Compilation of Heterogeneous Models: Motivations and Challenges

Matteo Bordin, Tonu Naks, Andres Toom, Marc Pantel

Compilation of Heterogeneous Models: Motivations and Challenges[1]

Matteo Bordin[1], Tonu Naks[2,3], Andres Toom[2,4,5], Marc Pantel[5]

bordin@adacore.com, {tonu, andres}@krates.ee, marc.pantel@enseeiht.fr

1: AdaCore, 46 rue d'Amsterdam, 75009 Paris, France
2: IB Krates OÜ, Mäealuse 4, 12618 Tallinn, Estonia
3: Laboratory for Proactive Technologies, Tallinn University of Technology, Ehitajate tee 5, 19086 Tallinn, Estonia
4: Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, 12618 Tallinn, Estonia
5: IRIT-ENSEEIHT, Université de Toulouse, 2 rue Charles Camichel, 31071 Toulouse Cedex, France

Abstract

The widespread use of model driven engineering in the development of software-intensive systems, including high-integrity embedded systems, gave rise to a "Tower of Babel" of modeling languages. System architects may use languages such as OMG SysML and MARTE**,** SAE AADL or EAST-ADL; control and command engineers tend to use graphical tools such as MathWorks Simulink/Stateflow or Esterel Technologies SCADE, or textual languages such as MathWorks Embedded Matlab; software engineers usually rely on OMG UML; and**,** of course, many in-house domain specific languages are equally used at any step of the development process. This heterogeneity of modeling formalisms raises several questions on the verification and code generation for systems described using heterogeneous models: *How can we ensure consistency across multiple modeling views? How can we generate code, which is optimized with respect to multiple modeling views? How can we ensure model-level verification is consistent with the run-time behavior of the generated executable application?*

In this position paper we describe the motivations and challenges of analysis and code generation from heterogeneous models when intra-view consistency, optimization and safety are major concerns. We will then introduce Project P[2] and Hi-MoCo[3] - respectively FUI and Eurostars -funded collaborative projects tackling the challenges above. This work continues and extends, in a wider context, the work carried out by the Gene-Auto[4] project [1], [2]. Hereby we will present the key elements of Project P and Hi-MoCo, in particular: (i) the philosophy for the identification of safe and minimal practical subsets of input modeling languages; (ii) the overall architecture of the toolsets, the supported analysis techniques and the target languages for code generation; and finally, (iii) the approach to cross-domain qualification for an open-source, community-driven toolset.

Keywords: model-driven development, code generation, DO-178, tool qualification, open-source.

Introduction

The core problem addressed by this paper has a social nature: the complexity of the interaction between the different departments which contribute to the development of high-integrity embedded systems – system, control, and software engineering. With the advent of model-based technologies, the industrial community hoped to have finally solved most communication issues by relying on a unifying, portable and semantically consistent representation of data: a model. However, such legitimate expectations turned to be a utopia: currently, each department is likely to

[1] This work is partly funded by the French and Estonian public funds through the FUI and Eurostars programs.
[2] www.open-do.org/projects/p
[3] www.eurekanetwork.org/project/-/id/6037
[4] www.geneauto.org

use a specific modeling language, which relies on its own semantics and assumptions on the run-time platform. Even worst, it is not unusual for inconsistencies to emerge even within representations of a single modeling language, for example, between the language documentation (specification), the model simulator and the code generator. The integration of heterogeneous models has thus become a major bottleneck in the whole development process. Over the time, the following questions have become more and more frequent:

- Are the data and interface representations consistent between different models (e.g. Simulink[5] and UML[6])?
- Is the model allocated on a software architecture that will ensure its correct rate of execution?
- Is the run-time behavior of the final executable application consistent with the model-level semantics?
- Is the generated code optimized for the software and hardware architecture?

While we are completely conscious that answering all of the questions above within a three-year project would be overly optimistic to say the least, still we believe it is feasible to trace a clear route towards a sound model-based integration approach. This is why the introduced projects were undertaken.

