Experience in the Integration of Heterogeneous Models in the Model-driven Engineering of High-Integrity Systems

Matteo Bordin, Thanassis Tsiodras, and Maxime Perrotin

University of Padua, Department of Pure and Applied Mathematics,
via Trieste 63, 35121 Padua, Italy
mbordin@math.unipd.it
Semantix Information Technologies, K. Tsaldari 62, 11476, Athens, Greece
ttsiodras@semantix.gr
European Space Agency, Keplerlaan 1, 2201 AZ Noordwijk, The Netherlands
maxime.perrotin@esa.int

Abstract. The development process of high-integrity systems has shifted from manual coding to designing with modeling tools that verify the correctness of the design well before production. The parallel application of several different modeling tools in the design of separate parts of the same system is now a common industrial practice. The advantage of using several, domain-specific tools is however balanced by an increasing complexity in the integration phase: it is indeed necessary to guarantee the correctness of the interaction of the several subapplications, which also includes the integration of the source code automatically generated by the different modeling tools. This constitutes a major concern for the introduction of several modeling tools in the industrial community, as well as for certification institutes. In this paper we present our practical experiences in the definition of a computer-aided sound development process to permit model-driven integration of heterogeneous models.

Keywords: Model-driven Integration, High-Integrity Systems, Automated Code Generation.

1 Introduction

The development of high-integrity systems stands to gain much from exploitation of different tools and languages in the implementation of a component based system. For example, in the domain of space-related applications (which is our main domain of interest), the system implementation is usually co-developed by several different providers, each one using a tool specifically suited for a particular subset of the application: Matlab [1] for the implementation of algorithms, SDL [2] for state machine logic, UML2 [3] for object-oriented architectures, AADL [4] or the emerging SysML [5] for system modeling, etc. The use of domain-specific tools offers two main advantages: (i) domain-specific semantics greatly simplifies the design and verification of a precise kind of applications; and (ii) the mentioned tools usually provide for automated source code generation through a specifically tailored process (for example, the code generator which comes with SCADE has been qualified for DO-178B level A systems [6]).

The exploitation of several, domain-specific tools providing automated code generation to implement the functional specification surely decreases the verification and validation costs; but it also increases the criticality of the integration phase, as the switch from *software* modeling to *system* modeling - which is the design of the system architecture - may negatively affect the semantics of common data types and the non-functional properties of the system. Currently, the integration process is handled manually, making it very error-prone and a possible source of defects.

The idea we present in this paper is to exploit model-driven technologies to automate the integration phase, guaranteeing that the whole process can be applied in domains subject to strict certification standards. Model-Driven Engineering (MDE, [7]) is currently one of the main innovation vectors in software engineering. The whole idea at the heart of model-driven engineering is to promote the use of a formal, high-abstraction, representation of the system, a *model*, during each phase of the development cycle. In a model-driven development process, models are designed, analyzed, transformed, verified and executed. The notion of *model transformation* is particularly meaningful in MDE. Models are usually designed at a very high abstraction level, which may be agnostic on aspects such as target execution platform, deployment, and distribution middleware: a model transformation which takes as input the model *and* the platform specification may automatically generate the implementation of the system for a particular platform. Ideally, the developers do not need to cope with low-level representations of the system at all: the generation of source code is for example *just one* of the several possible transformations a model is subject to. In mainstream software engineering, model-driven engineering has *de facto* taken the name of the OMG initiative named Model-Driven Architecture (MDA, [8]); MDE is however not limited to the OMG world: SCADE or Matlab Simulink [9] are indeed excellent examples of MDE infrastructures because they permit to design, verify and deploy systems using a high-abstraction modeling semantics. Another key aspect of model-driven engineering is the concept of domain-specific metamodeling (DSM), which is the definition of design languages and tools to fully support MDE in a particular domain.

Model-driven principles and technologies have already been applied to the integration problem [10]: the OMG MOF facility is by itself an integration framework for heterogeneous metamodels. The most common domain for the application of model-driven integration is enterprise computing. One of the most typical application is the reverse engineering of legacy components (usually in the from of source code, CORBA interfaces, XML) to UML models, so as to permit the generation of a middleware layer to interface the components [11]. Another common application is the interoperaction between metadata defined with domain-specific metamodels which are all traced to a common metamodel, usually in the form of UML profiles [12].

