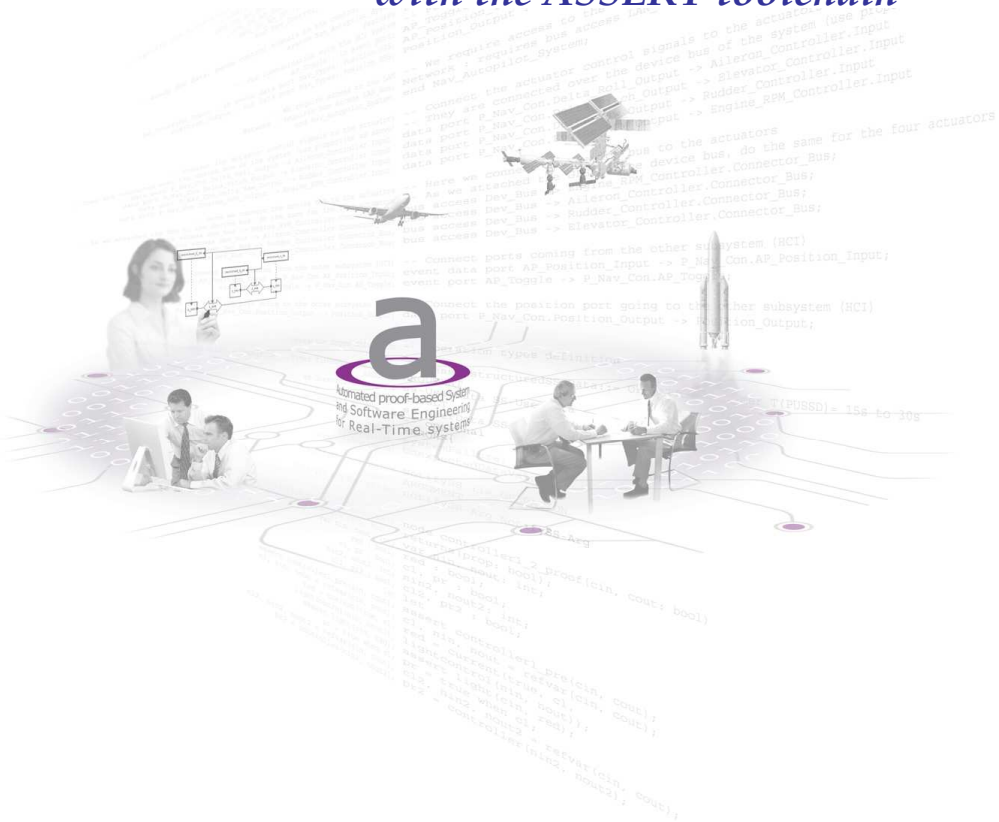
European Space Agency

esa

Design complex systems using multiple modeling tools with the ASSERT toolchain



Multi-Paradigm Model-based SW Engineering

Large scale software engineering often has to address issues related to heterogeneous development environments or run-time platforms. This is true for example when one tries to mix hand-written and automatically generated code, or when the system has to be deployed over a distributed architecture.

This short technical paper presents a new SW engineering process, along with its supporting tools, that enables the integration of systems whose components can be developed using disparate modeling technologies, as well as hand-crafted code.

A model-based approach to SW engineering is indispensable for producing robust, standards-adhering code especially in cases of complex developments where hand-crafted code is more expensive and more error-prone. Equally important, with a full model-driven code generation, the sources themselves become simply an intermediate artifact of the build process that's readily transformed to executable code. In this manner, the system is understood and analyzed solely on the model level, promoting understandability, maintainability and modifiability. Robust modeling notations and supporting tools are keys to actually transforming SW development from an art into a rigid engineering discipline. The proliferation of different and sometimes overlapping modeling notations (Simulink, Lustre) tools and initiatives (MDA, CORBA) in various domains of SW development reflects the importance that is attached to SW modeling from both industry and academia.

SW engineering is addressing multiple problem domains; for example, in a space system it is prudent to use synchronous, data-flow oriented languages (such as Scade or Matlab/Simulink) to define control laws and mathematical algorithms, and to specify behaviour of the system (orchestration of the calls, decisions based on results from the algorithms, fault detection and recovery, protocols, etc.) with state-machines (e.g. ObjectGeode/SDL). As might be expected, a single modeling notation / tool / methodology cannot be expected to be ideally suited to all cases. For small-scale projects this is usually not an issue as a single modeling methodology can be used for the scope of the entire development with good results. On the other hand, large-scale systems comprise multiple sub-systems often constructed by different teams. The requirements for each system are often disparate enough to warrant or even mandate use of different notations, tools or methodologies.

