Validate implementation correctness using simulation: the TASTE approach

Julien Delange[1], Jérôme Hugues[2], Pierre Dissaux[3]

1: European Space Agency, Keplerlaan 1, 2201AZ Noordwijk, The Netherlands
2. Toulouse University/ISAE, 10 Avenue Edouard Belin, 31055 Toulouse, France
3. Ellidiss Technologies, 24 Quai de la Douane, 29200 Brest, France

Abstract: High-integrity systems operate in hostile environment and must guarantee a continuous operational state, even if unexpected events happen. In addition, these systems have stringent requirements that must be validated and correctly translated from high-level specifications down to code. All these constraints make the overall development process more time-consuming. This becomes especially complex because the number of system functions keeps increasing over the years.

As a result, engineers must validate system implementation and check that its execution conforms to the specifications. To do so, a traditional approach consists in a manual instrumentation of the implementation code to trace system activity while operating. However, this might be error-prone because modifications are not automatic and still made manually. Furthermore, such modifications may have an impact on the actual behavior of the system.

In this paper, we present an approach to validate a system implementation by comparing execution against simulation. In that purpose, we adapt TASTE, a set of tools that eases system development by automating each step as much as possible. In particular, TASTE automates system implementation from functional (system functions description with their properties – period, deadline, priority, etc.) and deployment (processors, buses, devices to be used) models.

We tailored this tool-chain to create traces during system execution. Generated output shows activation time of each task, usage of communication ports (size of the queues, instant of events pushed/pulled, etc.) and other relevant execution metrics to be monitored. As a consequence, system engineers can check implementation correctness by comparing simulation and execution metrics.

Keywords: AADL, TASTE, code generation, instrumentation, GTKWave

1. Introduction

High-integrity systems are operating in hostile environment and must be designed to work safely even if unexpected event happens. They also perform mission- or life-critical operations so that a failure may have catastrophic impacts (mission abortion, loss of life, economic impacts, etc.).

Thus, engineers must demonstrate implementation correctness regarding its operating and environment constraints. However, this is difficult because the number of systems functions increase significantly by the time and they also have a large number of requirements that originate from different domains (electric, physics, etc.).

One solution is to abstract system representation and specify its properties and constraints using models. Such abstraction level may be used for different purposes: communication (exchange of models between engineers from different area of specialization), validation, etc. Depending on their engineering domain and their background, different languages and tools are used.

As such, model-based development (MBD) is now a widely accepted solution to perform a whole range of analysis prior to actually implement the system. This “virtual” test-bench relies on a well-defined set of concepts for modelling the building blocks of a system (interfaces, associated types), topology of connections and eventually links to concrete software and hardware elements. This is the path chosen by many tools like Simulink, SCADE, and also standards like OMG/SysML[8], OMG/MARTE [9] or the SAE AADL[11].

Companion tools support designer activities when modelling and navigating its model, but also to understand its actual behaviour through simulation (e.g. model animation or execution of scenarios step-by-step), verification activities through complete enumeration of the model state space (model checking techniques) and coupling with observer like techniques. In addition, traceability of model elements to requirements can be performed. Hence, the designer can be convinced his model is complete enough to faithfully represent a potential solution to its initial objective.

2. Problem

However, if models are accurate for abstracting system, turning requirements into a candidate solution and easing the communication across development teams, several issues need to be solved. In particular, one needs to translate models into an implementation code and thus, ensures that systems properties, requirements or constraints demonstrated at model level are correctly mapped or preserved. This step is usually a manual translation from the models to source code. This step is done based on the understanding of the engineer, but also with target limits (particular API, coding guidelines, etc.).

This implementation method is error-prone: source code correctness relies on the understanding of a software engineer and additional testing. Thus, implementation may contain bugs related to a misunderstanding of the specification or error due to the code itself. Furthermore, any change in the specifications (requirements modification or issues discovered when running the implementation code) may cause a significant code reengineering.

To address such issues, MBD toolchain now proposes mechanisms to generate implementation code from models. Existing tools can generate either application or architecture code (task and communication skeletons) from the system architecture, automating the implementation of each aspect of the system.

