TASTE: An open-source tool-chain for embedded system and software development

Maxime Perrotin, Eric Conquet, Julien Delange, Thanassis Tsiodras

TASTE:
An open-source tool-chain for embedded system and software development

Maxime Perrotin[1], Eric Conquet[1], Julien Delange[1,] Thanassis Tsiodras[2]

[1] European Space Agency, ESTEC, Keplerlaan 1, 2201AG Noordwijk, The Netherlands
{Maxime.Perrotin, Eric.Conquet, Julien.Delange}@esa.int,
[2] Semantix, K.Tsaldari 62, Poligono 114 76, Athens, Greece, ttsiodras@semantix.gr

Abstract. This paper presents the results of the past two years of continuous development of the TASTE tool-set. TASTE is a development environment dedicated to embedded, real-time systems and was created under the initiative of the European Space Agency back in 2008, after the completion of a FP6 project called ASSERT. TASTE is free and open source, and is currently used to design small to medium-size systems; it relies on two powerful and complementary modeling languages: ASN.1 and AADL, and comes together with a solid engineering approach. TASTE brings ideas on how a system can be optimally built, by taking and putting together components of heterogeneous nature, and making sure that they run according to their specification and without software hacking introduced during the development. The latest TASTE improvements include several major features such as the integration of VHDL components, the recording/replay of runtime scenarii using message sequence charts, the support of legacy encoding protocols, means to inspect and patch data at runtime, and the generation of systems for additional platforms such as RT Linux (Xenomai). Our graphical front-ends have also been redesigned to offer a better user experience.

Keywords: ASN.1, SDL, MSC, TASTE, SCADE, AADL, VHDL

1 Introduction – what is TASTE

TASTE is a development environment and set of tools helping users to produce software for embedded systems in a quick and efficient way, and without introducing unnecessary complexity to what good engineers already master. It is relying on a set of pillars, which are in fact simple principles that are often derived from the so-called KISS (keep it simple, stupid) approach: reuse what works, do not reinvent the wheel, automate tedious tasks.

We think (and hope) that engineers and scientists like to innovate, solve challenging problems and learn new things more than repeating boring activities over and over again. Making software can be a creative and enjoyable activity for many of us – it can also easily become a nightmare to code, maintain and validate for other people and companies. TASTE is addressed to those who know what software is, and who are convinced that some aspects of the development process can be improved without giving up any of the reasons why we like to program. It is a concentrated set of technologies that follow the Unix way of doing computer science: dedicated tools solving specific issues, sharing knowledge, releasing often, making concrete things happen. As such, in addition to be a functional and fully operational tool-set, TASTE is also a laboratory platform for experimenting and improving technologies in various fields of embedded, real-time software development. Almost all components are available for free following open-source licensing schemes, which allows for new contributors to join the team and bring their own stones to the edifice.

The systems that TASTE typically addresses share at least some of the following characteristics:

- They have limited resources (memory, CPU);
- They have real-time constraints (deadlines) ;
- They contain applications of very different natures (control laws, resource management, protocols, failure detection);
- Parts of the system are developed by different companies;
- They communicate with hardware (sensors, actuators, FPGA);
- They contain heterogeneous hardware (e.g. CPUs with different endianness);
- They can be distributed over several physically independent platforms;
- They may run autonomously for years;
- They may not be physically accessible for maintenance (satellites)

TASTE helps putting all the system components together, such that they can be easily deployed on distributed targets. The tools generate code that makes communication and real-time aspects transparent: no need to spend time on formatting and decoding messages, creating threads and configuring the real-time operating system, debugging synchronization and resource sharing – all these tedious tasks are automated.

When components of heterogeneous nature are involved in a system, we think that it is relevant to develop each of them by applying a dedicated method that takes into account their own constraints and characteristics. In the case of software, we believe that letting the user develop in the language of his choice is the right way to go, may it be a programming language such as C (e.g. for drivers), Ada or Python, or a modeling language for state machines and control laws. Note that in that respect, TASTE follows a path that is different from most MBSE approaches that tend to promote a single language or environment to cover all software facets.