Our first main contribution to the field of model-driven integration is the domain we target: high-integrity systems, in particular in the space domain. The most well-known (and probably unique) example of model integration in the high-integrity domain is the SCADE Simulink Gateway which permits to import Simulink models in SCADE, modify them and generate code using the SCADE code generation engine. As a first notable difference, we aim to integrate models designed with *radically* different tools *along* with their generated code. The models we plan to integrate are *functional* models because they represent the functional (sequential) specification of the system. The integration process must thus guarantee that the interaction between the models designed with different tools does not corrupt the properties proved during the modeling phase and that the interfacing of the generated code does not corrupt the semantics of the exchanged data: the solution of this last problem cannot be found in mainstream technologies like CORBA, SOAP or WEB-services, because of the peculiarities of the target domain (embedded systems with strict performance and predictability requirements). Finally, we also wish to verify system-level properties (in particular the timing behaviour) of the integrated system via model-based analysis: to achieve this goal, the integration process must be able to extract the information relevant to the analysis from the imported models.

1.1 The Overall Picture

The work presented in this paper is part of a toolchain infrastructure for the design, verification and implementation of high-integrity real-time systems. The main aim of our work is to define a new development process for high-integrity systems and develop a set of tools to support it. We have already developed a full Eclipse plug-in for the design of high-integrity, real-time systems. The plug-in is based on a domain-specific metamodel called RCM [13]. The RCM metamodel is conceptually traceable to a UML2 profile: it allows functional modeling by means of class and state machine diagrams and system modeling through component and deployment diagrams. The RCM metamodel guarantees that any designed system abides by the constraints enforced by the Ravenscar Computational Model [14] and can thus be statically analyzed for its timing behavior. The timing analysis is automatically performed on the model itself and encompasses logically and physically distributed systems [15]. The plug-in also comes with an automated code generation engine targeting Ada 2005 [16], which achieve 100% code generation for the concurrent and distributed architecture of the system [13].

2 System Models as an Integration Framework

The notion of *heterogeneity* entails that functional models are defined with different semantics: in MDE terms, we would say that the metamodel underlying each model is potentially different. This is in fact the case, as tools like SCADE, SDL and UML have their own semantics; the same fact applies also to manually written code, as a programming language - or more accurately, its grammar and semantics - is by itself a metamodel.

The place where the (different) semantics of (heterogeneous) functional models fit together is the *system model*, which is the model representing the overall system in terms of both functional and non-functional (like concurrency and deployment) features. We believe that a system model should be considered as a true integration framework — a pivotal representation of the system — because this is the place where software and system semantics merge: only in a system model is it possible to assure the correct interaction between integrated models and verify system-level properties which are affected by both the software and system modeling process. Our metamodel of choice for system modeling is the RCM metamodel, briefly introduced in section 1.1 and described in [13] and [17].

Merging heterogeneous functional models within a single system model may render particularly challenging to:

1. verify that the functional models do not interfere with the synchronization mechanism of the concurrent architecture;
2. assure that the concurrent architecture does not corrupt the properties of functional models by introducing race condition or deadlocks;
3. guarantee that the (possibly remote) interactions between heterogeneous functional models are semantically preserving, which basically means they do not corrupt the passed data;
4. keep the software and system view consistent with each other.

Coping with items 1 - 2 above is straightforward. If the concurrent semantics is prohibited in functional models, then the functional specification cannot affect the synchronization of the system; it is quite easy to identify the elements (or key words in a programming language) related to concurrency in the metamodel for functional modeling and prohibit their use. Tools for functional modeling usually permit to express some sort of aggregation properties for services accessing the same functional state: a SCADE block or a UML class are such examples. In order to avoid the corruption of the functional specification by the concurrent execution of the system, it is enough to constrain the concurrent semantics to permit at most one task at a time to access a functional model — in other words, to have a single executer behind the state machine underlying the functional model[1]. This constraint leaves two possible choices for stateful functional models: (i) a single dedicated task always executes the state machine; or (ii) each triggering procedure of the state machine presents a synchronization protocol assuring mutual access to the whole functional state. Stateless functional models (for example mathematical functions) do not require any particular attention in deciding their concurrent behavior. By choosing the concurrent semantics in a way the aforementioned constraints are guaranteed, no race condition may happen; deadlocks can also be prevented by enforcing the immediate priority ceiling protocol [18] and implementing remote communication with an asynchronous, message-based protocol.

