validator

Let us conclude this overview of the architecture and some core simulation concepts by summarizing the features that result from a bare-bone set-up of the Validator:

Variable monitoring. The Validator allows the logging of the time-stamped values of selected variables (global variables and variables local to tasks) to a file. The variable monitor in Figure 5 corresponds to that functionality.

Stop and restart simulation runs. The Validator allows stopping a simulation and saving the state of the overall simulation, that is the simulations of both the plant and the controller tasks. A simulation can later be restarted from a saved state.

3 Set-up of a simulation with the Validator

The current version of the Validator basically supports the co-simulation of a plant represented as variable-step model in MATLAB/Simulink with the controller software written in C. As an advanced feature for reengineering existing controller tasks or adding controller tasks, these tasks can be modeled in MATLAB/Simulink and the Validator simulates then the behavior of both the existing unchanged tasks and the modified or new controller tasks. The only constraint is that the modified or new controller tasks are modeled with discrete time semantics in MATLAB/Simulink. Let us now focus on the typical use case, that is, the co-simulation of a plant represented as variable-step model in MATLAB/Simulink with the controller software written in C.

Target platform specification. For an accurate simulation of the controller tasks on a virtual platform, the Validator must have configuration information about the target platform, which is
provided by setting properties of the corresponding model components. To specify this information, we use Ptolemy’s front end, as the original research prototype of the Validator was implemented harnessing Ptolemy’s discrete event simulation. The screen shot in Figure 7 exemplifies the specification of the behavior of an interrupt service routine (ISR).

![Figure 7: Specification of ISR behavior with the Ptolemy front-end](image)

An ISR is represented as a so-called actor in Ptolemy. The actor-oriented programming model is in essence a dataflow-based programming model in which data flow from actor to actor. When activated, each actor performs its specific data processing. An actor-oriented environment such as Ptolemy must account for the execution order of actors. The Validator library, which is used to specify the target platform, comprises various kinds of actors:

- Hardware actors model functionality and timing of common hardware parts such as interrupt controllers, timers, bus controllers, hardware sensors, and hardware actuators.
- Operating system actors, which implement the functionality of the operating system on
the target platform, including scheduling, resource management, and communication between tasks. Currently, the Validator provides actors for the OSEK operating system. Note that actors are best understood as plug-ins to the discrete event simulation of the Validator, providing the various platform details.

Task source code annotation. In addition to specifying the target platform, the source code of the controller tasks must be instrumented with callbacks to the simulation in the Validator. Detail on which aspects require a callback is available in other work [1]. All the spots in the source code are instrumented with callbacks. An example of a type of spot are accesses to global variables. Between each pair of spots, the execution time must be determined. This is another crucial aspect of target platform information that the Validator must have to achieve its accuracy. From the Validator user point of view it is only relevant that both the instrumentation and execution time estimation can be automated. Overall, a preparation of the controller tasks for a simulation with the Validator involves the following steps:

1) Execution time analysis of the application code. This is performed with existing program analysis tools such as AbsInt’s Advanced Analyzer (a3) tool [4]. To increase the accuracy of the estimates, generally details about the architecture of the execution platform must be made available to such tools.

2) Instrumentation of the code with execution time information.

3) Instrumentation with callbacks to pass control to the Validator simulation engine for the execution of the tasks.

4) Generation of what we call the Validator interface code between the Validator simulation engine and the tasks.
In the Validator, these steps are mostly automated by a tool set that achieves a straightforward preparation process. Nevertheless, this automation requires information about the hardware/software architecture, such as the list of lines of code where global variables are accessed.

4 Embedded system validation and verification with the Validator

This section describes two principal usage scenarios where the Validator excels compared to state-of-the-art SIL and HIL simulations: advanced debugging of embedded systems and the incremental reengineering of existing embedded systems, including regression testing. Case studies illustrate each particular usage scenario of the Validator.