In fact, when we look deeper in the way systems are actually developed, the number of languages that need to be put together is rather impressive, as shown in the picture below:

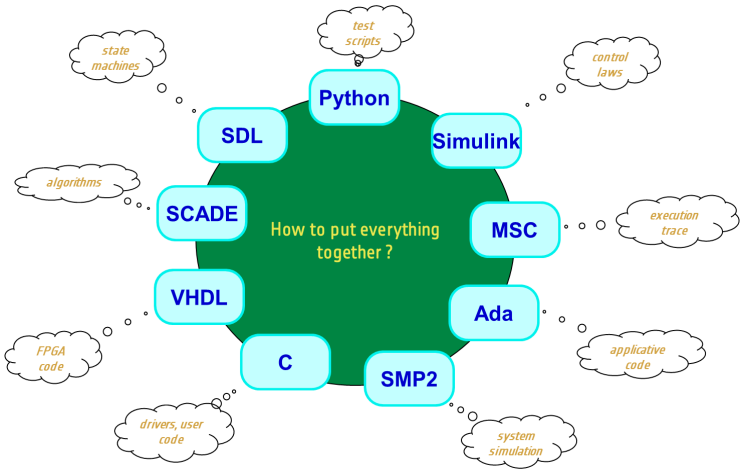

Figure 1 - Languages

In practice, putting all these technologies together can be tricky, especially when the resulting software has to follow standards that restrict the set of constructs that are allowed. For example, space standards and coding rules generally forbid the use of dynamic memory allocation onboard; when using automatic code generation, it means that the tools have to comply to these restrictions in a provable manner.

The way integration is done is mostly manual today; it is not difficult, but it requires a lot of hacking, and it is costly to maintain in particular when interfaces change. Commercial modeling tools do not help, as they do not support integration of heterogeneous models or support for sensor/actuator interfacing. It is one of the reasons why we developed TASTE.

Concretely, and from the end user point of view, TASTE consists in two graphical editors that allow to capture the logical architecture of a computer-based system and its deployment on hardware components, and a set of model analyzers and code generators that process user input to verify and glue all software components together.

The editors implement a graphical representation of a standardized textual language called AADL. The first tool is called " Interface view editor":

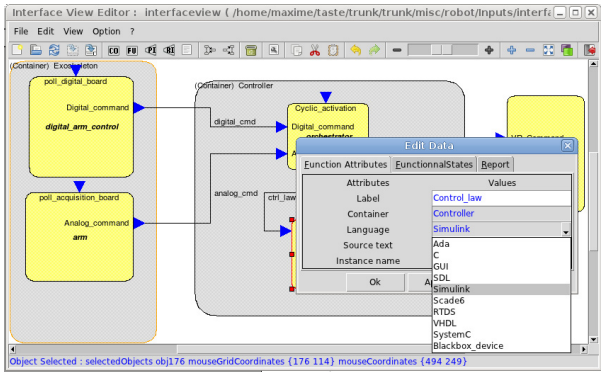

Figure 2 - Interface view editor

Using this tool, the user specifies what his system is made of in terms of functional blocks, what is the precise interface of each block, and in what implementation language or modeling tool the function shall be actually developed.

The TASTE process encourages a combined use of modeling and coding languages depending on the nature of the function – the toolset is then responsible for making them talk together at runtime : this is the heart of the TASTE technology. It is for example strongly encouraged to model embedded systems as a combination of state machines and control laws, because this is often what they are. Following this logic, take the best language for state machines (SDL), the one your scientist knows best for the mathematical models (SCADE or Simulink) and let TASTE fix interfaces at code level so that the SDL-generated code can transparently communicate with SCADE or Simulink-generated code. Of course, if you need to develop low-level functions to pilot a peripheral (sensor or actuator), use C or Ada.

Interface description takes time but is essential and central in the TASTE approach. The interface view editor lets you specify the activation condition of each interface, its timing constraints, the messages it carries, and if it is an entry point for a shared resource. You can also import and export components, including VHDL components.