In such a scenario, interfacing with and integrating all the independently developed subsystems into a comprehensive whole, is a technically daunting task. What's more, currently, this can only be done with hand-crafted code thus negating many of the advantages of a model-driven approach. In other words, the reliability of the independent subsystems may be excellent since the code was automatically generated but the links between the subsystems are implemented with error-prone hand-crafted code. Additionally, while it might be possible to reason about verifiability or schedulability properties for each of the independent subsystems the lack of an overall model spanning the entire system does not allow similar analysis at the system level.

This short paper describes a system-level SW engineering methodology to tackle the above issues. The methodology was originally conceived in the context of the ESA-led **ASSERT** Integrated Project (in the FP6 IST programme) and has been subsequently elaborated and fleshed-out with additional tool support in the course of follow-up GSTP ESA contracts. SEMANTIX had a pivotal role in all these projects and owns a large part of the IP related to the tool-chain that supports the methodology.

In what follows, we present the methodology and then describe in more details the tools that support it.

The ASSERT Methodology

In a nutshell, the conundrum faced by **ASSERT** was how to allow the teams developing individual components to choose the modeling methodology that best addresses the respective component's requirements and at the same time ensure seamless, model-driven integration at the system level as shown in Figure 1.

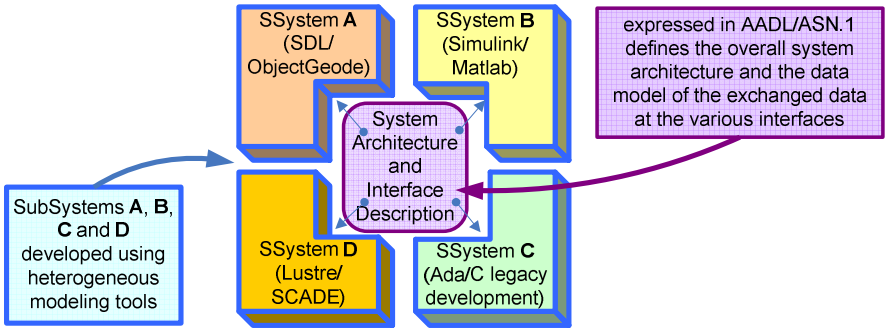


Figure 1: **ASSERT** context

Note, with respect to Figure 1, that individual components are developed using modeling notations that support behavior modeling. In contrast to that, the system-level logic in the centre is more geared towards interfacing between the subsystems. As such it is more amenable to be described by a notation that is more oriented towards the description of data and the overall system architecture than towards describing actions. The recognition of this dichotomy of dynamic (i.e. behavioral) versus static (i.e. data and architecture oriented) modeling notations is essential to understand the **ASSERT** approach.

ASSERT introduced a top-level model of the overall system that describes the individual subsystems and their interfaces in an abstract manner, which is modeling-tool independent. This was achieved by utilizing two notations:

- **AADL** (Architecture analysis and design language) in its graphical form is used to describe the high-level system model, i.e. the subsystems, their non-functional properties (e.g. WCET, the worst case execution time, etc) and their interfaces. It is also used to describe the system's deployment configuration (e.g. which subsystem is assigned to which CPU/network address).

- **ASN.1** (Abstract Syntax Notation number One) is used to describe the messages exchanged between subsystems in a language and modeling tool independent format.

The **1st** step in the **ASSERT** methodology is to model the overall architecture of the system and the data that are exchanged at the interfaces of the subsystems. Once this is done (using AADL/ASN.1), the **2nd** step is to "mediate" this system-level interface and data model to subsystem-specific data models (for the data relevant to each subsystem). This is depicted in Figure 2.

