On integration of open-source tools for system validation, example with the TASTE tool-chain

Julien Delange and Maxime Perrotin
European Space Agency
ESAESTEC, TEC-SWE
Keplerlaan 1
2200AG Noordwijk, The Netherlands
julien.delange@esa.int, maxime.perrotin@esa.int

Abstract

Design and implementation of safety-critical system is very difficult because they must ensure continuous correct operational state whereas they are deployed in hostile environments. An error either during design or implementation phases can have significant impacts and consequences. To avoid such issues, failure cases must be clearly identified and handled by software engineers to prevent any propagation from one faulty component to another. For that purpose, good practices and standards are applied during the development process, from the specifications to the implementation.

However, despite all existing efforts, bugs are still introduced. They are introduced at different levels of the development process: either in the specifications (as in the Mars Climate Orbiter mission - failure was due to a mix-up of metric units) or in the implementation (as in the Ariane 5 launch - wrong assumption was made about a data type so that the system generate an overflow).

Over the years, several solutions have been designed to address such issues. However, they rely on different system representations and are applicable at different level of the design process, so that their use could be difficult and may lead to design inconsistencies. In consequence, we have to avoid these problems and make their use more consistent.

In this paper, we present our tool-chain for system design, validation, implementation and certification. It relies on a modeling notation to capture both software and hardware concerns. The use of a single notation ensures specification consistency and avoiding potential errors when using different language to specify the same system aspect. We detail the support of this process in The Assert Set of Tools for Engineering (TASTE) development tool-chain.

1 Introduction

Safety-critical systems design and implementation is very difficult: they must operate correctly and continuously whereas they are usually deployed in hostile environments. Misconception or errors may have significant consequences and are potentially mission or life critical. To avoid such issues, failure cases must be clearly identified and handled by software engineers to avoid any propagation from one faulty component to another. For that purpose, several guidelines and standards have been designed and are currently used during all system development phases.

Safety-critical systems development is usually splitted in several phases, as illustrated in figure 1.

1. Validation: from specifications, engineers/developers check system feasibility and requirements enforcement

2. Code production: developers implements the system by translating the specifications into code (Ada/C) that can be compiled.

3. Certification: implementation execution is validated against its specifications and/or established standards (DO178B, ECSS, etc).

Even with a such careful process, errors remain either in the specifications (as in the Mars Climate Orbiter mission - failure was due to a mix-up of metric units) or in the implementation (as in the Ariane 5 launch - wrong assumption was made about a data type).
type so that the system generate an overflow). In particular, errors may be introduced at each step:

1. **Validation**: because it often rely on different system representations, a requirement validated using one notation may not be validated using another one.

2. **Code production**: developers often introduce errors or bugs either by misunderstanding system specifications or just by making syntax or semantic errors.

3. **Certification**: this also relies on a manual process where engineers can make errors by misunderstanding specifications.

![System specifications diagram]

**FIGURE 1**: Generic development process work-flow

To cope with these problems, the development process must be more consistent and automated as much as possible. It would automatically check requirements from specifications and ensure standards compliance enforcement at lower levels. As a result, this would also reduce development cost (system is verified by appropriate tools), ensure the development process reliability and robustness due to the automation of each step.

Next section details in details identified problems and presents our approach to address them.

## 2 Problem & approach

### 2.1 Problem statement

Despite all existing initiatives, errors/bugs are still introduced during development. To reduce them as much as possible, one has to:

1. **check and validate specifications automatically**

2. **verify implementation correctness regarding specifications**

Different tools already address these issues. However, they are loosely coupled and rely on a different notations that lead to potential semantic issues so that:

1. a requirement R1 validated using the specification language L1 and the validation tool T1 can be not validated when using the specification language L2 and the validation tool T2.

2. even if the user manages to translate system specification from one language to another, the process will require a manual translation which is error-prone: the user can introduce specification errors by himself (syntax error, misunderstanding of system specifications, etc.).

As a result, there is a strong need for a more consistent approach that strengthen system development with:

1. **A single notation** for all system aspects so that we avoid several representations of the same concepts and so, prevent any specification inconsistency.

2. **An automation of development steps** with tools that process specifications and produced development output without human guidance.