The second tool is the Deployment View editor :

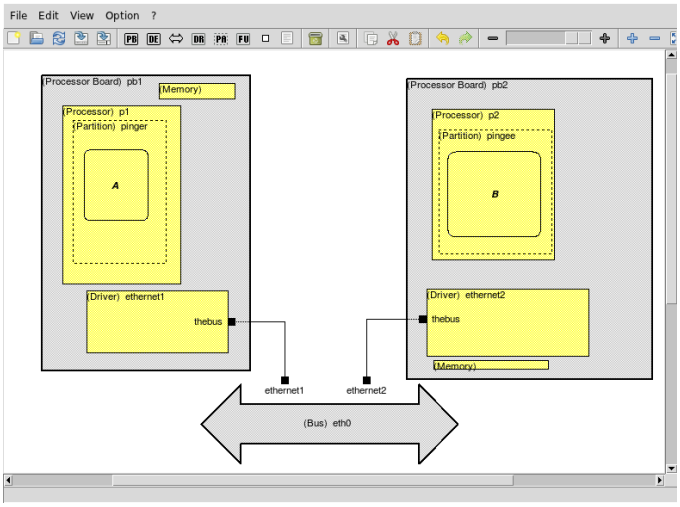

Figure 3 - Deployment view editor

As the name suggests, this tools allows to specify the system deployment, by mapping the functions that were captured in the interface view editor on hardware components (processors), and by specifying busses for inter-processor communication in case of distributed systems.

If drivers are needed to control busses, they can be configured in the tool as properties that the various code generators will process.

2 Two key languages: AADL and ASN.1

To achieve integration TASTE relies on two modeling languages that give enough power to capture all the essential elements of a system that are required to generate the tasks, threads, and glue around the user functional code. These two languages are **AADL and ASN.1**.

Despite not being well known, and sometimes not well understood by the open-source community (due to lack of good open-source implementations), ASN.1 is the key technology for making systems communicate over heterogeneous environments. It provides means to describe messages both in an abstract (human understandable) and physical (binary streams) manners ; tools make the translation from one form to the other, ensuring that messages will be carried from source to destination without any risk of data misinterpretation.

Because it is simple to use and straightforward to understand, ASN.1 was always a natural choice for TASTE given the constraints of embedded systems. Actually, it is even the only valid solution today that is relying on a widely deployed standard (from ISO), which benefit has been proven from years of use in critical areas such as banking transactions, aeronautical communication networks, etc., and for which many tools exist on the market.

However, because commercial ASN.1 tools suffered from unacceptable limitations for us (no support of Ada, generation of code relying on closed-source libraries), and because open-source solutions were not mature, in the scope of TASTE, we

developed a complete new open-source ASN.1 tool-set as a component that can be used in any other application field. It is part of our core technologies, and it provides unique features such as documentation generation, support of SPARK/Ada, C, SDL, SCADE, makes no memory allocation at runtime, no system calls, no dependency on any external library, has a tiny footprint, generates test cases for code coverage, etc.

To give a simple example, the following ASN.1 construct can be written to specify a message received from a sensor:

```
-- Analog inputs are 16 voltage lines in range 0 to 6 volts
Analog-Inputs ::= SEQUENCE (SIZE(16)) OF REAL (0.0 .. 6.0)
```

This means that the application expects to receive sixteen real numbers which values can be between 0.0 and 6.0. The ASN.1 compiler will generate a semantically equivalent representation of this data type in the implementation language chosen by the user, for example in C:

```
double Analog_Inputs[16];
```

But this is not sufficient: the sensors producing this data may not format it in the same way as the memory representation produced by the C compiler. To cope with that, our ASN.1 tools allow to specify how data is precisely marshaled:

```
Analog-Inputs[size 16] {
     dummy [encoding IEEE754-1985-64, endianness little]
}
```

This tells to the tools: “when data arrives, it is formatted using 64 bits, little-endian. Generate the code that makes it fit with the C structure that the user manipulates (which in case of a SPARC/Leon processor, is big endian)”.

The second key language used in TASTE is AADL. It is a textual language that we use to capture the architecture of TASTE systems, in terms of functional blocks enriched with non-functional attributes such as real-time constraints. Just like ASN.1, AADL is simple to understand and to write – two essential characteristics when developing tools. The AADL description of a functional block can look like this:

```
SYSTEM Control_law
    FEATURES
      Control_law : PROVIDES SUBPROGRAM ACCESS FV::Control_law
        {
          Taste::RCMoperationKind => unprotected;
        };
    PROPERTIES
      Source_Language => SIMULINK;
END Control_law;

SUBPROGRAM FV::Control_law
    FEATURES
      in_analog : IN PARAMETER DataView::Analog_Inputs
        { Taste::encoding => ACN; };
      out_vr    : OUT PARAMETER DataView::VR_Model_Output
        { Taste::encoding => UPER; };
END FV::Control_law;
```

3 What is new in TASTE – key features

Designing an embedded platform made of software and hardware blocks is not limited to the seamless integration of software components, may it be a very

important and possibly difficult activity. Here are the features TASTE provides to cope with the overall development process, including simulation and analysis.