However, even if these approaches strengthen the overall development process, they do not provide any guarantee on system behavior correctness. In particular, it does not guarantee that system execution at code-level is equivalent to the model-level. As a consequence, validation of the system is still done manually, which remains error-prone and costly. Demonstrating such equivalence is a challenge for both the industrial and academic communities. As of today, only the SCADE tool chain [10] demonstrated this equivalence, but for a limited semantics based on the synchronous model of computation.

3. Approach

Early verification of real-time requirements of an application firstly requires the use of a modeling language that supports the same semantics as the actual run time system.
Concretely, it means that the modeling language must precisely define real-time concepts like:

- scheduling protocols (rate monotonic, deadline monotonic, highest priority first, …)
- thread dispatch protocols (periodic, sporadic, background, …)
- thread scheduling and execution states (suspended, ready, running, …)
- thread real-time properties (period, deadline, Worst Case Execution Time, …)
- inter-threads communication and synchronization protocols (events, shared data, subprogram calls, …)

The second step consists in developing new analysis tools or adapting existing ones, so that they can work on such models.

Finally, it must be ensured that the actual real-time properties that are in use at the model verification layer are compatible with those in place in the actual run-time system that is operating for the final application.

If all these constraints can be met, it becomes realistic to consider performing an early verification of real-time software at a model level. Such verification activities may consist for instance of schedulability analysis using analytical methods or less deterministic approaches like non-exhaustive simulation. It is then possible to compare execution traces obtained while running these analysis tools with the execution of the actual running software.

The Architecture Analysis and Design Language (AADL) [11] fits these requirements and tools such as Cheddar [6] for schedulability analysis and Marzhin [4] for non-exhaustive simulation can be used for early verification of AADL models. Combined with code generation strategies embedding trace generation, the designer can simulate its design, but also extract execution traces of its system and compare them. This is the approach (illustrated in *fig 1*) presented in this paper.

4. Supporting the process

We implement this approach in the TASTE tool-set [1,2], which is the outcome of the European project ASSERT [1]. TASTE aims at providing guidance to system developers by reducing manual development efforts and detects potential errors as early as possible during the design process. TASTE imposes a stringent design process, inherited from best practice mandated in space projects, from early functional designs, mapping of the functions to devices and finally code generation.

TASTE requires capturing application and system concerns using a single modelling language, AADL. It specifies both architecture (processor, buses, devices) and application concerns (functions to be called and their interaction) with respect to their properties (tasks period, events inter-arrival time, etc). The tool chain provides two tools that produce AADL models from a graphical representation: TASTE-IV (Interface View, description of functions) and TASTE-DV

(Deployment View, description of target hardware). These tools enforce separation of functional and deployment concerns, possibly performed by different teams. These AADL descriptions are then combined and processed by dedicated tools to both analyze requirements correctness and generate system implementation. Interface and deployment views are analyses and transformed into a Concurrency View (CV) that represents the combination of functions defined by the user and the set of threads and communication buffers required to run them on top of the embedded Operating System.

In that context, we tailored the TASTE tools to check requirements by simulating the system or by retrieving metrics during execution. Both outputs (simulation/validation reports and execution traces) are compared to check implementation correctness with respect to validation (figure 1).

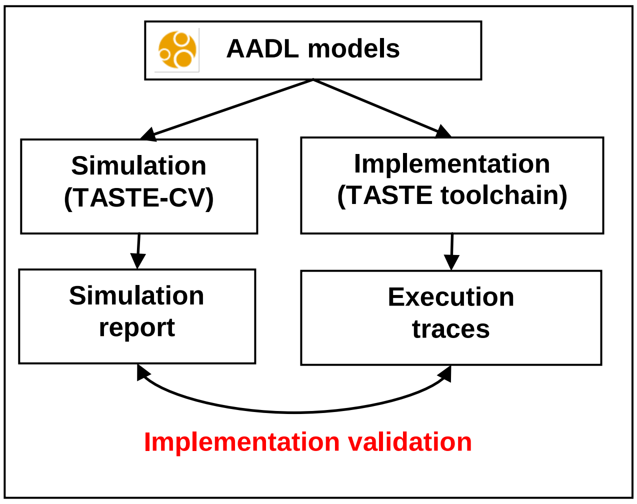


Figure 1- Application validation process

On the analysis part, the new TASTE-CV tool exploits the Concurrency View (CV) to support the validation of timing requirements through:

1. **Feasibility tests** that computes processor utilization factor by using application (period, deadline, etc.) and deployment (scheduling protocol to be used, etc.) information. This part of the process is supported by integrating Cheddar [6], a schedulability analysis tool developed in collaboration with Brest University.
2. **Simulation** that uses system description and produces expected behaviour at run-time. For that purpose, TASTE-CV integrates Marzhin, a multi-agents execution engine initially developed in collaboration by Virtualys [4] and Ellidiss [3] which has been tailored to support the AADL run-time semantics (threads state automata and communication protocols). While simulating system execution, Marzhin shows tasks activation as well as synchronizations between application components.

By using these two approaches, system designers may assess specifications feasibility (timing constraints to be met) and detect potential design errors without coding effort. Yet, such analysis may not be time accurate. Hence, we complemented these tools, and propose the validation of these traces with concrete implementation so that designers can check that execution is not only accurate from a functional point of view, but also accurate with respect to timing as seen on the target platform. This is done by an automatic instrumentation of generated applications.

The TASTE tool-chain produces implementation code from AADL description, generating glue code to interface with user-provided models or code, as well as tasks and communication buffers (see [7] for more details). We adapted the code generation process to add instrumentation instructions and produce metrics at run-time. Thus, generated applications output execution traces that report similar information as generated during the simulation (tasks activation time and data transfer across applications components). Since the binary will be executed on real hardware, the timing of events will be more precise.

To avoid any impact between instrumentation instructions and system metrics, integration of measurement instructions has to be as lightweight as possible. For this reason, instrumented applications store metrics in buffers and flush them once finished so that it avoids any I/O while operating. For the context of this work, this measurement capability was added on Linux targets.

To ease processing of traces, metrics are stored in files using the VCD file format that can be processed by open-source tools (such as GTKWave [5]) to produce a graphical output. The VCD format (Value Change Dump) is a widely used format from the Electronic domain. It stores traces from simulation of electronic circuit. It defines the chronogram of events and activation of elements. We used this format to preserve adaptability of the toolchain. An example is provided in figure 2, it shows activation time of two tasks.

By producing the same metrics, designers can check system correctness and validate implementation against validated specification either at model-level or code-level depending on the maturity of its design.

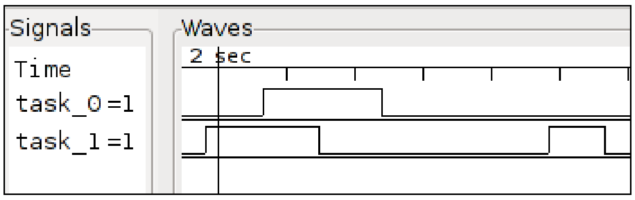


Figure 2- Graphical representation of VCD file using GTKWave

5. Case-Study

The following example is based on a basic producer/consumer application distributed on two nodes executing a regular linux kernel and connected with an Ethernet bus. Application aspects are designed using the TASTE-IV graphical tool, which avoid to write AADL textual specifications by hand. Application architecture, shown in *figure 3*, defines two functions:

- The *caller* function that contains a cyclic interface activator which purpose is to send data to the other.
- The *callee* function with a sporadic interface *printval* that receives a value and print it on the standard output.

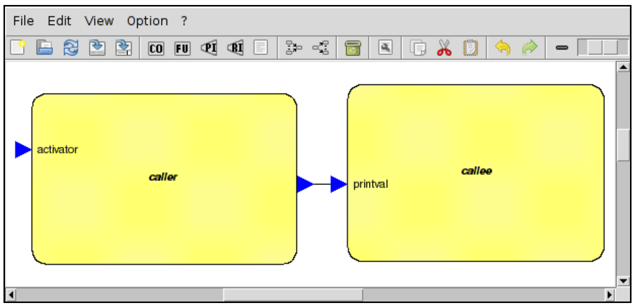
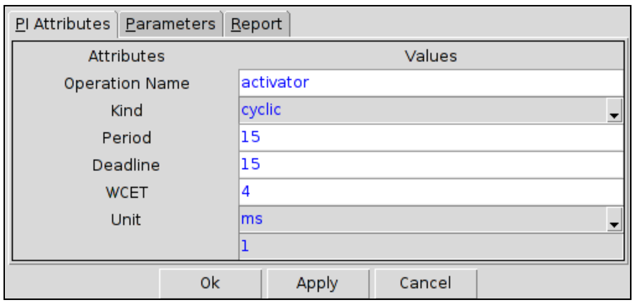