[1] We do not consider non-intersecting, parallel state machines within the same class.

To guarantee semantic preservation of exchanged data (point 3) we use ASN.1 [19] for data modeling and an appropriate compiler to generate stubs to convert the raw representation of data between different languages/architecture: it is however less clear how we can integrate with the code generated by different modeling tools (SCADE, SDL, etc.). View-consistency (point 4) requires more attention and is strongly related with *cross-cutting* concerns. Cross-cutting concerns are aspects of the system affected by both the software and the system modeling process. A typical example of cross-cutting concerns is the definition of (execution/information/data) flows: they of course depend on the connection between component instances and on their deployment; but they also depend on the functional specification, which basically tells which services are invoked in response to the execution of a functional procedure. A sound determination of flows is a fundamental requirement for several kinds of model-based analysis, it being related to, for example, timing performance or security preservation. In the scope of our experimentation, we used the modelization of execution flows to perform model-based timing analysis [15].

3 Semantic Preservation in Practice

Three main dimensions require particular care if several modeling tools are exploited: (i) the semantics of common data types, and in particular their physical representations on different execution platforms; (ii) the integration of the code generated by the different modeling tools; and (iii) the extraction and evaluation of cross-cutting concerns.

3.1 Data Semantics Preservation

Abstract Syntax Notation One [19,20] (ASN.1) is a standard and flexible notation that allows detailed data structure representation, as well as portable and efficient encoding and decoding of data into bitstreams.

In the context of our work, ASN.1 was used as the center of a "star formation"; *all* the communication taking place between the subsystems (possibly modeled in different modeling tools) is done through ASN.1 messages. This enforces a common semantic "contract" between the communicating entities, in terms of what information is exchanged; ASN.1 therefore guarantees the semantic equivalence of the data types used in the different modeling tools.

To enforce this semantic equivalence contract, a semantic translation takes place immediately after the definition of the ASN.1 grammar that describes the exchanged data. The ASN.1 definitions of the messages form the basis; the desired target definitions are the semantically equivalent ones in the data definition languages offered by the modeling tools. A custom tool was built [21] that reads the ASN.1 definitions and translated them into the equivalent data definitions, to the extent supported by the target modeling tool languages (e.g. Lustre (for SCADE), SDL (for ObjectGeode), etc).

The translation process is guided from the overall system view; by parsing it and learning about the implementation platform of each subsystem, the translation tool is in a position to know the desired target language per message and accurately translate it preserving the semantic information.

As an example, from this ASN.1 data definition:

```
EXAMPLE DEFINITIONS IMPLICIT TAGS ::= BEGIN PosData ::=
[APPLICATION 1] SEQUENCE {
    x INTEGER, y INTEGER,
    description OCTET STRING (SIZE(1..80))
} END
```

we get this translation in Lustre:

```
let type System_Types
    PosData = [x : int, y : int,description : char^ 80];
tel;
```

This translation is in fact the key to guarantee semantic consistency; e.g. the team developing a subsystem in SCADE will use the structure definitions as they are generated from the translation tool, knowing in advance that this process will neither introduce new content nor prune existing ones. If there is information in the ASN.1 definitions that is not translatable to the target modeling tool language, the translation tool will complain and warn the user about it - providing early feedback about the potential loss of information and preventing side effects from this loss. Notice also that by using the overall system view, the tool knows exactly what targets it needs to generate code for, thus being minimal and complete - optimal - in the generated definitions.