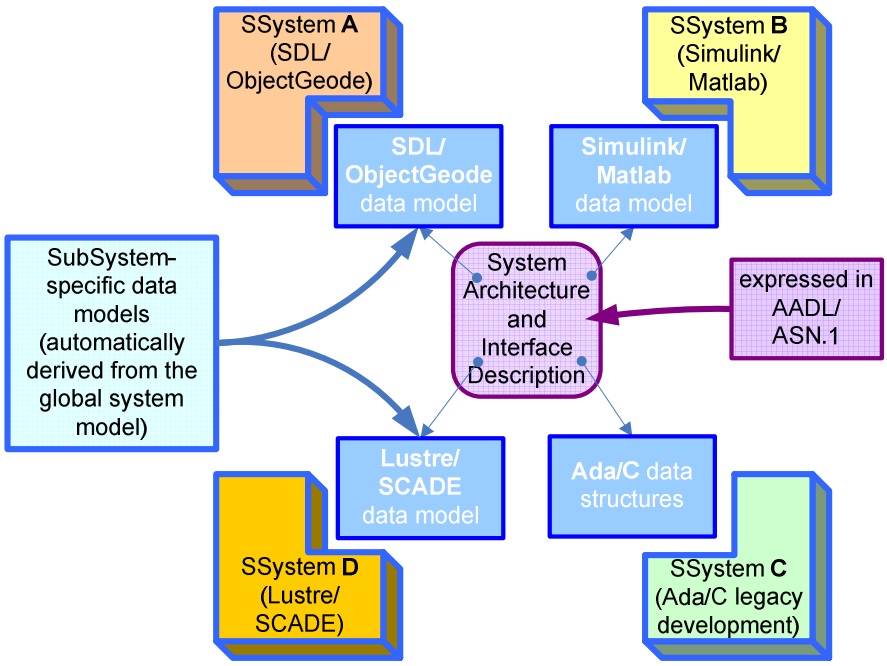


Figure 2. Mediating to tool-specific data models.

Note that for each sub-system only the relevant data structures need to be mediated. I.e., only the information that is exchanged at that subsystem's interfaces needs to be re-modeled from AADL/ASN.1 into the subsystem's modeling tool of choice. Note also that this re-modeling is fully automated and tool-supported.

This modeling mediation accomplishes two things:

- it ensures that the various subsystems are modeled in a consistent way as far as their interfaces and data are concerned. Even though different subsystems will model the same date structure in different ways (e.g. in MATLAB/Simulink, SDL or SCADE), all these data models will trace their ancestry from a common data model. In others words, each of the subsystems will use equivalent data models for the data structures it exchanges with the other subsystems even though they are expressed in different notations. This avoids integration problems while at the same time allowing maximum latitude as to the choice of the modeling notation / tool that is best suited for each subsystem.
- it provides a starting point for the modeling of the various subsystems in the form of the interface and data model of relevant data structures and communicating subsystems. Moreover, this is done in each subsystem's native modeling notation and is integrated into the respective tool's graphical environment.

Using the architecture and interface/data model of the system - in the native notation - as a springboard, each subsystem's behavior and internal data structures are subsequently modeled. Once a subsystem's entire modeling effort is concluded, the code for the respective sub-system is generated by the respective tool as depicted in Figure 3. This is the **3rd** step of the methodology.