### 2.2 Proposed approach

First, we propose to capture system architecture with its requirements, properties and constraints using a single modeling language. It would specify both software and hardware concerns with a unique notation, avoiding all usual semantic issues.

From this high-level representation (models), development steps (same as in figure 1) are automatically processed by appropriate tools. In particular:
1. **Validation**: tools automatically process models to check system correctness and feasibility so that designers can fix specifications errors before further development efforts.

2. **Code Production**: code generators transform specifications/models into implementation code (such as Ada or C). This generated code is automatically compiled and linked against a Real-Time execution platform that supports system entities (tasks, mutexes, etc.).

3. **Certification**: implementation is either simulated or executed on the target to check its behavior correctness and standards (such as DO178B or ECSS) compliance.

We implement this process in The ASSERT Set of Tools for Engineering (TASTE) by using AADL as a single specification language. We also design tools that process AADL models and support each development step. Next sections details the both language, its tailoring for our needs and the specific tools we develop in that context.

### 3 Overview of TASTE

The Assert Set Of Tools for Engineering (TASTE) is the outcomes of the European ASSERT project [2]. It [1] aims at providing a complete functional tool-chain for safety-critical system development, from the specifications to the certification/validation.

It relies on the Architecture Analysis and Design Language (AADL) [9] to represent both software and hardware concerns, their properties and constraints. First, software aspects are specified by the Data View and the Interface View, two AADL models that represent system functions (C/Ada code) and the data they share (using ASN.1 [19], a standardized notation for describing data types). Hardware and deployment concerns are described with the Deployment View, an AADL model that describes the execution platform (processors, devices, memories, etc.) and its association with system functions.

From these models, our tool-chain automates the development, as illustrated in figure 2:

1. **Validation**: it checks specification correctness by processing models and using appropriate tools:
   - (a) Cheddar [7] or MAST [8], two scheduling analysis tools released under free software
   - (b) REAL [15], an AADL [9] query tool integrated in Ocarina [10] that checks specification correctness

2. **Code production**: it transforms AADL models into C code using Ocarina [10], an AADL tool-suite released under GPL licensing terms. In particular, Ocarina is able to automatically generate code that targets real-time embedded platforms (RTEMS [12], VxWorks [13], etc.) and standards (RT-POSIX [14]).

   The code is then integrated on top of a real-time execution platform: Ocarina [10] actually supports the following free-licensed platforms: RTEMS [12], Xenomai [11] and POK [18].

3. **Certification**: it executes the code either on the target or a simulator (such as QEMU), checks:
   - its performance (using gprof, a performance analysis tools included in the GNU binutils suite [3])
• reproduces its behavior (by instrumenting the code and produce a Value Change Dump (VCD) [6] file to be used with GTKWave [5])
• produces code coverage reports using the COUVERTURE tool-set [4] (specific freer licensed tools from Adacore that aims at supporting code coverage using a specific tailored version of QEMU [16]).

The use of a single notation (AADL [9]), processed by dedicated tools for each development aspect makes the overall process more consistent. In addition, automation of model processing avoids issues of usual development process and ensures requirements traceability. Finally, while system feasibility and requirements are automatically checked during the development process, these tools also provides metrics (such as code coverage) that can be used for system certification.

Next sections focus on validation and certification functions of our tool-chain:

• Section 4 describes our system validation functions from using AADL specifications.
• Section 5 details the automatic certification process with respect to the implementation.

4 Model Analysis & Validation

The TASTE tool-chain relies on AADL [9] to specify software (Data View and Interface View) and hardware (Deployment View) aspects. AADL is a component-based language to specify hardware and software concerns with their execution constraints. It has all necessary constructs to express safety-critical system concerns and supports mechanisms to address specific modeling needs. They can be written using either a textual or a graphical notation and is supported by a large tool-set, from command-line interface tools (such as Ocarina [10]) to tools with advanced graphical interface (like OSATE [9]).

As this article does not aim at providing a full overview of AADL, readers that would like to learn more about it can refer to the introduction written by its designers [20].

4.1 AADL modeling benefits

By introducing a single specification notation for both software and hardware concerns, we strengthen the overall development process, making it more consistent. Extensions mechanisms allow us to tailor the language to our needs:

• Properties extension mechanism is used to define specific requirements from textual specifications to the AADL model (for example, to model memory concerns such as stack or heap size, etc.).
• Annex languages mechanism is used to associate our in-house AADL validation tool (REAL) to check requirements enforcement. It processes processes models according to its components hierarchy and check for system requirements validation (for example: can a process P1 with 1Mb of RAM contain three threads that require a stack of 800Kb ?).