Figure 3- Functions and Interfaces of our case-study

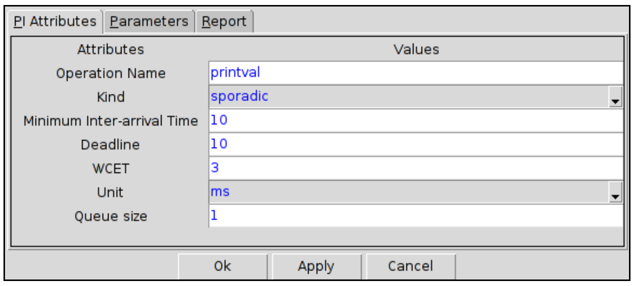
Timing requirements of functions interfaces are shown in *figure 4* and *figure 5*:

- The *activator* cyclic interface (*figure 4*) has a period of 15 ms and a Worst-Case Execution Time (WCET) of 5 ms.
- The *printval* sporadic interface (*figure 5*) has a Minimum Inter-Arrival Time (MIAT) of 10ms. It means that the task has a maximum receiving rate and is not triggered at each receiving data instance. In addition, it has a WCET of 3 ms, so that when activated, it consumes at most 3 ms of processor time.



Attributes	Values
Operation Name	activator
Kind	cyclic
Period	15
Deadline	15
WCET	4
Unit	ms
	1

Figure 4 - Timing requirements of the cyclic interface *activator*



Attributes	Values
Operation Name	printval
Kind	sporadic
Minimum Inter-arrival Time	10
Deadline	10
WCET	3
Unit	ms
Queue size	1

Figure 5- Timing Requirements of the sporadic interface *printval*

Then, the designer specifies the execution environment and functions allocation on processing resources. To do so, the TASTE-DV tool provides a convenient graphical tool to capture these aspects.

The resulting deployment model of our example is illustrated in *figure 6*. The platform contains two nodes connected through a standard Ethernet bus and executing a regular Linux operating system. Each processor is associated with one function so that the *sender* function is allocated to the first processor while the *receiver* is bound to the other.

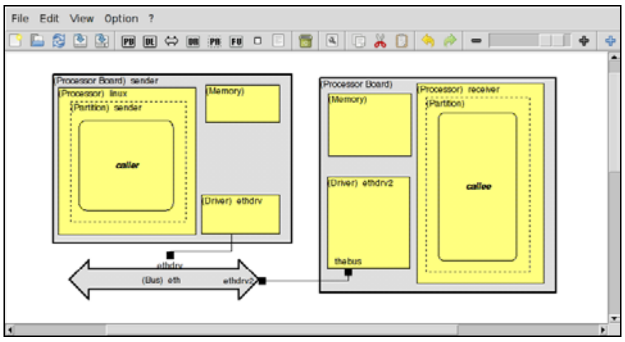


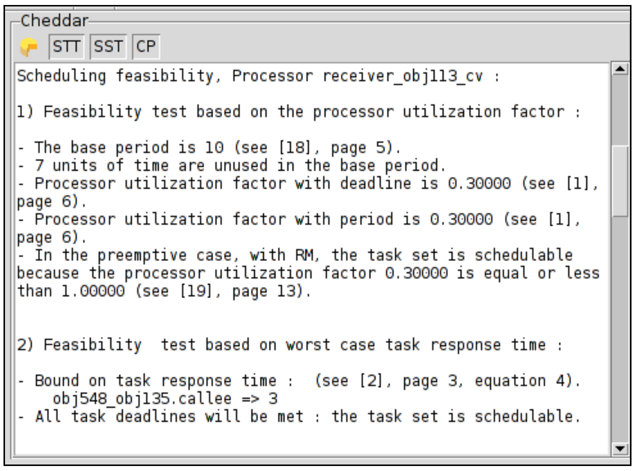
Figure 6 - Deployment of application functions

Once application and systems concerns are both specified, models are processed by validation tools to check timing requirements.

