This is an author-deposited version published in: http://oatao.univ-toulouse.fr/
Eprints ID: 3357

To cite this document: CONQUET Eric, PERROTIN Maxime, DISSAUX Pierre, TSIODRAS Thanassis, HUGUES Jérôme. The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software. In: *European Congress on Embedded Real-Time Software (ERTS 2010)*, 19-21 May 2010, Toulouse, France.

Any correspondence concerning this service should be sent to the repository administrator: staff-oatao@inp-toulouse.fr

The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software.

A. M.Perrotin[1], E. Conquet[1], P. Dissaux[2], T. Tsiodras[3], J. Hugues[4]

1: European Space Agency, Keplerlaan 1 - 2201 AZ Noordwijk - The Netherlands
2: Ellidiss Technologies, 24, quai de la douane, 29200 Brest, France
3: Semantix Information Technologies, K. Tsaldari 62, 11476, Athens, Greece
4: Toulouse University/ISAE at 1 place Emile Blouin, 31056 Toulouse, France

Abstract: The TASTE tool-set results from spin-off studies of the ASSERT project, which started in 2004 with the objective to propose innovative and pragmatic solutions to develop real-time software. One of the primary targets was satellite flight software, but it appeared quickly that their characteristics were shared among various embedded systems. The solutions that we developed now comprise a process and several tools ; the development process is based on the idea that real-time, embedded systems are heterogeneous by nature and that a unique UML-like language was not helping neither their construction, nor their validation. Rather than inventing yet another “ultimate” language, TASTE makes the link between existing and mature technologies such as Simulink, SDL, ASN.1, C, Ada, and generates complete, homogeneous software-based systems that one can straightforwardly download and execute on a physical target. Our current prototype is moving toward a marketed product, and sequel studies are already in place to support, among others, FPGA systems.

Keywords: AADL, SDL, ASN.1, TASTE

1. Introduction

The area of software engineering has hardly evolved in the past years. People often talk about the increasing complexity of today's and tomorrow's systems, as well as the fact that this complexity is difficult to master; however, factually based very little innovative methods have come out to propose solutions that could beat the classical engineering approach - consisting in capturing software requirements using a good old word processor and coding them on the fly. By chance, there are still experienced coders who are capable of interpreting system engineer's ambiguous wishes and make a system that at the end does something. But it must be said in the defence of software engineers that the task is hard: they have to come out with means that will be accepted both by system engineers – who don't like tools, and software developers – who don't like GUIs. A somewhat blind craze showed up for graphical approaches that were sold as the ones that would finally solve the so-called software chaos. Unfortunately, and partly because of a lack of thought on the nature of the systems under consideration and on the needs in terms of development and validation processes, this approach quickly reached an impasse.

This paper presents various facets of the software development methods that were explored by the actors of the space domain in the scope of the ASSERT project – a project that was co-funded by the European Union and by about 30 industrial and academic partners, and that ran from 2004 to 2007. This project is still very active thanks to an initiative from the European Space Agency to fund several follow-up activities, forming what we call now TASTE: The ASSERT Set of Tools for Engineering. The principal results comprise a development process, which is supported by the implementation of a consistent tool-chain that offers novel capabilities that are useful to cover the needs for capturing a system's properties and derive from it a distributed real-time software. We do not present this solution as revolutionary ; we rather think that it is federative, because it combines existing and mature technologies, making a link where it was missing, and bringing concrete, directly usable solutions to real user needs.

2. The starting point

When we started this project, we first asked ourselves if there was anything that was so typical to our systems that could explain or justify why barely any tool was used to support the development phase - no modeling tool, no code generators, no formal requirement capture. We have many standards which are very demanding in terms of software documentation and quality but operational projects will hardly comply to requirements regarding the need for early verification of the system based on models. Then by looking closer to the existing tools, and there are many on the market, we realized that nothing really addressed our problems in a way that was really improving the situation. Most of the tools

do not care about building complete systems, because they are not meant for engineers – actually, their target users is rather unclear ; they will produce nice drawings in the best case, and in the worst case will bring a strong headache to its users. Let's be fair! That's not entirely true: if we take a few tools independently from each other, we can make nice things: control laws, powerful state machines – but not a single tool addresses our systems as a whole.

To explain this last statement, look at these observations about our systems – and probably yours: first the nature of embedded systems is profoundly and fundamentally heterogeneous. This heterogeneity goes over at least three axis:

1. the nature of the software capabilities that are required
2. hardware buses and processors
3. industrial consortia making the systems

Inside a satellite, software realize various functionalities that are very different one from each other: control laws to move the spacecraft in orbit and perform attitude manoeuvres, system mode management, mission planning, thermal control, communication protocols, etc. In order to achieve these functions, project teams gather different specialized skills from scientists and software engineers.