If several modeling languages already exist for the specification of real-time embedded systems, no one provides the ability to capture both hardware and software aspects with such a flexibility. That is why our choice was focused on this language.

4.2 REAL validation tool-set

REAL (Requirements Enforcement Analysis Language) [15] is a language that associates validation theorems to AADL components. A dedicated solver analyzed model components with their theorems and check their enforcement.

To use REAL, users have to:

• Map properties and constraints from the textual specification to the AADL model (for example, execution time for each system function, period/deadline of each task, etc.).
• Design theorem to check requirements feasibility (for example: functions can be executed within task period).

One key aspect is the genericity of this approach: users can keep existing theorems in a library that would be reused for later projects.

Listings 1 and 2 give an example of the definition of a REAL theorem and its application on an AADL model. Listing 1 defines a (incomplete, due to lack of space) model with one main system containing:

• One process component with two tasks (thread components). The first one requires 35Kbytes of memory and the other 57Kbytes;
• One memory component with 40000 bytes.
process implementation p.i
subcomponents
  task1 : thread t.i
  { Source_S Stack_Size => 10 Kbytes;
    Source_Data_Size => 20 Kbytes;
    Source_Code_Size => 5Kbytes; }
  task2 : thread t.i
  { Source_S Stack_Size => 2 Kbytes;
    Source_Data_Size => 50 Kbytes;
    Source_Code_Size => 5Kbytes; }
end p.i;

system implementation s.i
subcomponents
  mem : memory ram .i
  { Word_Count => 10000;
    Word_Size => 4 bytes; }
  prs : process p.i;
properties
  Actual_Memory_Binding =>
  { (reference (mem)) applies to prs; }
end s.i;

Listing 1: AADL model example to be processed by the REAL validator

The REAL theorem (listing 2) checks that for each AADL process component of the model in listing 1 the amount of memory required by its tasks (lines 10 to 12 of listing 2) is less than the size of its associated memory (lines 14 and 15 of listing 2). Regarding the model of listing 1, this theorem is not be validated and validation tool would issue an error.

theorem check_memory
  foreach prs in process_set do
    t := { x in Thread_Set | is_subcomponent_of (x, prs) }
    m := { x in Memory_Set | is_bound_to (Prs, x) }
    check ( 
      (sum (property (t, "Source_S Stack_Size"))) + 
      (sum (property (t, "Source_Data_Size"))) + 
      (sum (property (t, "Source_Code_Size")))) < 
      (sum (property (m, "word_count"))) * 
      (property (m, "word_size")))
  end check_memory;

Listing 2: REAL theorem that checks task memory requirements

4.3 Scheduling validation

Scheduling is a very intensive topic in the context of embedded and real-time systems. Numerous scheduling analysis techniques and methods have been designed over the years trying to evaluate system scheduling feasibility.

TASTE interfaces AADL specifications with two scheduling analysis tools: Cheddar [7] and MAST [8]. Both are available under the GPL license terms. Next sections give an overview of these tools and explain how AADL models are exported to them.

Overview of Cheddar and MAST

Cheddar [7] is a scheduling analysis tool written in Ada that provides command-line as well as graphical interface (shown in figure 3). It validates timing constraints either by simulating system execution or performing feasibility tests. To do so, the user must describe system architecture (processor, task, scheduling policy, etc.). Cheddar supports state-of-the-art scheduling algorithms (RMS, EDF, LLF), as well as standardized algorithms (like the one available in POSIX 1003b). Cheddar analysis also takes inter-tasks dependencies into account with an analysis of different sharing methods such as PIP, PCP or IPCP.

FIGURE 3: Cheddar scheduling validation

As for Cheddar, MAST [8] is a tool (shown in figure 4) that aims at validating scheduling feasibility of a system. It can analyzes system using several algorithms either for task scheduling (RMS, EDF, etc.) or data locking (PIP, PCP, etc.). On the other hand, MAST takes into account distributed systems concerns, which is especially critical for real-time systems. For example, when a task execution is triggered an incoming data from another task, analysis has to take into account scheduling concerns since a delay on the sender side would have an impact. In addition, network-related aspects (such as latency, jitter, etc.) may also impact system execution as a whole. A system specification in MAST takes these
aspects into account, offering a convenient way to analyze distributed systems.