Beforehand, it is important to notice that from an implementation point of view, this application description is mapped into execution entities supported by the target operating system (threads, mutexes, etc.). In our process, this is done by a dedicated step called the Vertical Transformation that refines the abstract application description (Interface and Deployment views) into a model specifying execution components (the Concurrency View – CV – that specifies tasks, shared variables, locking mechanisms, etc. to be instantiated by the execution). Even if the description of this process is not the topic of this article, it is important to understand that our case-study is finally transformed into an application with two tasks scheduled using the traditional FIFO within Priorities scheduling policy available in POSIX-compliant execution platforms:

- One periodic task corresponding to the *activator* interface of the *sender* function. It has a period of 15 ms and a WCET of 5 ms.
- One sporadic task corresponding to the *receive_int* interface of the *receiver* function. It is triggered when receiving incoming data and is put to the idle state at least 10ms between two occurrences of a new incoming data. Its WCET value is 3 ms.

Figure 7 illustrates the scheduling validation using TASTE-CV and Cheddar [6]. It performs feasibility tests by processing tasks description with state-of-the-art algorithms validation algorithms, such as RMA. Thus, it computes processors utilization factor and evaluates timing requirements (tasks deadlines) enforcement. Using this tool on our specification reports that timing constraints are met with a processor utilization factor of 0.3 using the RM analysis.

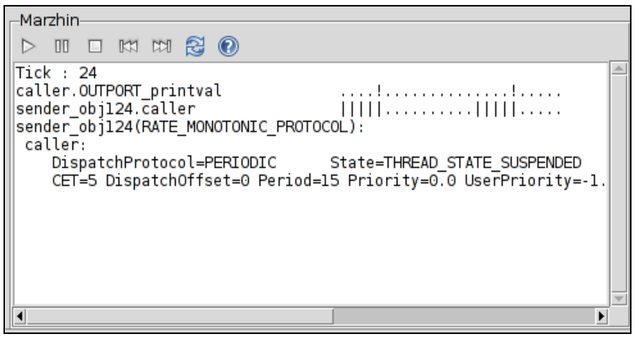


```
Scheduling feasibility, Processor receiver_obj113_cv :

1) Feasibility test based on the processor utilization factor :

- The base period is 10 (see [18], page 5).
- 7 units of time are unused in the base period.
- Processor utilization factor with deadline is 0.30000 (see [1],
page 6).
- Processor utilization factor with period is 0.30000 (see [1],
page 6).
- In the preemptive case, with RM, the task set is schedulable
because the processor utilization factor 0.30000 is equal or less
than 1.00000 (see [19], page 13).


2) Feasibility  test based on worst case task response time :

- Bound on task response time :  (see [2], page 3, equation 4).
    obj548_obj135.callee => 3
- All task deadlines will be met : the task set is schedulable.
```

Figure 7 - Scheduling validation with feasibility tests

Figure 8 illustrates the simulation of the *sender* node using TASTE-CV and the Marzhin [4] multi-agent tool. The tool reports the execution events occurrences of the tasks as well as produced data instance by considering the worst case scenario. In the following example (as shown in *figure 8*), we notice that the simulated behaviour of the sender task is compliant with the specification:

- It is activated on a 15ms basis (task period)
- At each activation, it consumes 5 ms of the processing resource (task WCET)
- Once executed, one data item is produced



```
Tick : 24
caller.OUTPORT_printval                ....!...............!.....
sender_obj124.caller                   |||||..........|||||.....
sender_obj124(RATE_MONOTONIC_PROTOCOL):
 caller:
    DispatchProtocol=PERIODIC      State=THREAD_STATE_SUSPENDED
    CET=5 DispatchOffset=0 Period=15 Priority=0.0 UserPriority=-1.
```

Figure 8 - Scheduling validation using Simulation

However, results issued from specifications have to be compared with execution behaviour. For that purpose, we tailored the TASTE code generators and add instructions to report execution events at run-time. As the objective is to check application behaviour correctness (without considering real-time concerns of the execution platform), our experiments were done on a regular Linux distribution for convenient purposes (easy file management, etc.). However, it can also be implemented on top of real-time specific run-time such as RTEMS or Linux/Xenomai but would require specific setup (such NFS mounts on RTEMS to store traces, etc.)