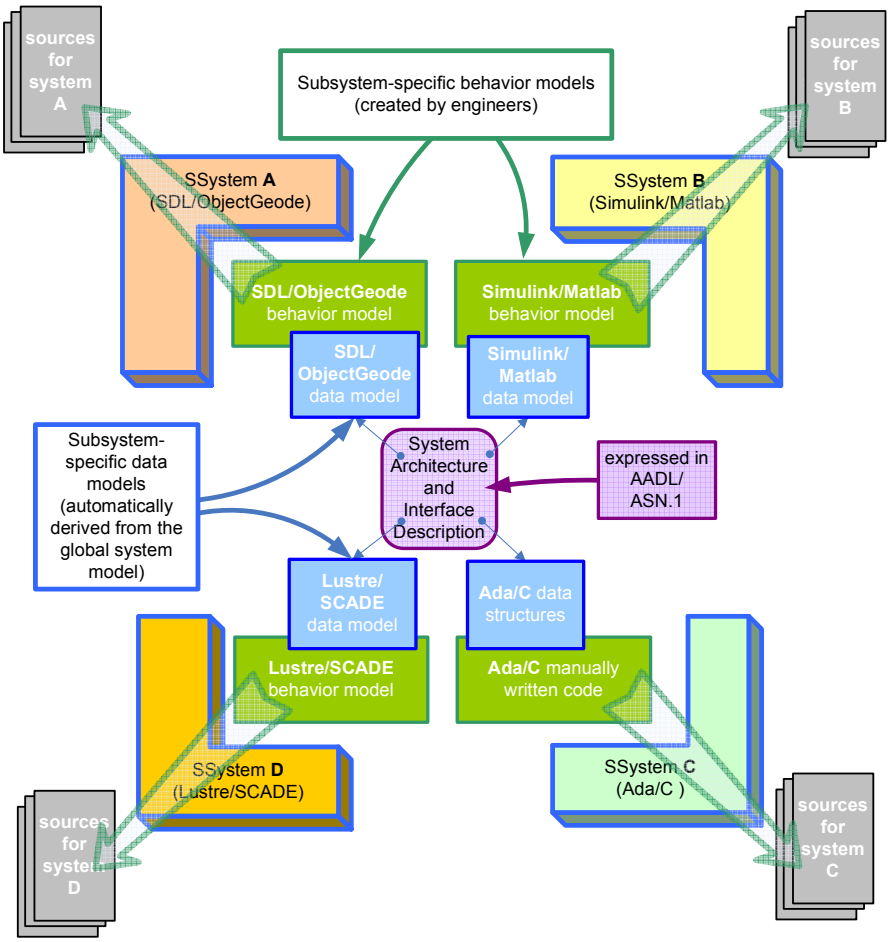


Figure 3. Behavior modeling (by human engineers) builds upon the automatically generated data models and concludes with the generation of sources.

It should be noted that the methodology can also accommodate subsystems that are implemented by hand (without using any modeling tool for the generation of sources). It also handles the more frequent case of subsystems that are built with a hybrid approach whereby part of the code is automatically generated from the models (e.g. skeleton implementations) and the rest is fleshed out manually.

Once the implementations for the various subsystems are available, the next, **4th** step of the methodology is simply the integration of the subsystems. The challenge at this step is to ensure that the subsystems can communicate effectively according to the system and interface architecture laid out in the **1st** step. Different tools (even for the same notation) use different code generation patterns and different strategies to represent the data structures that have been modeled in them. Therefore, the methodology foresees code-mediation whereby data "bridges" are automatically generated in order to mediate (at runtime) between the data structures used by the various subsystems. This effectively allows a subsystem modeled in Lustre whose code is generated using SCADE to exchange data structures with a subsystem modeled in SDL whose code is generated using ObjectGeode.
At first inspection, this approach points to a fully-meshed graph whereby automatically generated bridges bilaterally mediate between the data structures exchanged from one component to another. However, such a fully-meshed approach is cumbersome for a number of ways. Instead, a star topology is used whereby the automatically generated bridges mediate between each modeling tool's native data structure format to an ASN.1 format. In this approach ASN.1 functions as the "hub" of the star topology. This is depicted in part (b) of Figure 4.

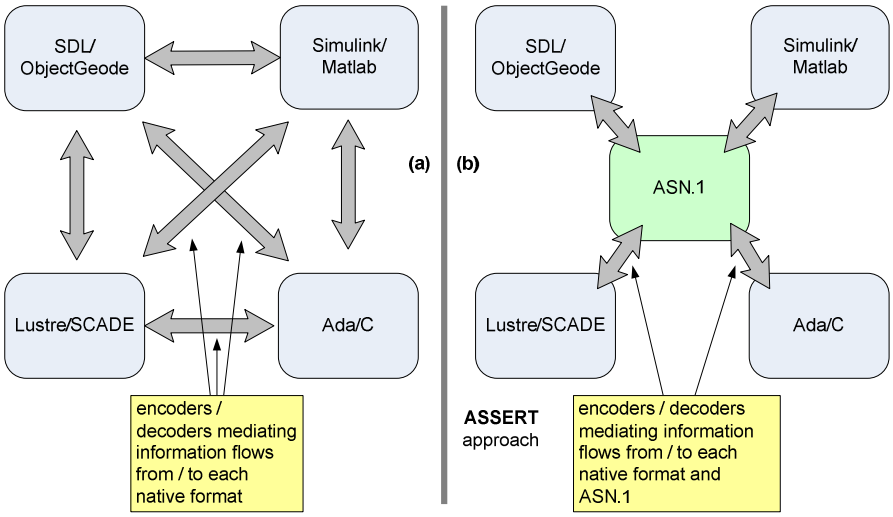


Figure 4. **ASSERT** uses ASN.1 as the core encoding mechanism for mediating between different internal formats.

Note that the exchange of messages between the various subsystems (mediated in ASN.1 format according to Figure 4, part b) takes place over a distributed processing infrastructure which is not depicted in Figure 4. In **ASSERT**, this infrastructure is called "**ASSERT** VM" and is providing functionality similar to that of a CORBA ORB.