However MAST and Cheddar rely on their own specification language. Consequently, engineers have to translate system specifications into a new representation dedicated to scheduling analysis. This mapping is error-prone: engineers can make a mistake so evaluation would be done using wrong assumptions. For that reason, TASTE automates this translation from AADL to a specific representation that would be used by either Cheddar or MAST, as detailed in the next section.

5 Implementation analysis

TASTE automatically creates system implementation from its Interface View, Deployment View, and Data View (AADL models) by generating code that targets Real-Time operating systems. However, even if this automatic process offers many benefits (error avoidance, requirements enforcement & traceability, etc.), system implementation still has to be validated and also met certification requirements.

5.1 Performance analysis

Once system implementation is generated, developers can deploy it on the execution target. Then, appropriate tools trace/monitor system behavior to evaluate its performance. For that purpose, several tools already exist and are released either under proprietary or free-software license.

To assess generated application performance, TASTE uses gprof, an execution profiling program available in the binutils tool-set. Its main advantage is its integration within the GNU compilation tool-chain: just by adding a flag in the compilation options enable application profiling that can be later processed by analysis tools.
TASTE provides its own interface method with `gprof`, as illustrated in figure 6. It parses profiling results by its own and produces an execution report with the execution time and the number of execution for each function. By using this report, engineers check execution traces compliance with system requirements.

### 5.2 Specifications compliance enforcement

Execution profiling provides metrics and data that could detect some erroneous execution case (a function called too many times, a call that would not happen, etc.), but may be not sufficient to check implementation correctness. In particular, implementation validation requires to check implementation consistency with the specifications (AADL models). This consist in monitoring system events, and check their compliance with the model. For that purpose, TASTE provides functions to monitor system events at run-time and create appropriate metrics that can be compared with its specifications. To do so, it instruments generated application with profiling instructions that produces VCD files at run-time (example of events reported is available in figure 7) with the following metrics:

- Task activation time
- Data sent/received through tasks port
- Shared data usage (semaphore/mutex acquisition and release)

Once produced, programs such as GTKWave [5] as used to depict system events with a graphical interface and provide the ability to analyze system behavior. It offers the ability to check run-time behavior consistency with system specifications (for example, the task activation time is correct regarding specified period and deadline).

### 5.3 Code coverage analysis

Standards such as DO178B [21] (for avionics systems) or ECSS [22] (for aerospace applications) requires that safety-critical systems enforces a predefined code coverage, depending on their criticality level.

To do so, different methods are commonly used, but most of the time, they require a manual instrumentation or inspection of application code. Code instrumentation is intrusive: the code under inspection is not the one that would be deployed and so, validation results may not be relevant. In addition, a manual inspection is still error-prone, due to the human-factor errors.

To cope with these issues and provide an accurate coverage analysis, TASTE relies on the COVERAGE tool-set [4], a code coverage analyzer released under free-software license terms. It relies on two main tools that produce coverage reports (as shown in figure 8):

1. A tailored version of QEMU [16] traces all executed instructions when executing the system.
2. An analysis tool, `xcov`, compares executed instructions with the program under execution and produces a coverage analysis report.

By tracking executed instructions and establishing a mapping with the source file, `xcov` produces a complete coverage report, as shown in figure 8. It
details the execution of each line of code so that developers are able to assess if some block could be removed or not.

However, to evaluate system implementation, this coverage functionality would be integrated with a test framework that would execute generated applications with different input values that are representative of a real environment. This would provide a better assessment of system quality, force each condition/decision of the code to be executed and lead to a better coverage analysis.

One particular interest of the COUVERTURE tool-set is its non-intrusive characteristic: coverage analysis is representative of the quality of applications that are finally deployed. In addition, COUVERTURE supports several coverage methods that required by certification standards (Statement Coverage - SC, Decision Coverage - DC or Modified Condition Decision Coverage - MCDC). Using these different methods, we could evaluate and potentially certify generated systems at different levels.