Figure 9 and *figure 10* depict these activation events for the sender task; the one analyzed previously using the specifications.

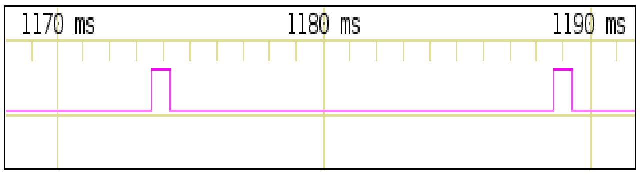


Figure 9 – Activation events of the sender task at run-time (1)

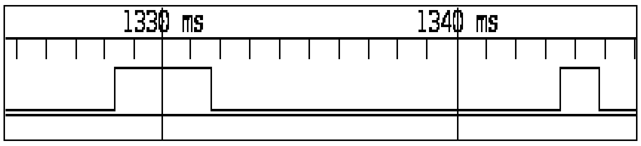


Figure 10 - Activation events of the sender task at run-time (2)

From these metrics, we can check timing requirements correctness:

- The activation period is enforced, the task is triggered each 15ms
- The WCET (5ms) complies with the specifications; the task is never executed longer than 5ms. However, this execution time is smaller than during simulation because the former reports the real execution time on the target while the latter always consider the worst case scenario.

6. Conclusion & Perspectives

The design and implementation of complex real-time embedded systems requires multiple steps, from early requirements elicitation down to final coding. To ease this process, the TASTE toolchain promotes extensive code generation from well-defined semantics abstractions based on the AADL.

One difficulty in model-based engineering is to understand precisely the behaviour of the system under consideration. Simulation or virtual execution of the system should be available early in the design flow to increase confidence in the models.

In this paper, we demonstrated how simulation at model-level and execution at binary-level can be combined to have a first level of analysis on early designs and then complete the understanding of the actual system behavior through execution of the system. Simulation capabilities are provided through a multi-agent system. Run-time traces are automatically integrated to generated code, guaranteeing both accuracy and limited run-time overhead. We evaluated this approach on a case study, demonstrating increased confidence in the system being built.

7. References

[1] Project website: www.assert-project.net

[2] Tools download: http://download.tuxfamily.org/taste

[3] Ellidiss (TASTE toolsuite): www.ellidiss.com

[4] Virtualys (Marzhin): http://www.virtualys.com

[5] GTKWave: http://gtkwave.sourceforge.net

[6] *Cheddar: a Flexible Real Time Scheduling Framework*. F. Singhoff, J. Legrand, L. Nana and L. Marcé. ACM SIGAda Ada Letters, volume 24, number 4, pages 1-8. ACM Press, New York, USA. December 2004, ISSN:1094-3641. Also published in the proceedings of the international ACM SIGAda conference, Atlanta, Georgia, USA.

[7] Conquet, Eric and Perrotin, Maxime and Dissaux, Pierre and Tsiodras, Thanassis and Hugues, Jérôme *The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software.* (2010) In: European Congress on Embedded Real-Time Software (ERTS 2010), 19-21 May 2010, Toulouse, France.

[8] Sanford Friedentahl, Alan Moore and Rick Steiner. A Practial Guide to SysML: The Systems Modelling Language. 2008. Morgan Kaufman Publishers Inc.

[9] UML Profile for Modeling and Analysis of Real Time and Embedded Systems (MARTE). OMG standard.

[10] Parosh Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren, Ove Åkerlund, Tiziana Margaria and Bernhard Steffen. ”Designing Safe, Reliable Systems Using Scade”. In Leveraging Applications of Formal Methods. Lecture Notes in Computer Science. 2006. Springer Berlin / Heidelberg. ISBN: 978-3-540-48928-3. DOI: 10.1007/11925040_8

[11] Architecture Analysis & Design Language (AADL). As-2 Embedded Computing Systems Committee SAE. SAE standard,

8. Glossary

AADL: Architecture Analysis & Design Language

I/O : Input/Output

MIAT: Minimal Inter-Arrival Time

RMA: Rate Monotonic Analysis

RMS: Rate Monotonic Scheduling

TASTE: The Assert Set of Tools for Engineering

VCD: Value Change Dump

WCET: Worst Case Execution Time