In the hardware world,it is the same story: the microprocessors we send in space are most often derived from the SPARC architecture, in which the internal data representation in different from the ones of the standard PC used to control spacecrafts from the ground. For this reason, it is for example not an option to send a raw data to the ground without doing a conversion to make sure it will be interpreted correctly. On-board buses can themselves have addressing schemes that require a pre-processing to invert bits before using them. This kind of manipulation, related to bit endianess, is of course well known and frequent, but remains hard to implement because programming languages do not provide easy mechanisms for bit manipulation.

Third, in the space sector it is quite common that the development of one on-board software involves teams spread in several companies, from several countries, each having its own way of working, development environment, and so on.

It is therefore clear that our systems deal with heterogeneity, and it is obviously also the case for most embedded systems.

The second observation we made is that when we build a new system – whatever system it is - what counts most is what this system is thought for, its goal. We want to make sure that the biggest effort is made on what makes the system novel and different from what has already been done in the past. For instance, the orbit and the trajectory of a satellite are central issues that require an important engineering effort. During this phase, do we need, do we want to be embarrassed with implementation details and already introduce bulky software artifacts? If the answer seems obvious, look at existing, real software system specifications and count how many times you find references to semaphores, binary frame definitions, thread identifications at the very first outline of a project. The temptation is strong because it is important to occupy software engineers early, when they have no direct knowledge of the system, hence no real added value regarding its definition. What we want to point out here is that whatever solution we come with, it has to be related to the system needs, and not to software issues.

A third observation ensues from this last points: skills of software engineers in this context are often misused ; it is a known problem: we ask to software experts, whose aspiration is to solve technical challenges (such as optimal resource usage) to develop applicative code. This code, which is often quite simple, consists in performing algorithms that other people have conceived, as they are the domain experts. At best, this generates a frustration, and at worst the best software developers prefer moving to pure software companies where they think they can be more efficient. The space sector is partially protected from this extreme situation, thanks to its particular appeal but of course it is not always the case. Then replacing experienced and valuable people can become a real challenge now that many schools and universities have given up with low-level languages and concepts to privilege web-based developers and Java.

3. Common solutions

In practice, very few solutions exist that address the problems that we just exposed. In fact, for about 15 years, it is not exaggerated to say that an important part of the software community simply ignored them, favouring the solution that seemed to be accepted by everybody: the UML language. Flouting all efforts in formalizing both syntax and semantics that used to be considered as so essential not only to programming languages but also to most other existing modeling languages (SDL, Lustre), UML gave up with the idea that the development of systems needed to be supported by a process, and rather proposed a huge palette of sometimes abstruse graphical editors. Nobody really understood anything of these, so everybody started enhancing the language by adding new concepts, new profiles in an inconsistent manner. The unfortunate result is that UML is mostly used only to help making drawings for documentation, and that most tool vendors have disappeared from the market. In addition to this, companies feel that they have

invested a lot in new technologies and that nothing concrete came out if it, which of course had the bad side effect to undermine the credibility of people proposing alternate technologies, since in general the arguments used by one of the other are very close. In theory, modeling software is a major evolution of the discipline – combined with powerful tools, it can be a breakthrough in how we think and develop systems, we have no doubt about that. But decision makers now hardly believe it, and we cannot completely blame them for that.

The idea that a unique language can address all the facets of a software development is absurd and unrealistic, as it amounts to denying the heterogeneous nature of systems. In practice, who has seriously thought of replacing Matlab or SCADE by UML to design a control law?

4. What ASSERT/TASTE propose

The ASSERT process and tools propose a different solution to address in a very pragmatic way the problematic of capturing and implementing systems using formal modeling techniques.

Systems being heterogeneous by nature, one issue is to make them communicate, in other words integrate the heterogeneous subsystems in the most efficient and transparent way. What TASTE does is to automate this integration phase in a way that it replaces any manual (risky) intervention on the code.

Using formal languages early in the development process allows to envisage on the models various property checks, using specialized tools. TASTE allows domain-specific experts to develop their part of software in the language that they think is the most appropriate for their need, without having to care about any implementation constraints. The tool takes care of the rest. Only the non-functional, critical properties are captured at system level ; then a sophisticated and evolutional machinery is invoked that produces a consistent software set, guaranteeing that the constraints imposed by the system designers are respected at run-time. At the end, TASTE generates complete real-time, possibly distributed applications running either on top of a real-time operating system combined with a middleware, or simply on a native, non-real time environment such as a Linux box. In addition to this, many functionalities allow to put in place an efficient and iterative development, facilitating tests and analysis of results at runtime. All these features are presented in the following parts of this paper.

To sum up the overall ASSERT process, we have four major steps:

1. a system **modeling** phase, abstracting away all purely software constraints,
2. a **transformation** phase, resulting in a real-time software architecture containing tasks, threads, and shared resources,
3. a **feasibility analysis** phase, verifying statically that the user expected properties are effectively attainable using the selected physical architecture,
4. and finally a **code generation** phase, putting all the individual blocks together and finally generating a set of binary files that can be directly executed on target ; all the code that handles communication between subsystems is produced here with no need for any manual intervention.