6 Case study

The following sections illustrate the use of our TASTE tool-chain through a case-study that deploys several functions into a heterogeneous and distributed architecture.

6.1 Overview

This case study consists in simulating a temperature sensor with a basic forecast management system. It is composed of three functions:

1. A sensor for temperature acquisition.
2. A filter for bad data detection.
3. An average computer that receives each new temperature value from the filter and prints the average temperature.

Each function is deployed on top a Real-Time Operating System, executed on a single processor:

- The sensor function is deployed on a LEON2 processor with RTEMS.
- The filter function is executed on an Intel i386 processor with a Linux operating system.
- The average function is deployed on a LEON2 processor with RTEMS.

Finally, to enable the communication between functions sensor/filter and and filter/average, the processors are connected using a SpaceWire bus, as shown in figure 11.
it filters the data and send it when it is considered as valid. For the needs of this simulation, 50% of received data are considered as correct so that this function sends data to the average function every two seconds.

- The **average** function is also sporadic and activated when receiving incoming data from the filter. As the filter function sends data every two seconds, this function execution follows this period.

**FIGURE 11:** Deployment view of the system

Then, once these functional aspects have been described, their allocation on the platform has to be specified using the **Deployment View**. It defines the hardware to be used and its association with the previously defined functions of the **Interface View**. The deployment view of the case study (illustrated in figure 11) is composed of three processors (two LEON using RTEMS - acq_board and avg_board and one intel i386 that uses Linux - filter_board) interconnected using SpaceWire links.

**6.3 Implementation validation**

Then, TASTE processes system specifications (**Interface view** and **Deployment view**), to generate binaries that would be executed on the target. Then, it analyzes the implementation and verifies its compliance with the specifications in order to check certification requirements enforcement.

First of all, system profiling is performed using **gprof**, as illustrated in figure 13. Profiling report shows how many time each function has been executed. In the following example, the results refer to the execution of the first ten seconds of the filter function. We can see that the _pohi_delay_until() function (called by each function at the end of each cycle) has been invoked almost 30 times, which can seem to be inconsistent: this function is triggered each second so that function would be activated at most 10 times. However, the node that hosts it communicates using two SpaceWire bus and each one is using a task that polls the bus for incoming data periodically each second. Consequently, if the functional aspects generates only one task in the system, the deployment concerns (device drivers) add additional resources that have an impact on system execution.

**FIGURE 12:** Scheduling analysis result for the node acquisition

Finally, from these both models, we can validate some of its aspects prior to implementation efforts. In this case-study, we run a schedulability feasibility using Cheddar 7, as illustrated in figure 11. The scheduling feasibility test is based on simulation and is performed for each processor of the system (figure 11 illustrates the result for the acq_board).

**FIGURE 13:** Profiling of the filter node using gprof

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Execution number</th>
</tr>
</thead>
<tbody>
<tr>
<td>_po_hi_marshall_array</td>
<td>51</td>
</tr>
<tr>
<td>_po_hi_unmarshall_part</td>
<td>51</td>
</tr>
<tr>
<td>_po_hi_microseconds</td>
<td>50</td>
</tr>
<tr>
<td>main</td>
<td>47</td>
</tr>
<tr>
<td>_po_hi_c_driver_spw_usb_brick_sender</td>
<td>46</td>
</tr>
<tr>
<td>_po_hi_unmarshall Uint16</td>
<td>45</td>
</tr>
<tr>
<td>_mod03</td>
<td>44</td>
</tr>
<tr>
<td>_po_hi_q�数_get_value</td>
<td>44</td>
</tr>
<tr>
<td>BitStream_DecodeConstraintWholeNumber</td>
<td>43</td>
</tr>
<tr>
<td>_po_hi_task_delay_until</td>
<td>43</td>
</tr>
<tr>
<td>_po_hi_unmarshall Uint8</td>
<td>42</td>
</tr>
<tr>
<td>_po_hi_delay_until</td>
<td>42</td>
</tr>
<tr>
<td>_po_hi_marshall_int32</td>
<td>31</td>
</tr>
<tr>
<td>filter_job</td>
<td>31</td>
</tr>
</tbody>
</table>
Then, as detailed in section 5.1, TASTE analyzes system implementation to check system behavior compliance with its specifications. Figures 14 and 15 report run-time events that occur when executing of the implementation of the node filter_board. Figure 14 reports the events at a coarse grain: we can see that system activity happens on a periodic basis, each second. Then, figure 15 details the events at a finer grain, each second:

1. The poller task of the SpaceWire driver (task_0 on figure 15) is activated. It receives incoming data from the acq_board (that acquires temperature from the sensors).
2. When receiving data, poller task from the SpaceWire driver (task_0 on fig 15) transfers data to system port (port_2_0 on fig 15).
3. New data instance triggers the execution of the filter function and its associated task (task_2 on figure 15) which retrieves the data (so that the size of the port port_2_0 fallbacks to 0), executes its code and waits for new data.

We can see that these events are consistent with system specifications: the SpaceWire driver receives data each second and triggers the execution of the sporadic function filter. However, we didn’t detail the execution of the task task_1 which seems always active. In fact, this task corresponds to the poller function connected to the other SpaceWire bus. As it never receives data (this node only sends data through this bus), the associated task is always waiting for incoming data and never go to the sleep mode.

Reports can be produced in different formats (text, HTML, etc.). They detail, for each file and function, the coverage information, so that engineers assess system quality based on execution metrics. In our case-study, most functions of the RTEMS executive are not used so that it significantly decreases the code coverage level of produced applications.

7 Conclusions & Perspectives

This article gives an overview of open-source tools that provide help and guidance for safety-critical systems design. They are used as early as possible to support each step of the development process. Such tools are usually loosely-coupled and require manual efforts to be tailored to the development process of each system. To cope with these issues, TASTE makes their use more consistent by linking them with a single specification notation.

For that purpose, AADL models describes system architecture with its execution constraints using. Then, tools translates this specification notation to be processed by validation programs that check architecture correctness and requirements enforcement. This process automates the process, avoiding issues of usual development methods.
Use of such a tool-chain strengthens the development process and makes it more robust and reliable. Moreover, as potential errors are discovered early in the development process and integration issues would likely be reduced, development cost are expected to decrease significantly.

Further work would cover other aspects of safety-critical systems development. In particular, our tool-chain could also support additional guidance for safety-critical standards (such as DO178 or ECSS) enforcement by providing documentation generation facilities or additional implementation code validation (coding rules to be checked, etc.).

7.1 Perspectives

Automation of AADL models production from usual text-based specifications or linking these two notation would be particularly useful and integrates our tool-chain with traditional design methods. Such a translation process will require that:

- all system entity and its associated requirement from the initial specifications are correctly translated into AADL components
- there is no specification inconsistency (due to semantic issues, mapping error, etc.) between the AADL model and the initial specifications.

Tools that address these issues are currently being developed. For example, the TOPCASED [23] requirements importer tool provides the ability to connect a requirement from a textual document (with an extension such as .odt, .pdf, .txt, etc.) to a model (potentially AADL). This work is emerging and available tools are still considered as experimental, but this topic is a particular interest and would be a major interest to trace a requirement from its description (in the textual document) to the implementation (the code).

Then, another idea is to strengthen our tool-chain by improving its functions. In fact, some TASTE functions are limited to several architectures or platforms (for example, system analysis that produces VCD files is limited to Linux platforms). An important improvement would consist in supporting all potential deployment platforms.

References

[1] The ASSERT Set of Tools for Engineering
   http://www.assert-project.net/taste
[2] The ASSERT project
   http://www.assert-project.net
[3] GNU binutils
   http://www.gnu.org/software/binutils/
[4] The Couverture project
[5] GTKWave
   http://gtkwave.sourceforge.net/
[6] Value Change Dump file format
[7] Cheddar scheduling analyzer
[8] MAST scheduling analyzer
   http://mast.unican.es/
   http://www.aadl.info
[10] Ocarina AADL Toolset
    http://ocarina.enst.fr
[14] POSIX 1003.1b
[15] Olivier Gilles and Jérôme Hugues - Validating requirements at model-level in Ingénierie Dirige par les modèles (IDM08)
[18] Partitioned Operating Kernel - POK
    http://pok.safety-critical.net
[19] Gerald W. Neufeld and Son Vuong - Overview of ASN1 in NetISDN, 1992
[22] ECSS-E-40, Space Engineering Software