3.2 Integration of Generated Code

Creating semantically equivalent definitions is the first step - it guarantees that all subsystems will be functionally modeled with equivalent message definitions. This is not enough, however. Each modeling tool follows its own scheme in terms of how it generates code. To be precise, the code generated by the tools can be conceptually split in two categories:

- Code that implements the logic of the subsystem: state machines, algorithmic descriptions of work to be done in response to incoming signals, etc
- Code that describes the data structures of the exchanged messages

Since the data definitions have been produced by the translation tool, the data structures generated are certainly equipped with the same data. The details however - e.g. variable names, ordering, language-specific type definitions, etc - vary a great deal between different modeling tools. As a consequence, the actual generated code cannot interoperate as it is; error-prone manual labour is required to "glue" the pieces together. This is the source of multiple problems[2], and it is

[2] http://awads.net/wp/2005/12/05/ten-worst-software-bugs/

another reason for using ASN.1: by placing it at the center of a star formation amongst all modeling tools, the "glue-ing" can be done automatically:

– An ASN.1 compiler is used to create encoders and decoders for the messages exchanged between subsystems [22]
– Another custom tool is used [21], that creates mapping code ("glue" code); code that translates the data at runtime between the data structures of the ASN.1 compiler, and the data structures generated by the modeling tools.

As long as the mapping is a well defined one - that is, as long as the modeling tools follow specific rules in how they translate message data into data structures - this mapping work is feasible *at compile time*. This translation tool starts from the overall system view, just as the first one (Section 3.1) did: it learns about all the "paths" that messages have to go across, and thus, it knows what kind of glue code to generate at the source and the destination of each message exchange.

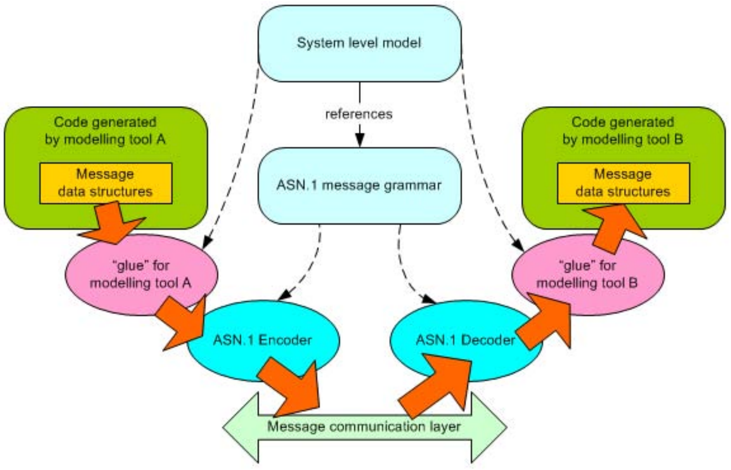
System level model
references
Code generated by modelling tool A
Message data structures
ASN.1 message grammar
Code generated by modelling tool B
Message data structures
"glue" for modelling tool A
"glue" for modelling tool B
ASN.1 Encoder
ASN.1 Decoder
Message communication layer

Fig. 1. Mapping data using ASN.1

This process is significantly easier to test and verify - compared to the manual translation process that would have to take place in its absence. Instead of painstakingly checking all the manually written code parts that marshal data back and forth between the data structures of the modeling tools' generated code, the only tests that need to be done are performed on the code generating primitives, that is; mapping of integers, mapping of floating point numbers, mapping of strings, etc. When each category has been tested and is known to be handled in the correct manner, *no further testing is necessary*, regardless of the complexity of the message structure itself. This significantly lessens the effort required to use complex messages in the exchanges taking place between different modeling tools.

Additionally, this glue layer offers a central point for tool-indifferent marshalling: common mapping API can be developed that pertain to specific type mapping and tool categories; as an example, SCADE/Lustre are just one of the technologies adhering to a synchronous modeling paradigm; common patterns for all such tools can be extracted into a message marshalling portability layer.