The organisation of the following chapters is the following: sections 5, 6 and 7 detail the bowels of the TASTE technologies, explaining each step of the process and tools to give the reader a complete picture of what we have today. Then the last section relates our first feedback from external users who worked with TASTE, and the future of the toolset.

5. TASTE modeling process and tools

Overview

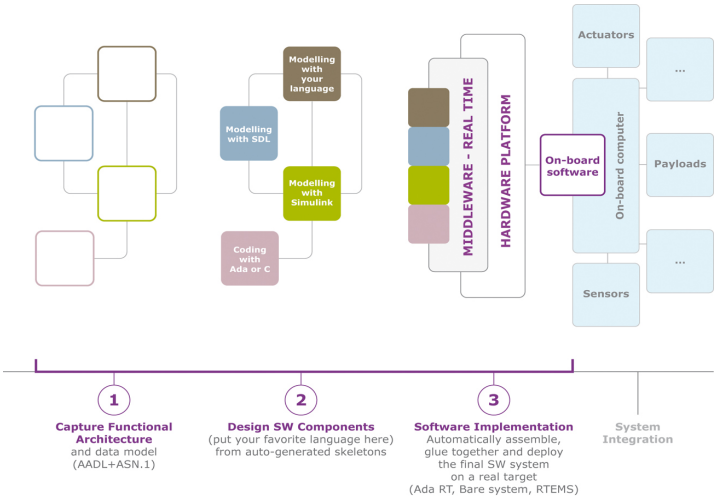

Figure 1: Process overview

Figure 1 shows an overview of the global modelling process that is proposed by TASTE. Prerequisites consist in the availability of a preliminary software system logical architecture, which we suppose results from a joint work between system engineers (who know what they want the system to do) and a software architect. This phase is today out of our scope. What we need is to know the main capabilities of the system, preferably already translated into a set of functional blocks. Our tools

then allow to capture this knowledge and make an intensive use of this precious information.

The main idea is that we do not want to impose a particular language or tool to implement the functional blocks themselves: users can choose the ones they consider to be the most appropriate for each block, and let TASTE take care of the integration. In practice, TASTE currently supports Matlab/Simulink, SDL (ObjectGEODE and Real-Time Developer Studio), C and Ada languages.

To support proper communication flows between the functions, a common standard is used to specify involved data types. In TASTE, the chosen data modelling standard is ASN.1 from which it is easy to automatically generate the encoding and decoding procedures that are required for the final integration of the application.

Another common language is used throughout the whole tool chain as an architectural framework to support the functional and non functional properties of the system. The AADL has been selected to play this role in TASTE. However, it is not required for the end user to write AADL code at any time as it is automatically generated by the various domain specific tools that compose the TASTE tool chain. But if user wants it, nothing is hidden in cumbersome XML files and it is always possible to work at language level – we think that development and debugging are more efficient when information is readable by human.

Interface view editor

The interface view editor is a graphical tool that aims at describing the logical interactions between the various functions of the system. In order to support large scale architectures, functions can be grouped into hierarchical containers.

Each function (represented by a blue box in figure 2) is described by its provided and required interfaces. Provided interfaces (blue triangles) are themselves characterized by a set of non functional properties and represent activation entry points of the function.

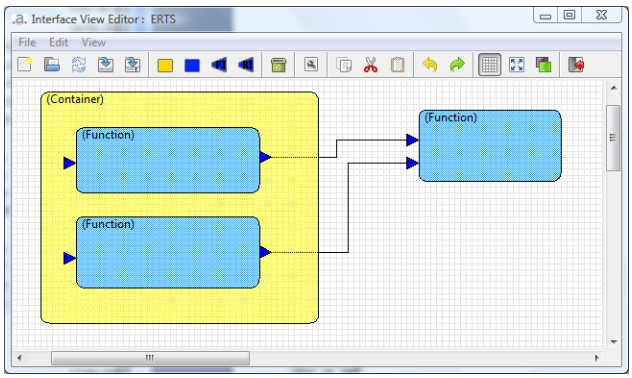

Figure 2: Interface view editor

Finally, connections can be defined between required and provided interfaces to express the logical functional dependencies inside the system. The result of this modelling work is saved into an AADL file for further use within the tool-chain.

Deployment view editor

The deployment view editor is another graphical tool that is used to describe the hardware architecture of the system and allocate the functions identified in the interface view onto partitions located on a processor.