Note also that all the "mediators" (the grey arrows in Figure 4) are automatically generated from the AADL/ASN.1 system model. These mediators perform two functions:

- they implement the conversion of the data structures that are exchanged at the interfaces of the various subsystems between their native representation (e.g. as Simulink data structures) and their equivalent representation in ASN.1.
- they are responsible for routing information and invoking methods across the integrated system by using the underlying distributed processing infrastructure (the **ASSERT** VM).

These automatically generated mediators provide in effect the glue that integrates the subsystems together.

The 4-steps of this methodology are depicted in Figure 5 for a hypothetical system that is comprised of three subsystems:

- subsystem A: modeled in Lustre/SCADE
- subsystem B: modeled in SDL/ObjectGEODE
- subsystem C: modeled in Matlab

1. The process begins with the overall system specification in AADL and ASN.1. At this point, the designer simply defines the subsystems that compose the overall system and their interfaces. Interface definition includes the ASN.1 specifications of the messages exchanged between subsystems. This is where details about types and constraints of message members are specified. It also includes non-functional attributes of the interfaces (like periodicity). Figure 6 depicts a typical view of the graphical system modeler.

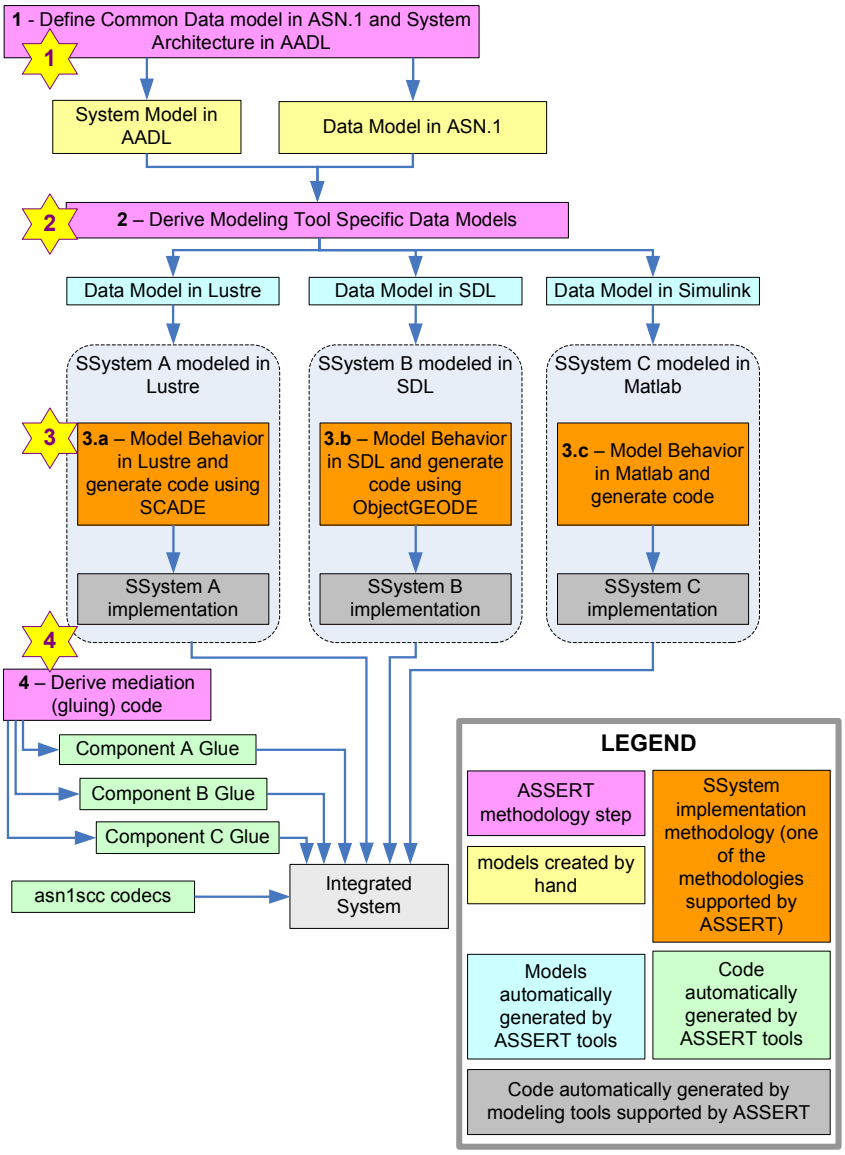


Figure 5. Schematic of the **ASSERT** methodology.