Here is an example declaration section from the glue code generated:

```
#include <stdlib.h> /* for size_t */ int
Convert_From_ASN1C_To_TCLink_In__TC_Parser__Id(
    void *pBuffer, size_t iBufferSize);
```

A number of marshalling functions are generated, one (or more) per message marshalling interface: they convey the message data as (pBuffer, iBufferSize) pairs into the appropriate data structures generated by the modeling tools; data are passed via (pBuffer, iBufferSize) pairs because it is a language-neutral representation of a series of octet - the ASN.1 message. Their implementation is completely automated, and their code generation process can cope with arbitrarily complex message definitions in a transparent way. The end user simply calls them, without ever worrying about the details of the mapping code.

3.3 Managing Cross-Cutting Concerns

Cross-cutting concerns (as the representation of functional provided/required services or the identification of execution flows) require the correct understanding of both the software and the system specification. A simple but illustrative example is the following. Given a state machine SM, suppose that the state entered by invoking method p() includes, among its actions, the invocation of method r() on object o (a class member). It is evident that there is a flow from p() to o.r(), but from the pure software specification it is not possible to determine which object resolves the invocation, because this information is contained in the deployment diagram where objects are linked; similarly the semantics of the invocation of o.r() is obscure in the functional model, as it may be synchronous or asynchronous, local or remote: this information is again contained in the system specification. On the other side, by looking at a pure system specification (it being written in RCM, SysML or AADL), it is not possible to determine the functional behavior behind the invocation of a service, as the action semantics is not completely visible from a system model. The current industrial practice would require the *manual* translation of cross-cutting concerns from a software to a system model (or viceversa): for example, flows may be manually identified in the system level, assuming the designer has a complete knowledge of the underlying functional specification. This process is inevitably error-prone: design errors may of course be caught during the verification phase, but this approach still requires additional manual intervention and thus increases the cost of validation. The presence of several possible modeling tools (each one with its own metamodel), makes it even harder for the designer to completely understand all possible formalisms for software design and extract the required information from the software models. Furthermore, to correctly identify cross-cutting concerns, the designer is actually required to define a semantic mapping between the metamodel used for system modeling and the metamodel(s) for software modeling. Industrial practitioners often seem to overlook the hidden complexity of that kind of mapping, which is definitely a potential source of inconsistency between system and software models.

In order to try to overcome the limits intrinsic to the manual nature of coping with cross-cutting concerns, we reasoned on the possibility of automating the interchange of semantic information between software and system models, and in particular to automatically import heterogeneous functional models in the system model. In section 2 we have acknowledged the pivotal role of the system model: our idea is to automatically determine the system representation of (part of) cross-cutting concerns by *importing* software models into the system model; in this manner, the design process would be guided by the automatically imported information, avoiding errors and inconsistencies between the system and software specification by construction. The import of a functional model is basically a model transformation which transforms a functional model conforming to the metamodel of the tool used to design it, to a model which conforms to the RCM metamodel: the first metamodel is said to be the *source* metamodel, while the second is the *target* metamodel. When importing a functional model, two distinct options are possible: we can import the *entire* semantics of the source model or we can extract *just* the information required to create a valid RCM model. In the first case, we should guarantee that the target metamodel (RCM) is expressive enough for every possible functional metamodel and develop a complex model transformation encompassing the *entire* semantics of the source metamodel: this solution may not even be possible to implement. In the case of importing a selected subset of the functional model, we are required to extract *only* the information that is needed for an RCM model: basically, the target metamodel specifies which kind of semantics must be present in the source metamodel to permit a meaningful import process. The required subset of semantics of the source metamodel is usually determined by the needs for system-level model-based analysis and code generation: in our case, to perform model-based timing analysis, we are interested in just provided/required services of each functional model and in their relation (basically, the execution flow: which required service is invoked during the execution of a provided one). In order to obtain the best cost/benefit ratio, we chose the second option.

The following step is the definition of the semantic mapping between the source metamodel and the RCM metamodel. From a purely conceptual point of view, a semantic mapping requires the comparison of the semantics of two different languages and the definition of a series of functions to move from the source to the target metamodel. Unfortunately, a mathematically sound methodology to specify *and* prove semantic mappings in MDE is yet to come: we thus still rely on the comparison of the language standards to define the model transformations. On a more pragmatic dimension, the model transformation requires to extract a set of information from a functional model, which may come in the form of a textual language (SDL, SCADE) or via an XML-based representation (UML2). If the source model is encoded using the latest (meta)modeling technologies, the model importer can exploit state-of-the-art tools to directly transform the source model into the target model. On the other side, importing a model specified via a textual language is more complex, because it requires a sort of "double-pass" transformation: first the model is parsed, then it must be transformed into an XML tree on which perform a model transformation.