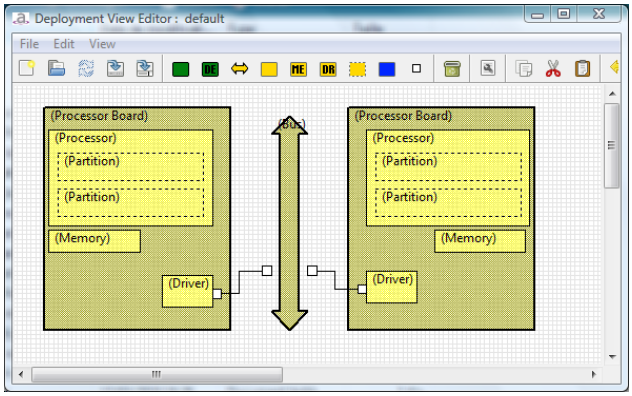

Figure 3: Deployment view editor

Inter processor communications can be specified through buses and bus drivers. Each of these modelling entities can be characterized by a set of properties that are necessary for further code generation. Like for the interface view, this modelling work is stored in an equivalent AADL textual representation.

Vertical transformation

The result of both interface and deployment views edition can then be submitted to a "vertical transformation" tool. The aim of this fully automated activity is to produce a complete combined software and hardware architecture encompassing all the real-time and distribution properties of the system (in particular a set of processes, threads, shared

resources...). The output of the transformation is another AADL specification that is called the concurrency view.

Concurrency view editor

Although the concurrency view can be seen only as an intermediate internal step within the tool chain, it brings a unique opportunity to perform performance analysis on the system at a model level.

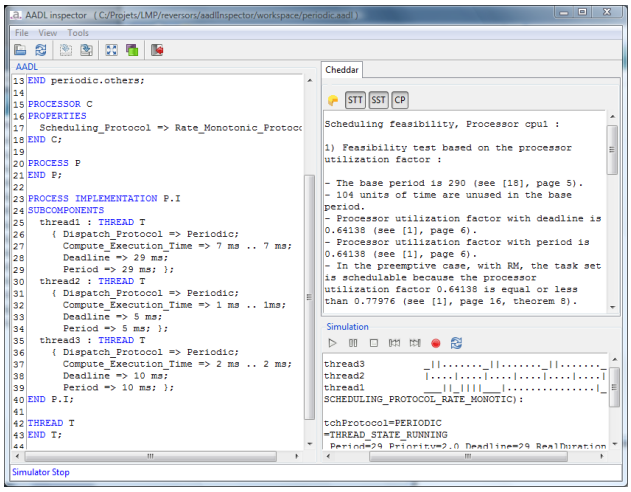

Figure 4: Concurrency view editor

As the concurrency view is described as a complete and legal AADL architecture, all the existing AADL analysis tools can be used at that stage. The tools that are currently integrated into the TASTE concurrency view editor are Cheddar (see [4]) and a dynamic simulator.

Code generation

The last step of the TASTE modelling process consists in building the executable application from the functional blocks, the glue code generated by TASTE to handle transparent communication, and the AADL architecture defining the hardware and software interactions of the system.

The Ocarina (see [5]) tool is in charge of this task and generates the complete compilable set of source files while taking into accounts the run time execution characteristics of the Ravenscar Computation Model that has been selected for TASTE. Then compilation and link are performed automatically by the tool-chain orchestrator. Several possible operating systems can be used: bare systems using the Ada runtime (i.e. any operating system having a an implementation of the GNAT compiler), but also the RTEMS real-time operating system (not depending on the Ada runtime), which is a standard operating system in space and military applications. Note however that using a C runtime such as RTEMS prevents from the benefits of Ada compiler checks (that can make sure the user does not use forbidden constructs in his code).

We will now give some concrete example of what you can find in TASTE models and what important features it proposes.

6. Technology

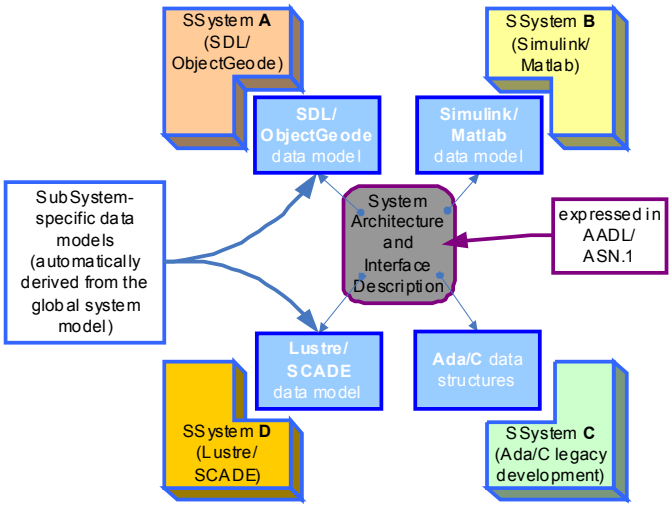

Figure 5: Technology overview

With the work described in the previous section, TASTE uses a high-level architectural view of the system, that formally depicts the partitioning of the overall system in distinct subsystems and their interfaces. This information is expressed in the Architecture Analysis and Design Language (AADL).