3.1 Support of device drivers with hardware legacy encoding

A functional block in TASTE can be implemented with a special language we call "blackbox device". When a message is sent to this block, the data is marshaled following the binary encoding rule specified by the user in his model, and corresponding to the format usually imposed by the device itself. The user has to implement the code corresponding to the driver of the device (access to low level registers, bus, etc.), and can directly send to it the properly formatted message. He can define a periodic activation to the "blackbox device" if he needs to poll his hardware. When he gets some data, this data will be decoded automatically also thanks to the ASN.1 description – no need to manipulate bits and bytes, but only focus on the behavior of the system.

3.2 Generation of function skeletons

For each supported language, TASTE generates function skeletons, which leaves to the user the sole responsibility for filling the gaps between received inputs and outputs to emit. For example, using Simulink, such a model is generated by TASTE:

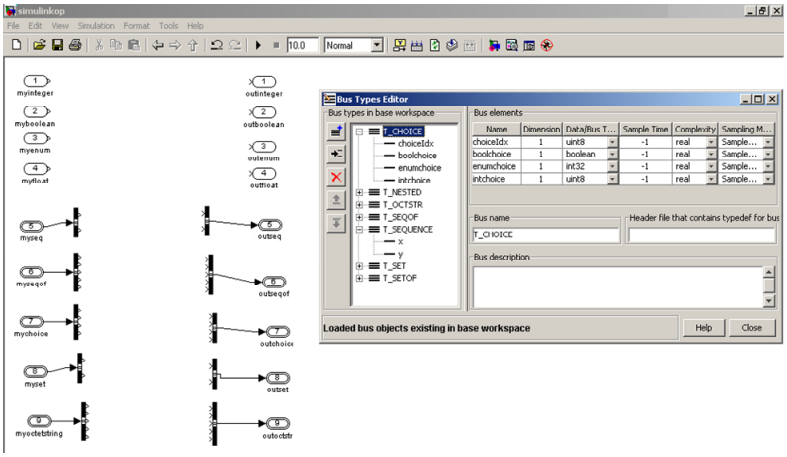

Figure 4 - Simulink code skeleton

3.2 Support of multi-formalism in one system

TASTE allows in a single system to mix functions that are implemented in different languages: SDL (using Real-Time Developer Studio or ObjectGEODE for the code generation), SCADE, C, Ada, Matlab Simulink. The principle is that what counts for TASTE is the interface of the functions: data types, and activation entry point. Knowing how tools generate code, and knowing the ASN.1 data structures converted to Ada and C code, TASTE is capable of producing wrappers around the code to make sure data is always properly conveyed from one function to the other.

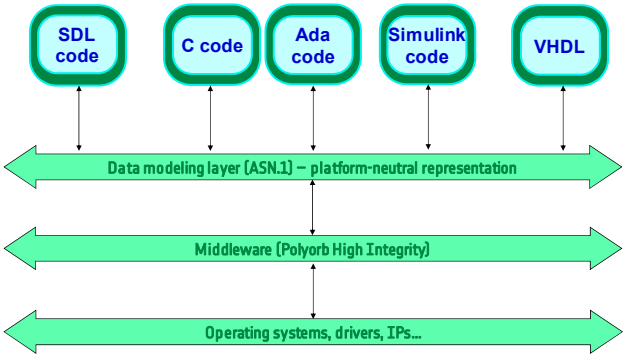

Figure 5 - Runtime architecture

3.3 Support of VHDL on FPGA devices

A functional block can be implemented in VHDL and deployed on an FPGA hardware component. This is useful for example to implement accelerators for some computing functions. In such case, TASTE converts the ASN.1 types to VHDL, generates the function skeleton and all necessary code (both in VHDL and C) that makes the communication between the blocks.