The import process creates RCM entities representing the imported functional models within the functional view of an RCM model. Such entities are basically read-only, because they were designed, verified and deployed (transformed to code) with extern modeling tools. At the same time, the generated entities are marked with an appropriate tag to permit the RCM code generator to generate the code required to interface with the source generated by the original modeling tool. Once the software model is imported into an RCM model, the RCM representations of system-level provided/required services and possible execution flows are *automatically* determined out of the functional specification (see [17] for a complete explanation): it is thus impossible for system-level properties to be specified in a manner which is inconsistent with the imported software models. From this perspective, the RCM metamodel presents a clear advantage over other system modeling languages: it guarantees view consistency by strongly relating semantic element of each view. The RCM system model thus contains all the information required to perform model-based timing analysis: from this point on, the verification process proceeds as described in [15].

At the moment, we have developed prototype importers of UML2, ObjectGeode (SDL) and SCADE (Lustre) models into an RCM model: the UML2 importer is not particular interesting because, since RCM mimics the UML2 semantics, its development is purely a technical (not conceptual) exercise; SDL and SCADE importers are indeed worth of additional explanation. The tools we used to parse and transform SDL and SCADE models are, respectively, OpenArchitectureWare xText and the Atlas Transformation Language (ATL).

Importing SCADE models. In order to preserve the properties of SCADE blocks (verified and proved with the appropriate modeling tool), we decided to prohibit the invocation of extern operations from within a SCADE block. For this reason, a SCADE block is always the leaf of an execution flow: in other words, it does not present required interfaces. The importing process of SCADE block is thus quite simple: it is simply mapped as a RCM class providing the service(s) offered by the block.

Importing SDL models. The import of SDL models is more complicated because they may have both provided and required interfaces - meaning that it is necessary to extract not only provided and required services, but also the execution flow. After the parsing, the transformation process is divided into three main steps:

1. Each SDL process is mapped onto an RCM class in the functional model: the accepted input signal are mapped as public methods of the class.

2. The output signals or calls executed by the SDL process compose the required interface of the RCM class: they are grouped by their target element, which may be another SDL state machine, a SCADE block or an UML class.
3. For every output signal or call sent in response to an input signal, an execution flow between the method corresponding to the input signal and the required interface corresponding to the output signal or call is generated in the RCM class.

All information imported from the SDL model are automatically represented in the system view, thanks to the view consistency enforced by the RCM metamodel. To some extent, the SDL importer is still primitive because it does not take into account conditional execution: *all* possible execution flows are considered *at the same time*, even those which are mutually exclusive. For the purpose of timing analysis, this limit induces a clear pessimism, because the worst case execution path is composed by the union of *all* paths. To limit the pessimism, in the first prototype of our tool we permit to manually select which flows must be considered for the analysis: we are aware that a manual intervention may potentially corrupt the model consistency, but we consider this solution as a temporary defect induced by technical reasons, rather than by conceptual difficulties.

4 Results and Discussion

To evaluate our approach and the tools we developed, we designed a simple example using the RCM metamodel and related tools. The prototype is a simplification of the software architecture of a subset of the embedded software of a satellite, in particular the positioning and guidance and navigation system: it is composed by communicating applications designed in SCADE (algorithmical computations), SDL (state machines modeling) and RCM (system modeling). The prototype has been demonstrated during the final review of the ASSERT project (cf. the Acknowledgements section). The designed system is an approximation of a real-life architecture, but it demonstrates most of the components categories usually present in this family of applications; our purpose of evaluating our approach in model-driven integration of heterogenous models is adequately illustrated by this simplified prototype. Our evaluation is based on a set of metrics quite common in model-driven development, namely semantic preservation in model transformations, ease of model-based analysis, model-to-code traceability and the quality and size of generated code.