The following is an excerpt from an actual design:

```
TASTE_Properties::RCMoperation=>SUBPROGRAM
Packet_Router_Deposit;
TASTE_Properties::RCMoperationKind => sporadic;
TASTE_Properties::RCMperiod => 50 ms;
…
SUBPROGRAM Packet_Router_Deposit
 FEATURES
  Packet31 : in PARAMETER DataView::RequestGNC
    { TASTE_Properties::encoding => UPER;};
…
 PROPERTIES
  Source_Language => Simulink;
…
```

As seen in the example, the interface descriptions include information about the

- execution profile of the interface – e.g. timing information like period or worst case execution time (WCET), call type (cyclic, sporadic, etc)

- implementation language/tool of the interface (e.g. “Simulink”)
- naming and direction of the interface parameters (e.g. “Packet31”, “in”)
- type of the interface parameters through ASN.1 grammar specifications (in the example above, “RequestGNC” is a type of the ASN.1 module “DataView”).
- ASN.1 encoding specifications (e.g. “UPER” stands for Unaligned Packed Encoding Instructions, one of the many available ASN.1 encodings).

The types of the interface parameters are described in ASN.1 specifications. ASN.1 is an ISO/IEC and ITU-T standard that allows for specification of data structures, both from the semantic as well as the encoding point of view. It is widely used in telecommunication protocols, and has been selected for use in TASTE. The following is the definition for the example “RequestGNC” used above:

```
RequestGNC ::= SEQUENCE {
...
  small-source INTEGER ( 0.. 40),
  coarse-time-rep Time-Component,
  category ENUMERATED {
    urgent(0),
    normal(1)
  }
}
```

It includes all the semantic information about the data carried across the interface’s invocation, as well as the limitations (ASN.1 constraints) on the values that are allowed to pass through. For example, the first field (“small-source”) is an integer that must be limited in the [0 .. 40] range.

The formal descriptions of interfaces (in AADL and ASN.1) allow TASTE to automatically handle a number of issues by using the provided information.

Model translations (“model to model” phase)

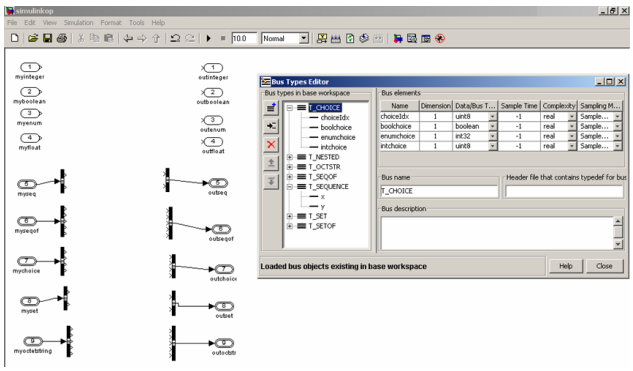

Figure 6: Skeleton file example generated for Simulink

This transformation is supported for a variety of modelling tools (Simulink/RTW, ObjectGEODE, Pragmadev RTDS, etc.) and implementation languages (Ada, C/C++, SystemC/VHDL). Since it works based on the AADL/ASN.1 model, it is always guaranteed to generate the same semantic content for the interface parameters, regardless of the implementation tool/language – i.e. the “translated” definitions of the ASN.1 types are semantically equivalent in all the supported target tools/languages.

Code translations (“model to code” phase)

When functional modeling is completed, the modeling tools' code generators are invoked, and C code is generated. Modeling tools generate code in different ways, however – and even though (thanks to the previous step) the data structures of the generated code across different modeling tools are carrying semantically equivalent information, the actual code generated cannot interoperate as is:

```
// Declaration from ObjectGeode
typedef struct {
    GU_RG_51_10 fd_height;
    GU_RG_50_9 fd_latitude;
    GU_RG_49_8 fd_longitude;
    GU_SEQOF_52_11 fd_subtypearray;
} GU_T_POS;

// Declaration from Simulink
typedef struct {
    real_T longitude;
    real_T latitude;
    real_T height;
    Subtypearray_type subTypeArray;
} T_POS;
```

Figure 7: Code generated by commercial tools

Therefore, integrating the code generated by different modeling tools requires “data bridges” to be built that translate (at run-time) the data structures from one modeling tool to those of the other and vice versa. Manually creating these data bridges would be a very error-prone process, and would have to be repeated if the messages were changed. In TASTE, they are automatically built by our custom-made code generators.

Automated GUIs and regression checking Python scaffolding

In the overall AADL system design, the designer can specify the subsystems for which a graphical user

interface should be created. The TASTE toolchain reads the interface information of these subsystems and automatically generates code for interactive graphical user interfaces that operate on these interfaces. These GUIs provide real-time access to running systems, allowing information exchange, e.g. invocation of telecommands or receiving real-time telemetry.

The same information is also used in order to build Python run-time bridges that allow real-time interaction with a running system. Complex regression checking suites can be written easily, with the combined clarity and brevity (and developing speed) of a ubiquitous scripting language.