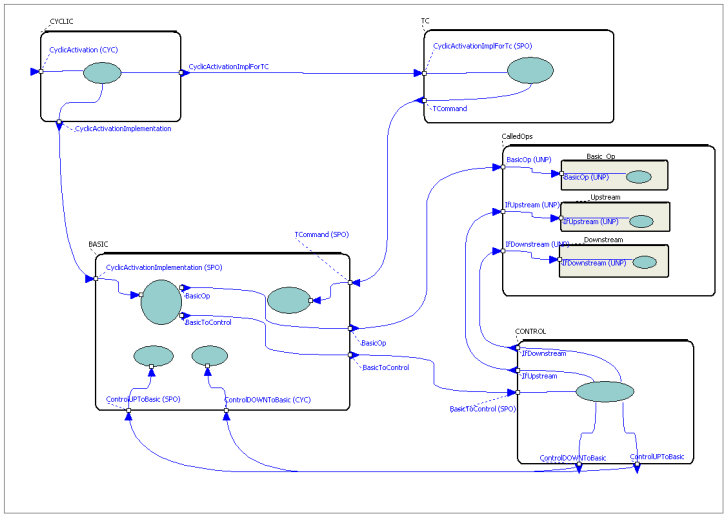


Figure 6. System architecture interfaces modeling in AADL/ASN.1

2. The system specification is processed by the **ASSERT** toolchain, and semantically equivalent definitions of the data messages are created for each modeling tool's language (e.g. Lustre definitions for SCADE subsystems, Matlab definitions for Matlab/Simulink subsystems, etc). This is shown in Figure 7. This way, the teams building the individual subsystems know that their message representations are semantically equivalent and that no loss of information can occur at subsystem borders. At the same time, modeling tool-specific project skeletons are automatically generated, allowing the subsystem developers to easily start working in their modeling tool of choice.

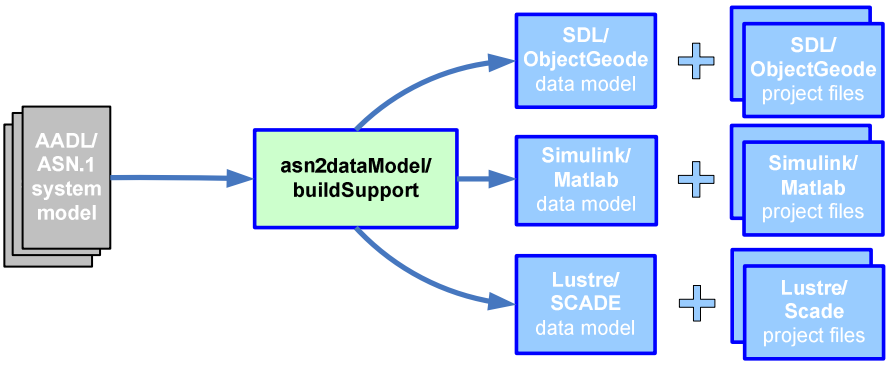


Figure 7. Generation of semantically equivalent models for the exchanged data in various modeling notations/tools.