4.1 A simulation with the Validator as basis for advanced debugging

The key feature of the Validator that allows advanced debugging is that at every source code line in the controller tasks, the overall simulation, that is, of both the controller tasks and the plant, can be stopped. Then variables can be inspected and modified, external code can be executed, etc. Any C debugger can be attached to the Validator to perform the common debugging activities on the controller tasks. A state-of-the-art HIL simulation environment does not offer debugging capabilities. On the other hand, the impreciseness of state-of-the-art SIL environments makes debugging unattractive or at least less helpful. Whereas the accuracy of a Validator simulation makes debugging a valuable means for the validation and verification of embedded systems. Figure 8 shows the schematic attachment of a debugger to the Validator. The jet fighter picture represents the plant model.
Figure 8  Attaching a debugger to the Validator

The screen shot in Figure 9 shows a sample set of controller tasks in Eclipse with the gnu debugger (gdb) plugin. The screen shot is discussed in more detail below. We have used gdb as it also supports reverse (or historical) debugging. This allows the following advanced debugging:

Once you have turned on reverse debugging, the debugger records all state changes. So if the debugger stops execution at a breakpoint you can not only step forward as usual, but also step backward from that point in the code. For example, you want to find the cause why an actuator value exceeds a certain limit. In this case you would set a conditional breakpoint where the actuator value is set. When the condition holds, the execution stops there and you can step back step by step.

The following sequence of screen shots illustrates reverse debugging from a developer's point of view, with Eclipse and gdb as the graphical front end of the Validator.

In the state shown in Figure 9, we just entered the debugging mode by pressing the debug button (the bug in the menu bar on the top left side of the overall window). The execution stopped at an unconditional break point in file dynamicsController.c. The statement at the breakpoint is an assignment statement in which the value of a variable called rtb_deg2rad2 is set. The line is
highlighted in green in the tab labeled dynamicsController.c. According to the subwindow in the top right part of the window, the value of rtb_deg2rad2 is zero.

![Debugging Session in Eclipse](image)

**Figure 9** Start of a sample debugging session in Eclipse with the GNU debugger plugin

As a next step we turn reverse debugging on by pressing the corresponding icon-button (see Figure 10). When stepping forward the debug control panel changes to reflect the feature of reverse debugging, that is, being able to step forward and backward (see Figure 11).
Let us assume we just stepped forward one statement (see Figure 12). The assignment statement where we had originally stopped at the breakpoint has apparently changed the value of variable rtb_deg2rad2 from zero to approximately 1.807.
We can now press the button to go back one step in the debugging process. Figure 13 shows the result, that is, as expected, the value of variable rtb_deg2rad2 is again zero. Note that the discrete event simulation of the Validator was implemented such that reverse debugging also functions across multiple tasks. For example, if you have turned on reverse debugging, step forward and the task is interrupted by another task, that is, the simulation switches to another task, you can still step back up to the point where you started reverse debugging.

![Figure 13 A step back.](image)

As future extension to the Validator we will add the feature to also be able to set break points in the plant simulation that halt the overall simulation.

### 4.2 Simulation with the Validator to reengineer legacy systems

The initial motivation for developing the Validator was to provide solid support for the incremental migration of the Engine Controller System (ECS) of a large automotive manufacturer to a version in which the timing behavior is explicitly modeled with the Timing Definition Language (TDL) [23][24]. For that purpose, the behavior of the legacy ECS and the TDL-based ECS
should be compared in detail in a SIL simulation that is as close to the behavior on the actual platform as possible. Figure 14 shows the generic set-up for this kind of regression testing by means of the ECS example. The Validator can simulate both versions in parallel.

Figure 13 Regression testing with the Validator.

Note that the ECS comprises millions of lines of code, mostly written in C, and runs on top of an OSEK operating system. This required an efficient implementation of the discrete event simulation of the Validator and an efficient co-simulation with the automobile engine (= plant) model. The engine model is represented in MATLAB/Simulink. In order to accurately capture the times of the crank angle events, the engine model is simulated with a variable step solver.

Implementation of the TDL semantics in the re-engineered ECS [23] required a dedicated TDL component called TDL-Machine to be executed every 0.5ms from the task with highest priority in the system. The TDL-Machine used additional global variables to store and restore values of original global variables at certain points in time.

The highest priority task in the original ECS had a period of 1ms. To avoid introducing a new task, it was decided to change the original task to have a period of 0.5 ms, to call the TDL-Machine at every task invocation, and to execute the original task code every second invocation.
Thus, the execution period of the original code was unaffected.

**Sample analysis.** Let us take a look at one of the results of the regression tests: Figure 14 shows a selection of three signals monitored during a simulation of the two ECS versions with the Validator. Signals S1 and S2 are similar in the two versions. One can notice some delays introduced in the modified version by the execution times of the TDL-Machine. Signal S3 differs significantly towards the end of the simulated time frame.