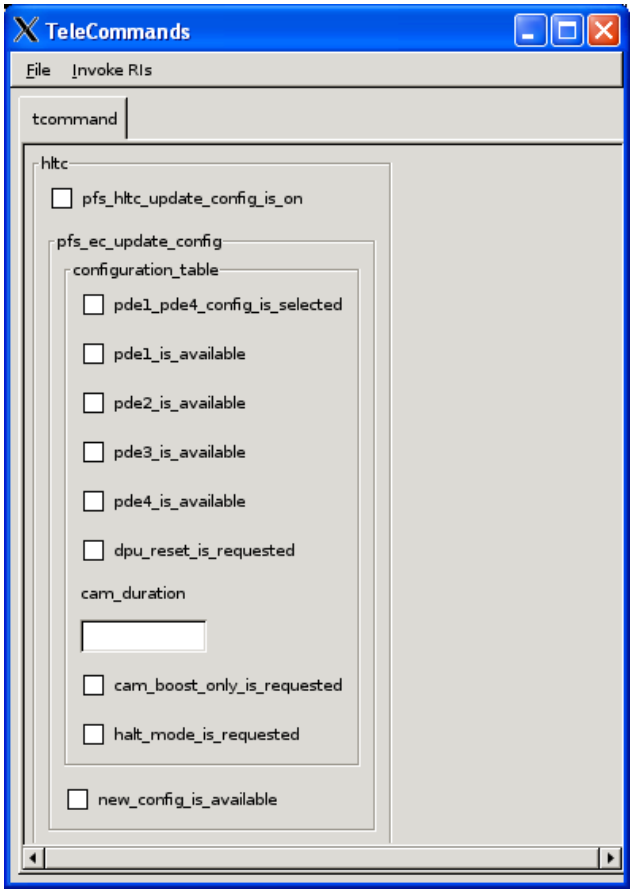

Figure 8: Auto-generated GUI

Telemetry can then be piped to plotting and monitoring applications, for easy real-time monitoring and control of running systems.

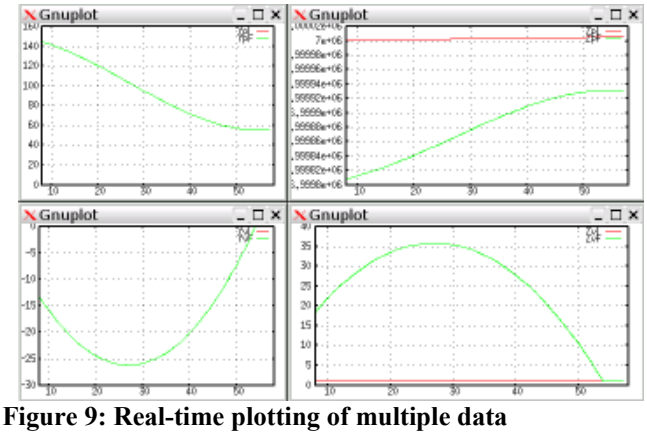

Figure 9: Real-time plotting of multiple data

Interface Control Documents (ICDs)

In order to support legacy development, one of the TASTE tools (the ICD generator) automatically creates an Interface Control Document that describes all interface parameters as they get encoded at the bit-level - from ASN.1 encoding. This allows interoperating with other development teams that choose – for whatever reason – to not use the TASTE tools. Following the same philosophy as the rest of the TASTE tools, the ICD generator allows the designers to get free and immediate updates of their ICD, without the cost (and potential errors) that is involved in a manually-maintained ICD.

```
MY-MODULE DEFINITIONS ::= BEGIN

MySequence ::= SEQUENCE {
   field1     INTEGER (5..4294967295),
   field2     INTEGER (5..4096) OPTIONAL,
   field3     BOOLEAN ,
   field4     MyChoice,
```

MySequence (SEQUENCE)				min 46	max ∞
Sequence preamble		Bit mask		2	2
No	Field	Type	Optional	Min length	Max length
1	field1	INTEGER	No	32	32
2	field2	INTEGER	Yes	12	12
3	field3	BOOLEAN	No	1	1
4	field4	MyChoice	No	3	162
5	field5	OCTECT STRING	No	8	∞
6	field6	MySequenceOf	Yes	16	1207

Figure 10: Auto-generated ICD

ASN1SCC and ACN (ASN.1 encoding Control Notation)

Since the primary target of the TASTE process and tools is the space domain, we created a custom ASN.1 compiler (ASN1SCC) that generates code specifically designed to be executed in limited-resource environments. It involves no dynamic memory, it uses no system calls, and is portable to all the target architectures, including Leon (i.e. the generated code includes no outside references to “black-box” libraries).

To support legacy encodings and be able to communicate with existing protocols and implementations, the ASN.1 compiler was enhanced with the ASN.1 encoding Control Notation (ACN) that allows for direct control of the encoding – that is, the binary format of the generated streams.