3. When functional modeling is completed, the modeling tools' code generators are invoked, and C code is generated. Modeling tools generate code in different ways. Even though (thanks to step **2**) the data structures of the generated code across different modeling tools are carrying semantically equivalent information, the actual code generated cannot interoperate as is; see Figure 8.

```
// 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 8. Different tools generate syntactically incompatible representations for semantically equivalent code.

Therefore, integrating the code generated by different modeling tools requires "data bridges" to be built that translate the data structures from one modeling tool to those of the other and vice versa. Manually creating these data bridges is a very error-prone process, and one that would have to be repeated if the messages are even slightly changed.

4. To solve this integration problem, ASN.1 is used as the center of a star formation amongst all modeling tools. Data bridges are created automatically by the toolchain's code generators that translate the data at runtime between (a) the data structures of the modeling tools and (b) the data structures generated by the space certifiable ASN.1 compiler (Asn1scc) built by SEMANTIX. In this manner, any modeling tool can interoperate with any other, via ASN.1 encodings. In parallel, Asn1scc is invoked to create the necessary ASN.1 encoders and decoders, for all the messages. Code from the ASN.1 compiler (Asn1scc), code from the modeling tools and code from the "data bridges" are compiled and linked together.

Tools

The build process described above is supported by a collection of tools.

Model and Code Integration Tools

The build process described in the previous chapter makes use of the following integration tools:

asn2aadlPlus : translates the ASN.1 specification of the messages exchanged between subsystems into the corresponding AADL definitions, which are then imported by the system modeler. The system designer can then use it to visually perform the overall system modeling in AADL (see Figure 6).

asn2dataModel : translates the ASN.1 specification into the semantically equivalent modeling tools notations. Currently, this tool supports SCADE5/Lustre, SCADE6/Lustre, ObjectGeode/ SDL and Matlab/Simulink notations. To assist with legacy development, the tool also generates C and Ada data definitions.

buildSupport : creates skeleton project files in modeling tool-specific formats based on the overall system model in AADL/ASN.1.

aadl2glueC : creates the "data bridges" that translate at runtime the data between (a) the data structures of the modeling tools and (b) the data structures generated by our space certifiable ASN.1 compiler.

ASN.1 Space Certifiable Compiler (Asn1scc)

Asn1scc (ASN.1 Space Certifiable Compiler) is an ASN.1 compiler built by SEMANTIX. Asn1scc parses a subset of ASN.1 (International Standard 8824-1) and generates unaligned PER encoders and decoders for the C programming language that make (a) no calls to malloc() or to any other function that allocates memory dynamically and (b) no system calls, thus creating perfectly portable code.

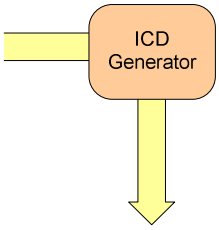
Asn1scc achieves the "no dynamic memory" goal by applying two simple principles: (a) The generated C structures do not contain any pointers (b) All fields are either primitive types or static arrays or other structures that don't include any pointers. The maximum number of bytes required for encoding any ASN.1 type in unaligned PER is determined by Asn1scc during the grammar processing and is available via a generated macro. In this manner, the size of all required memory, both for the C data structures as well as the buffers where the PER streams will be written, is known at compile time and consequently, all memory can be reserved statically.

Automatic ICD generator

An Interface Control Document (ICD) is a document identifying interface data that is exchanged between software components. ICDs typically allow a visual overview of the messages that are exchanged between applications. Such documents are authored manually after a cumbersome inspection of the exchanged data structures and after also accounting for the marshalling logic that will transform them to the wire format. Therefore, the process of creating an ICD is error-prone and updates of these documents are awkward, especially in projects with big data models that change often.

The automatic ICD generator is a tool that takes as input a formal data model in ASN.1, as defined in the overall system model, and produces the equivalent ICD automatically (Figure 9). The generated document contains tables for each type defined in the ASN.1 data model and each table has as many rows as the number of the ASN.1 fields (see figure below). The minimum and maximum numbers for each field represent the number of bits required to encode the specific field using the unaligned Packed Encoding Rules (uPER).