The prototype importer tools developed to generate RCM functional models from SCADE and SDL models enabled us to accurately determine the system-level representation of provided and required services; at the same time, possible execution paths are identified during the importing process, permitting a safer identification of the flows of interest for model-based (timing) analysis: with the described approach, the identification of cross-cutting concerns cannot be a possible cause of semantic inconsistency anymore. The presence of an XML-based and well-defined metamodel for all involved modeling tools is a highly desired requirement to simplify any importing process by using more productive modeling technologies: modeling technologies using XML and OCL (or equivalent) -based technologies render the development of the model importer/analysis tools easier and more cost effective. The effort we spent in developing an SDL importer (starting from a textual specification) is a practical indication of the truth behind this statement: state-of-the-art modeling technologies permit much easier model query and manipulation, thanks to the exploitation of domain-specific tools; as an exemplary quantitative evaluation, *just* the implementation (and *not* the conception of the semantic mapping) of a UML2 importer from an EMF-based implementation took us less than half of the time than the development of the corresponding tool for SDL. Our belief is strengthened by the general industrial trend toward some sort of XML-based metamodeling technology, even for languages originally born as a pure textual specification such as SCADE or AADL (cf. the TOPCASED project).

Our automated code generation process proved to be useful and efficient when applied to the described test case: the code generated from SCADE and Object-GEODE (an SDL tool) could be *automatically* integrated with the code generated from the system view (designed in RCM) to implement the concurrent and deployment architecture of the system. From a quantitative point of view, the amount of generated code is comparable in sheer size to the source generated to handle the concurrent and distribution infrastructure, but probably not more than what we would have written in a manual development process: such an evaluation is a good empirical estimation of the productive advantages of the developed tools. We are currently working to integrate the overall transformation chain (including the code generator for RCM models and the tools described in sections 3.1 and 3.2) within a single Eclipse plug-in: in this manner, we plan to decrease the effort required by the end user to generate code integrating heterogenous models within a single system model.

From a purely technological point of view, the results we achieved are quite important, since they represent one of the first (if not *the* first) successful attempt to apply model-driven integration in the space application domain: our test case — while simple — is a valid proof of concepts and exploits tools widely used in the industrial community. Some concerns however still remain.

First, some optimization concerns: while ASN1 modeling is surely an effective way to guarantee the preservation of the semantics of data types across different languages/architectures, when the interacting subsystems are designed in the same modeling tool and they "live" in the same process space, they can communicate more optimally (speed-wise) by directly accessing each other's data structures. This would avoid the overhead of needless data conversions. On the code generation side, the choice of *always* passing through ASN1 (un)marshalling has two main drawbacks: (i) it induces a penalty on the execution time: the penalty is not evident in the model and cannot be evaluated on the functional specification (it is introduced by the code generator), making it difficult to perform accurate model-based timing analysis; and (ii) it makes model-to-code traceability difficult, as the invocation of any required service is actually mapped as an invocation to a sort of middleware composed by the ASN1 (un)marshallers, instead of a proper method invocation like in the originating model. To partially overcome the cited problems, we may consider to extend our tools to apply ASN1 (un)marshalling only when strictly required and add traceability information to the generated code.

5 Conclusion

In this paper we have described an experimental approach to model-driven integration for the development of high-integrity systems exploiting multiple modeling tools. By identifying in the system model the place where heterogeneous models should be integrated, we developed a set of tools allowing a highly automated integration process encompassing model importing and automated generation of glue code. The main difference of our approach when compared to mainstream solutions is its focus on integrating radically different models *and* their generated code, with particular attention for the consistency of cross-cutting concerns and the verification of system-level properties in the *integrated* system model: the integration process indeed also includes the extraction of information relevant to model-based analysis from the imported models. During our investigation, two main results rose. First of all, multiple-view consistency emerged as a highly desirable property for system modeling languages aiming to support model-based analysis in the high-integrity domain: contrary to the RCM metamodel, current state-of-the-art modeling languages do not enforces any form of view consistency, forcing the designer to manually guarantee it. In addition, selective[3] model import via automated model transformations showed to be a worthy solution for analysis-oriented model-driven integration.