The cause of the difference could be found by an investigation of task execution profiles. Task execution profile plots are shown in Figures 15 (a)–(c). In each plot every task is assigned an identification number (ID). The execution state of a task with ID $i$ is represented by a signal with y-coordinates between $i$ and $i + 0.6$. A level of $i + 0.6$ indicates the execution mode $E$, a level of $i + 0.4$ means preempted mode $P$, a level of $i + 0.2$ indicates the waiting mode $W$ and a level of $i$ means the task is in the suspended mode $S$. The plot in Figure 15(a) is obtained from the simulation of the original software, the plot in Figure 15(b) is obtained from the simulation of the software with the TDL-Machine and its execution time, and the plot in Figure 15(c) corresponds to the simulation of the software with the TDL-Machine but without its execution time. In case of Figure 15-(c), the TDL-Machine was executed in simulation, but its execution time was set to zero, for the purpose of debugging.
The ID of the highest priority task is 8. In the simulation represented by the plot in Figure 15(a), the execution time of this task at time instant 5 is approximately 0.05 ms. In the simulation of the plot in Figure 15(b) the same execution requires about 0.20 ms. In Figure 15(c) the same task execution takes about 0.17 ms. It follows that the main difference between the executions in Figure 15(a) and Figure 15(b) is not given by the execution time of the TDL-Machine, but it is due to the different state of the hardware platform resulted from the execution of the TDL function. Since the TDL-Machine performs many accesses to new memory locations, the main difference occurs most probably in the cache state. If the delays in the signals in the new version are not acceptable, then one should focus on minimizing the effect of the TDL-Machine on the platform state rather than on minimizing the execution time of the function code. For example, the TDL-Machine could be changed to only operate on local variables, or additional variables used by this function could be stored in the processor’s internal memory space.
Figure 15 (a) Original ECS.

Figure 15 (b) ECS with additional functionality.

Figure 15 (c) ECS with additional functionality whose execution time is set to zero.
5 Related work

This section groups existing work in the area of simulation of embedded applications into two categories, related to the main features of the Validator: the ability to synchronize with an external plant model for closed-loop simulation (co-simulation) and the special focus on legacy software and its execution time.

5.1 Co-simulation

The main purpose of co-simulation is validating the functionality of hardware (HW) and software (SW) components by simulating two or more system parts that are described on different levels of abstraction. The challenge is the interface between the different abstraction levels. A simulation of the system should be possible throughout the entire design process where the model of the same component is refined iteratively [5]. Co-simulation as a basis for co-design and system verification can be performed in various manners where typically a trade-off between accuracy and performance must be made [6]. Various commercial and academic co-simulation frameworks have been proposed in literature; surveys can be found in [6, 7, 8].

In HW/SW co-simulation the processor model is responsible for connecting hardware and software models. The processor can be modeled at gate level, which is the most accurate but also the slowest solution, with clock cycle accuracy or on an instruction-set level. Faster co-simulation tools do not model the processor but implement a synchronization handshake [9]. Some co-simulation environments also provide a virtual operating system to emulate or simulate the hardware [10].

Many approaches use instruction set simulators (ISS) in order to obtain correct timing information. However, ISS are slow because of the fine granularity of the simulation. Performance issues are addressed for instance with caching [11] and distributed simulation by applying distributed
event-driven simulation techniques.

The co-simulation framework used by the Validator does not provide a model of the CPU and does not employ an ISS because of performance reasons. We work with execution time at the source code level. Hardware components are modeled at a higher level of abstraction. The simulation tool was in an original prototype implemented based on Ptolemy. Another Ptolemy based co-simulation approach can be found in [11].

5.2 Modeling and Simulating Legacy Code

There are various approaches that generate models from legacy code, but only a few of them include the timing aspect in the modeling. Some software reverse engineering efforts take the software and find equivalent modeling constructs in a modeling language to reconstruct the same behavior as exhibited by the software. An example is provided in [12] where C programs are reverse engineered to Simulink models. This, however, usually leads to complex models that are not understandable and thus do not aid in gaining new insights in the embedded software system. Code instrumentation and delaying of task execution to obtain a certain behavior is used by Wang et al. [13]. In this approach the authors employ code instrumentation to generate deadlock free code for multi-core architectures. Timed Petri nets are generated from (legacy) code by instrumenting the code at points where locks to shared resources are accessed in order to model blocking behavior of software. A controller is synthesized from the code and used at run-time to ensure deadlock-free behavior of the software on multi-core platforms by delaying task executions that would lead to deadlocks. The objective of the Validator is different: to replicate the real-time behavior of a given application (within certain accuracy limits). Thus, the Validator does not alter the functional behavior of the application.