```
MY-MODULE DEFINITIONS AUTOMATIC TAGS ::= BEGIN
        MySequence ::= SEQUENCE {
                -- number of components
                field1  INTEGER (5..1000),
                -- number of values
                field2  INTEGER (1..16384) OPTIONAL,
                -- valve flag
                field3  BOOLEAN,
                -- operational mode
                field4  MyChoice,
                -- description
                field5  OCTET STRING,
                -- vector coordinates
                field6  Vector3 OPTIONAL
        }
```



MySequence(SEQUENCE) ASN.1				min = 3 bytes		max = ∞ bytes	
No	Field	Comment	Optional	Type	Constraint	Min Length (bits)	Max Length (bits)
1	Preamble	Special field used by PER to indicate the presence/absence of optional and default fields. ▪ bit0 == 1 ⇒ field2 is present ▪ bit1 == 1 ⇒ field6 is present	No	Bit mask	N.A.	2	2
2	field1	number of components	No	INTEGER	5..1000	10	10
3	field2	number of values	Yes	INTEGER	1..16384	14	14
4	field3	valve flag	No	BOOLEAN	N.A.	1	1
5	field4	operational mode	No	MyChoice	N.A.	2	∞
6	field5	description	No	OCTET STRING	N.A.	8	∞
7	field6	vector coordinates	Yes	Vector3	N.A.	24	312

Figure 9. ICD Generator creates an browsable Interface Control Document based on the ASN.1 grammar of the exchanged messages.

The ICD generator can be used early in the design process - as soon as the overall system model in AADL/ASN.1 is completed. This provides ICDs to the development teams, thus enabling those teams that choose to work with legacy developing techniques to interoperate with the rest of the subsystems (the ones developed in modeling tools).

Automatically generated GUIs for TM/TC

In the overall AADL system design, the designer can specify the subsystems for which a graphical user interface should be created. The 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. Telemetry can then be piped to plotting and monitoring applications, for easy real-time monitoring of systems.

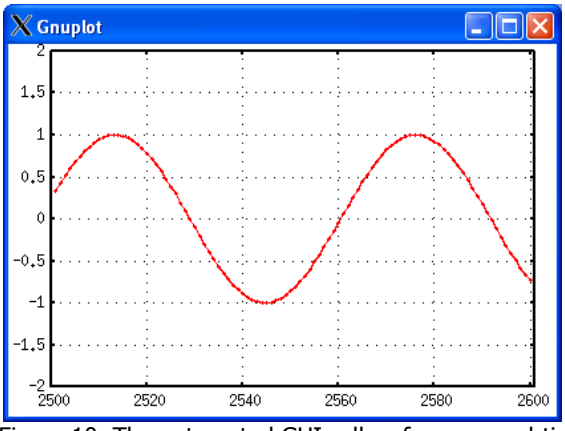


Figure 10. The automated GUIs allow for easy real-time monitoring of systems

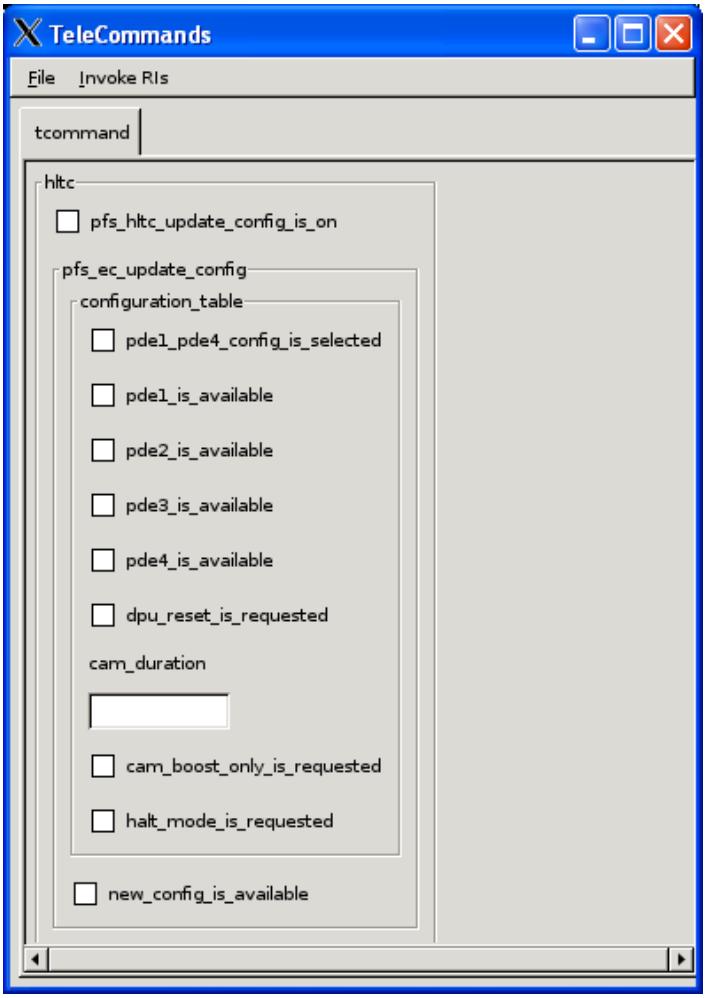


Figure 11. Users of the automated GUIs can invoke Telecommands and read Telemetry in real-time.

Conclusion

The ASSERT methodology along with its support tools significantly improves the design process of complex systems. It was field tested in two multi-modeling tool scenarios during the development of the ASSERT project, with the MA3S/PFS Pilot Project being the most prominent one. The resulting binaries were successfully downloaded and executed on the embedded LEON processors that ESA is using, proving in a conclusive manner the feasibility of multi-modeling tool development.

Download links

The latest version of the toolchain, Asn1scc and the ICD generator is available from the following link:
http://www.semantix.gr/assertTools/

For more info

ESA:
Maxime.Perrotin@esa.int
SEMANTIX:
assertTools@semantix.gr