The industrial need for the developed technologies is strongly related to the heterogeneity of modeling tools/platforms/architectures for the domain of interest: the more the variety, the more useful our tools are. In *current-generation* systems the weight of the side-effects introduced by the chosen technological solutions is not small, in particular for what regards model-based analysis, model-to-code traceability and performance; *next-generation* applications are however expected to drastically increase their complexity, along with the amount of exploited modeling formalisms and programming languages: the recent rise of AADL, SysML and RTSJ are a clear example of this trend. We thus expect the integration issue to gain more and more importance in the development of future systems; the industrial community must then strive to find effective and cost-wise solutions to solve it: the approach we presented in this paper is a good starting point in that direction and surely a valid reference milestone for future improvements.

Acknowledgments. The research work from which this paper has originated was carried out in the ASSERT project (IST-FP6-2004 004033) partially funded by the European Commission as part of the 6th Framework Programme. The authors gratefully acknowledge Yuri Yushtein (European Space Agency) for his valuable suggestions.

[3] Only part of the imported models is mapped onto the target metamodel.

References

1. Matlab: `http://www.mathworks.com/`
2. SDL: Specification and Description Language, `http://www.sdl-forum.org/`
3. OMG: UML2 Metamodel Superstructure (2005)
4. AADL: Architecture Analysis and Design Language, `http://www.aadl.info`
5. SysML: Systems Modeling Language, `http://www.omgsysml.org/`
6. RTCA: Radio Technical Commission for Aeronautics, `rtca.org`
7. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. Computer 39(2), 25–31 (2006)
8. The Object Management Group: (Model Driven Architecture), `www.omg.org`
9. Mathworks: (Simulink), `http://www.mathworks.com/products/simulink/`
10. Denno, P., Steves, M.P., Libes, D., Barkmeyer, E.J.: Model-driven integration using existing models. IEEE Software 20(5), 59–63 (2003)
11. E2E: Model Driven Integration: Transparent Virtualization of Distributed Applications (E2E technical white paper), `http://www.e2ebridge.com/live/files/E2E-WP-MDI-070112en.pdf`
12. Noogle, B.J., Lang, M.: Model Driven Information Architecture. TDAN.com (2002), `http://www.tdan.com/view-articles/4989`
13. Bordin, M., Vardanega, T.: Correctness by Construction for High-Integrity Real-Time Systems: a Metamodel-driven Approach. In: Reliable Software Technologies - Ada-Europe (2007)
14. Burns, A., Dobbing, B., Vardanega, T.: Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems. Technical Report YCS-2003-348, University of York (2003)
15. Panunzio, M., Vardanega, T.: A Metamodel-Driven Process Featuring Advanced Model-Based Timing Analysis. In: Abdennahder, N., Kordon, F. (eds.) Ada-Europe 2007. LNCS, vol. 4498, pp. 128–141. Springer, Heidelberg (2007)
16. Bordin, M., Vardanega, T.: Automated Model-Based Generation of Ravenscar-Compliant Source Code. In: Proc. of the 17th Euromicro Conference on Real-Time Systems (2005)
17. Bordin, M., Vardanega, T.: A Domain-specific Metamodel for Reusable, Object-Oriented, High-Integrity Components. In: OOPSLA DSM 2007. ACM, New York (2007)
18. Goodenough, J.B., Sha, L.: The priority ceiling protocol: a method for minimizing the blocking of high priority Ada tasks. In: IRTAW 1988: Proc. of the second international workshop on Real-time Ada issues, pp. 20–31 (1988)
19. ITU-T: (Rec. X.680-X.683, ISO/IEC: Abstract Syntax Notation One (ASN.1))
20. Dubuisson, O.: ASN.1 - Communication between heterogeneous systems (2000)
21. Semantix Information Technologies: The ASSERT project ASN.1 toolchain (2002), `http://www.semantix.gr/assert/`
22. asn1c: The Open Source ASN.1 compiler (2002-2007), `http://lionet.info/asn1c/`