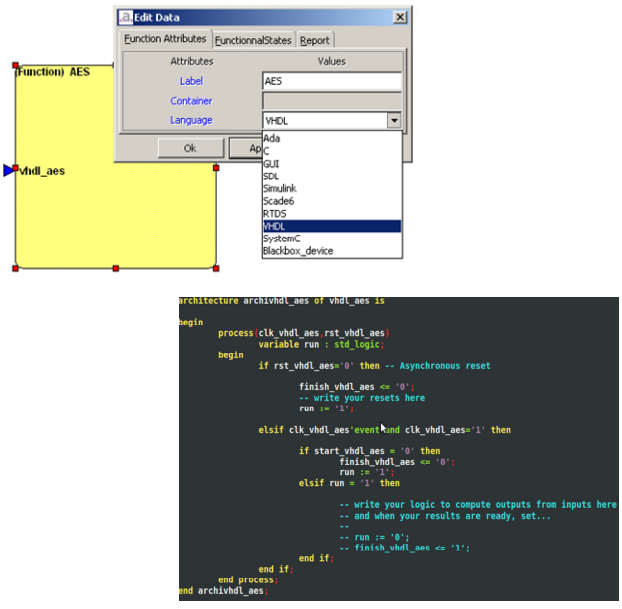

Figure 6 - VHDL Code skeleton

3.3 Support for distributed systems

The TASTE deployment view allows to specify processors and busses. A number of drivers is already provided to communicate between processors with various means such as sockets (Ethernet), serial link and Spacewire. Several processors are supported natively, such as x86 and SPARC/Leon2 (the typical processor used in today's satellites), and we are running experiments on the ARM processor. TASTE

applications run on top of different operating systems, either real-time (RTEMS, Xenomai/Linux, GNATforLEON) or not (native Linux).

3.3 Generation of GUIs

There are many situations where the user may want to interact with the running system: for example to "stub" a missing function before it is implemented, or simply because the system needs to be fed with external data. For those situations, TASTE automatically generates graphical user interfaces allowing to feed data types according to the user interface definition (in ASN.1), and the mechanisms to send the messages to the system.

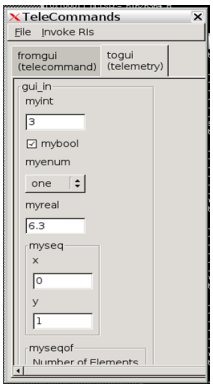

Figure 7 - TASTE auto-generated GUI

3.3 Data inspection and patching at runtime

In addition to communicating with the system on user-defined interfaces, debugging an embedded application may also mean to be able to monitor and patch variables in memory without the need for adding explicit interface for each memory address you want to inspect (think of the Simulink "scopes"). This service is offered by TASTE when needed. Any place in memory can be inspected at runtime, possibly periodically, and if required, a value can be updated.

Values are displayed in a tabular format, but can also be plotted in real-time, or displayed using a meter form.

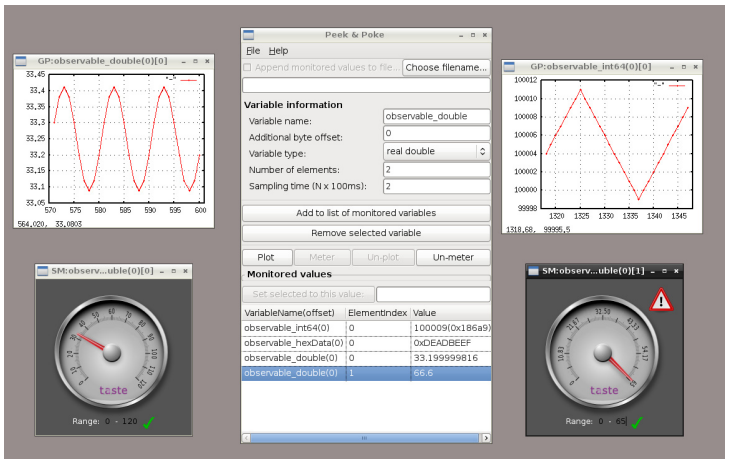

Figure 8 - TASTE PeekPoke function

3.3 Tracing messages with MSC

When analyzing a system, TASTE can trace all interactions between functions, and display them at runtime using sequence diagrams. This is a very convenient way to see what is happening and possibly derive regression tests on the system.