In [14], a formal framework is described for building timed models of real-time systems in order
to verify functional and timing correctness. Software and environment models are considered to operate in different timing domains that are carefully related at input and output operations. A timed automaton of the software is created by annotating code with execution time information. The tool presented in that work restricts the control part of task implementations and the plant model to Esterel programs. The authors state that for tasks written in general purpose languages such as C an analysis must reveal observable states and computation steps. The Validator provides such an analysis and can be used for application code written in C and environment models in Matlab/Simulink.

The benefits of modeling all aspects of an embedded system at a suitable level of abstraction are well known and have been addressed in the platform-based design approach. Tools such as Metropolis [15] offer a framework for platform-based design where functionality and platform concerns are specified independently. A mapping between functionality and a given platform must be provided. Representation of components and the mapping between functional and architectural networks is possible at different levels of refinement. The purpose of Metropolis is to support the top-down design process. All system components are modeled in an abstract specification language which is parsed to an abstract syntax tree and provided to back-end tools for analysis and simulation. Although Metropolis allows the inclusion of legacy components, its main goal is not a bottom-up analysis of legacy systems but a top-down specification of the required behavior and a specification of the platform. In the approach presented here, the behavior of the components such as the data flow or temporal constraints are retrieved from simulation. As opposed to Metropolis, we do not require a specification of the functionality in a metamodel language which is further translated into SystemC [16] for simulation. Our approach directly includes the software as simulation components. Simulation tools related to the
Validator are TrueTime [3] from the academia and the commercial tool ChronSim from INCHRON [17]. Compared to these tools, the Validator offers a series of advanced features, as described in the chapter.

6 Conclusions

The Validator is a tool suite for a significantly improved SIL verification and validation of real-time embedded applications. The first prototype was developed with Ptolemy and was originally called the Access Point Event Simulator (APES). Chrona [18] developed the Validator as product out of APES. Chrona’s Validator has evolved to a product version that scales to the simulation of real-world embedded software consisting of millions of lines of code. The Validator achieves time-functional simulation of application software and execution platform in closed-loop with a plant model. The Validator works independently of a specific domain as long as a simulation model for a particular plant is available.

A simulation with the Validator is based on a systematic manner to instrument the application code with execution time information and execution control statements which enables capturing real-time behavior at a finer and more appropriate time granularity than most of the currently available similar tools. Chrona’s Validator is able to simulate preemption at the highest level of abstraction that still allows for capturing the effect of preemption on data values, avoiding at the same time the slow, detailed simulation achieved by instruction set simulators. Moreover, the Validator can operate in closed loop with plant models simulated by a different tool. Also, Chrona’s Validator enables traversing preemption points during forward and reverse debugging of the application. In the Validator, one can start a simulation from a previously saved state. Being implemented entirely in C, the Validator can be easily interfaced or even integrated with existing simulation tools such as MATLAB/Simulink.
Acknowledgements

We would like to thank Edward Lee (University of California, Berkeley) for valuable insights during numerous discussions of the Validator concepts and his support in implementing an earlier prototype of the Validator with Ptolemy II.
References


ACM.

using hw/sw cosimulation and instruction set simulator. In CODES/CASHE ’98: Pro-
ceedings of the 6th international workshop on Hardware/software codesign, pages 65–69,

[12] Alberto Sangiovanni-Vincentelli. Defining platform-based design. EEDesign of EE-
Times, February 2002.

The theory of deadlock avoidance via discrete control. In POPL ’09: Proceedings of the
36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming lan-
guages, pages 252–263, New York, NY, USA, 2009. ACM.

systems from application software. Proceedings of the IEEE 91(1):100–111, January
2003.

Meyerowitz, Roberto Passerone, Alessandro Pinto, Alberto Sangiovanni-Vincentelli,
Alena Simalatsar, Yosinori Watanabe, Guang Yang, Qi Zhu. Platform-Based Design and
Frameworks: Metropolis and Metro II. In Model-Based Design of Heterogeneous Em-

[16] Frank Ghenassia. Transaction-Level Modeling with Systemc: Tlm Concepts and Appli-
cations for Embedded Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA,
2006.