Support for HW development

The TASTE methodology and tools have been recently upgraded to support development (and automatic integration) of hardware components as well. If a subsystem is marked with “VHDL” or “SystemC” in the high-level AADL specification of the interfaces, it automatically gets VHDL and SystemC skeletons (in the “model-to-model” phase described before) as well as the appropriate device drivers (in the “model-to-code” phase) that communicate with the chip at runtime.

7. The TASTE runtime

Modelling time and effort is a valuable asset that is to be used and preserved down to the construction of the final system. To do so, the TASTE tool-chain integrates a set of code generation tools to map all models down to source code targeting a dedicated run-time environment. Let us describes it from a top-down perspective.

From the full set of models (ASN.1 and AADL), we have a complete description of the system: types manipulated, interfaces of processes and threads, connection topology and flow of information and interaction. We rely on Ocarina code generation facilities to generate optimized code for all entities that can be optimised through a careful examination of the architecture: communication buffers, structure of requests, request marshalling/unmarshalling, optimized task body so as to avoid dead code.

We have extended the Ocarina AADL-to-code tool-chain to also integrate device drivers as model artefacts. Such modelling allows seamless integration of both functional code (as application blocks), but also device drivers.

In the context of TASTE, functional code is the output of the previous code generation steps: code generated for marshalling ASN.1 data types definition, or generated from other modelling framework supported by TASTE: SDL, Simulink, …

Device drivers are integrated as functional models with a specific interface for (1) initializing the driver using dedicated API provided by the underlying RTOS, (2) sending or receiving data. (2) is modelled as any functional block using the same modelling artefacts as the functional code for concurrency (e.g. how to process data in parallel, etc.), and the call the driver API to perform the actual send/receive. Such approach greatly eases the integration of protocols or drivers: they are seen at the same level as functional block, and take advantage of the whole TASTE tool-chain to combine functional blocks, drivers and the generated code.

The PolyORB-HI runtime

Generated code is targeting the high-integrity run-time infrastructure PolyORB-HI. This infrastructure acts as a portably layer for the integration of multiple languages (C or Ada), RTOS APIs (Ada Ravenscar, RT-POSIX, RTEMS), but also for the integration of device drivers (serial, Ethernet, SpaceWire).

PolyORB-HI acts as an AADL runtime: it provides support for each model patterns defined at the upper-level. Two variants of PolyORB-HI have been implemented:

An Ada variant, that relies on the Ravenscar Computational Model (RCM). It defines a set of patterns for deterministic concurrency. It makes provision for analyzability through the RMA and RTA frameworks. Besides, great care has been taken to ensure the code meets more stringent requirements for High-Integrity: the compiler to ease code review, and strengthen quality enforces restrictions that forbid explicitly dynamic memory, object-orientation or pointers. This variant runs either on native systems, RTEMS, or on the bare-board ORK+ or GNAT Pro for High-Integrity runtimes.

A C variant, that uses the same concepts from the RCM, on top of the RTEMS operating system, or the RT-POSIX. Although C provides less support to check code quality, great care has been taken to ensure a level of quality similar to the Ada variant.

The choice of one variant is mainly dictated by the availability of specific device drivers (e.g. CAN, MIL-1553, GPS receiver, etc.), or non-functional properties like memory overhead of the RTOS, run-time performance (such as WCET or jitter), and analyzability features.

Current case studies did not evaluate in full depth schedulability of systems. This is a current on-going work. We evaluated the impact of each variant in term of memory consumption. Ada on top of RTEMS is obviously more demanding in term of memory, then RTEMS/C and ORB+, which is a restricted kernel. Let us note that ORK+ also provides better safety capability thanks to the use of Ada, yet it lacks driver support of RTEMS/C.

	.text	.data	Total
ORK+/Ada	91’392	7’516	98’908
RTEMS/Ada	285’760	12’068	297’828
RTEMS/C	100’016	3’732	103’748

Table 1 Memory consumption (in Bytes)

Both variants provide the same level of support to the application: the same patterns can be applied. Besides, we are currently integrating more drivers to

ORK+ to ensure both variants stand equal from the designer perspective.

8. First user feedback and TASTE future

The complete development of TASTE required a significant amount of work and time to reach the level of a full working prototype with an appropriate level of maturity. Most of the work was initially concentrated on the development process that TASTE is supporting as we consider it as more important than the technologies to be used. Then, we focused on the modeling languages and integration issues to deliver something that requires a small effort at the beginning but brings strong benefits at the end, by automating most of the development phases and ensuring system consistency. When the initial prototype turned to be an efficient product, we decided to give the system and software designers a chance to experiment it.

As for any new product or technology, potential users are first confused with the richness and complexity of the proposed solution. Developers and users have to play together with real open mind for the experiment to succeed. The very first steps on user side were carefully accompanied by a strong support provided by tool developers. Questions were asked and answered, comments were processed and disagreements about the way TASTE was dealing with the process were expressed and discussed. Strong cooperation between teams was essential to pass the first blocking barriers and address the real topics.