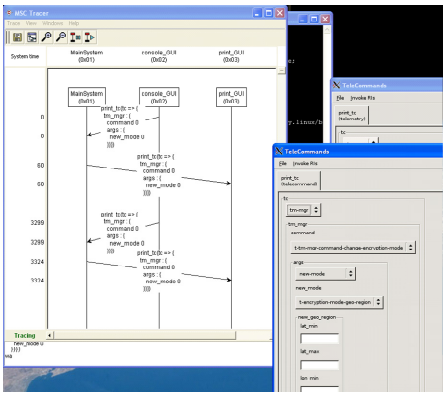

Figure 9 - MSC Tracing and recording

3.3 Model and code analysis

TASTE integrates two scheduling analysis tools (MAST and CHEDDAR), which work on AADL models. When all timing values are known, it can be useful to check if the system can actually run on the selected target. This is what these off-the-shelf components provide.

Based on the generated binaries, TASTE analyses stack usage and warn the user in case there is a risk of stack overflow.

3.3 Script testing

Another major feature of TASTE is the possibility to write scripts and execute them against the binary running on target, for testing purposes. There are three ways of doing this:

1. By writing scripts from scratch, in Python (TASTE generates a Python interface to exchange message with the running system);
2. By writing MSCs (sequence diagrams) using an editor and using the MSC to Python converter provided by TASTE;
3. By using the MSC recording capability that automatically generates regression tests by producing Python code to verify on a subsequent run of the system that it behaves the same way as when the script was recorded.

3.3 Generation of documentation

TASTE has the capability to generate documentation on the data streams used when conveying a message on a different target (when a *memcpy* of the data is not sufficient to guarantee that it will be properly understood at the other end).

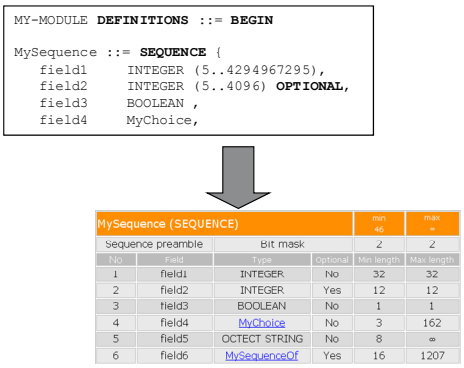

Figure 10 - ICD Generator

4 Conclusion and future

TASTE exists today as a full-featured application. It is available and can be used out of the box, as we distribute it in the form of a complete, pre-configured virtual machine. At the moment, TASTE has been used for the implementation of various medium-sized systems (robotics, control systems).

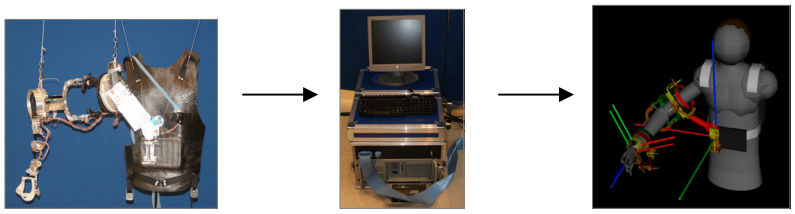

Figure 11 - A TASTE case study set-up

The level of maturity of TASTE fundamental elements makes it a platform of choice for experimenting technologies and for educational purposes. We are deploying TASTE at ESA to study hardware-software co-design, partitioned systems, but also to improve our system-based simulators, as well as to consolidate our standards and means for producing specifications of new systems. Another path we are working on is to see how we can improve system analyzability using the information from the models.

The current main contributors to TASTE are the European Space Agency (leading the project), Semantix Information Technologies, Telecom ParisTech, ISAE, Ellidiss, and UPM.

References

1. Perrotin, M., Conquet, E., Delange, J., Schiele, A., Tsiodras, T. : TASTE: A Real-Time Software Engineering Tool-Chain, Oveview, Status and Future, Proceeding of the 15th International SDL Forum, July 2011, Toulouse, France
2. Perrotin, M., Conquet, E., Dissaux, P., Tsiodras, T., Hugues, H. : The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software, ERTS'2010, Toulouse, Toulouse, France
3. Mamais, G., Tsiodras, T., Lesens, D., Perrotin, M. : An ASN.1 compiler for embedded/space systems, ERTS'2012, Toulouse, France