Although we claim that TASTE was open to different kinds of existing programming or modeling languages that were familiar to software designers, using the toolset requires the use of two additional languages: AADL and ASN.1. But TASTE designers were clever enough to ease the life of its user by providing a graphical user interface that hide the AADL description behind the scene. This clearly speeds up the learning curve while keeping the advantage of using a system design language backstage for possible future property verification and connection to additional tools. Using such an approach keeps the benefit high with a limited pain at system design level.

ASN.1 was somewhat newer to most of the users but at the same time quite close to very well known programming languages. The idea of having a data model fully connected to a system design and consistently used to produce full software by ensuring the right integration of software components, that was really new to many users. In existing projects, data model are not formally defined and nothing exist to ensure automatic consistency from top to bottom, except the Interface Control Documents but they are just papers. Most of the users found ASN.1 very valuable up to the point where they could envisage the use of this language outside the TASTE environment.

Although the benefits were clearly identified (a user even claimed he successfully generated a complete software implementation that exhibits higher performances than the manually coded version), some limitations were found. A category of users claimed they did not need such technology as they usually do not have heterogeneous systems. In a sense they were right when they see software development as a pure programming activity, and not as a combination of modeling and programming. Another kind of users regretted the absence of key support functions such as traceability management tools, configuration management facilities or document generation features. At least such remarks prove that the core facilities offered by the tool were found efficient up to the point where people may envisage the full deployment of the tool.

The future of TASTE

Following the long and hard development phases of TASTE, and having analysed the first user feedbacks, we are now at a point where the future of this technology shall be carefully defined. Part of this future is made of technical perspectives; the rest is dealing with the toolset itself as a potential commercial product.

From the technical side, we see many open opportunities related to the use of standard languages or coming from user feedback. The wide openness offered by AADL is clearly a strong advantage as it ensures that TASTE can be easily integrated into a system development process using the language to capture and verify system designs. The flexible tool architecture also guarantees the future inclusion of additional languages in a way similar to what we did for the currently supported languages. User suggestions provided us a large set of interesting ideas to improve the usability in a real industrial environment (connection to process support tools, extension of testing features, …).

TASTE in its current state is close to a commercial product that would be usable in an industrial context. Each underlying technology (AADL, ASN.1, …) can be used independently with already a positive impact in a standard development process, but TASTE by itself is more than the sum of its components and brings additional benefits when used in its entirety: automatic design and code generation, consistency insurance with the data model, flexibility with respect to the various development platforms. This led us to open discussions with the development team and

potential users to clearly identify the interest and will and build a commercialization strategy for TASTE, possibly outside the space domain.

Regarding licensing schemes, at the moment most of the TASTE tools follow a GPL licence for non-commercial use (see [2] for details).

Conclusion

The flexibility brought to digital systems by software components is so high that it seems that there is no limit to the functions those systems can handle. But increasing system complexity is now pushing software engineering to the limits of currently used technologies and that convinced the initiators of ASSERT to propose a new approach. The main drivers of this new process are first to capture a minimal set of inputs from the system designer, to automate most of the software implementation tasks and to constrain programmers to the use of rigorous rules. As a positive result, a strong consistency is preserved during system design, multiple implementations can be generated from one unique model and time from design to code is drastically reduced.

This new approach is the result of initial efforts partially funded by the European Commission under the FP6 ASSERT Project and further completed by ESA funding. TASTE is now a fully operational toolset that captures the system architecture with AADL, defines the data model with ASN.1, and finally combines heterogeneous components into an homogeneous software application to be uploaded on different targets up to the flight model. Different extensions are today on-going or planned by the community with the financial and technical support of ESA (Link to system modeling tasks, introduction of hardware components, and connection to development process support tools such as configuration management tools).

The choice of standard languages such as AADL and ASN.1, together with the wide openness of the tool implementation leaves open many doors for extensions to better cover all design steps from system requirements capture down to software deployment. First user feedback clearly indicates that TASTE does not everything a system designer may wish to have but provides a strong support to ensure consistency down to software deployment and reduces the risk of having tricky integration difficulties that generally impact the development schedule. The community born with ASSERT is now contemplating the different options to disseminate and possibly commercialize TASTE while keeping the effort steady to extend its capacity: the main goal is to push forward the current technological barriers and release the system designer creativity to develop new ambitious missions in the space domain within acceptable budget and quality envelopes.

7. References

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

[2] Project tools (download) and documentation: www.semantix.gr/assert

[3] Ellidiss (ASSERT GUI): www.ellidiss.com

[4] Cheddar scheduling analyzer: http://beru.univ-brest.fr/~singhoff/cheddar/

[5] Ocarina: http://ocarina.enst.fr/

9. Glossary

ASN.1: Abstract Syntax Notation One

SDL: Specification and Description Language

AADL: Architecture Analysis & Design Language