The ultimate goal of these projects is to provide a verification and code generation oriented framework for heterogeneous models. The resulting toolset is intended to be:

- *Open*, by supporting existing modeling languages, which are either *de facto* (e.g. Simulink) or *de jure* (e.g. UML) standards. In our opinion, there is no need of creating a new language with new semantics, but rather to pragmatically and carefully select a safe subset of the existing languages and to provide a full formal definition including the execution semantics**.**
- *Flexible*, by splitting the verification and code generation process into a sequence of *elementary tools* which can be easily customized by the final users to introduce domain-specific tuning. This is the same ideal architecture, which made GCC, the GNU Compiler Collection, one of the most successful engineering projects ever. It will also allow simpler and better verification activities using the most appropriate technology for each atomic tool including formal methods.
- *Qualifiable*, by defining and deploying an agile and repeatable qualification process which would enable the end user to easily re-qualify the toolset in its own context and including user or domain-specific behavior such as: changes to the code generation strategy, integration with existing component frameworks or operating systems and addition of new input modeling languages or target programming languages.

The overall technical approach of both projects is to define a pivot formalism (for simplicity called the P formalism), which is used as an intermediate representation for heterogeneous models. Intra-view consistency verification and code generation relies on this intermediate representation, thus allowing distributed code generation from heterogeneous models. The code generators from the P formalism are intended to generate 100% of code deriving from imported models without asking to manually modify or process the generated code.

This paper now proceeds as follows. First we clarify what we mean by "qualification" and why it is so relevant for model compilers. Then we discuss the constraints imposed to the input modeling languages. And in the end, we introduce the P toolset, and in particular its overall architecture and envisioned qualification strategy.

[5] www.mathworks.com/products/simulink
[6] www.uml.org

The meaning of qualification

To avoid any possible misinterpretation of the term *qualification*, in this paper we consider DO-178, the international standard for certification aspects of airborne software, our main reference.

DO-178 requires achieving a set of objectives in order to certify airborne software. Examples of objectives range from verifying that requirements are allocated to the design, to measuring the structural coverage and checking the coding standard. Some of these objectives can be reasonably achieved by using a tool automating the required activity: for example, to achieve the compliance to the coding standard objective, we can use a coding standard checking tool. For the tool output to be part of the certification evidence *without any additional manual* review, the tool quality shall be assessed by *qualifying* the tool itself. In DO-178 jargon, we would say that the applicant *gains certification credit from the tool qualification*. The reader is invited to notice the dichotomy of certification and qualification in DO-178: certification refers to airborne software, while qualification refers to the tool used to build this airborne software.

The complexity of the qualification process depends on the tool typology and on the amount of certification credit the user would like to gain from the tool qualification. In the example above, the tool could at most *fail to detect an error*, and is thus called a *verification tool*. In this case, the process required by DO-178 to qualify the tool is pretty lightweight: a requirement-driven, "black box" verification. A model compiler is categorized as a *development tool* – it actually creates the source code and could potentially *introduce an error* in the airborne software. Without going into the details of DO-178, we can state that the process for qualifying a development tool is as complex as the process for certifying airborne software. For example, if a model compiler is used in a DAL level A project (the highest software safety Design Assurance Level of DO-178), then the tool must be developed following a process corresponding to the same safety level as the flight control system (so called Tool Qualification Level 1 in the forthcoming version C of DO-178), including achieving Modified Condition/Decision Coverage (MC/DC) structural coverage of the tool source code. The additional expenses on the tool qualification are, however, expectedly won back in certification credit: while the qualification of a coding standard checking tool would permit to get certification credit for a single verification activity, the qualification of a model compiler would permit to gain certification credit for several development and verification activities, including development of source code, compliance of source code with low-level requirements and compliance of source code with standards. One key point is that, even if qualification is related to each specific system developed with the tool, most of the qualification elements can be shared between the various systems, whereas the system verification activities that are alleviated by using a qualified tool must be conducted for each system.

The brief introduction above is of course just a minimal insight of DO-178, but it explains why the qualification of a model compiler is particularly appealing to the applicants.

Overview of the input modeling languages

In this section we will describe the main aspects of the subsets of modeling languages selected by the industrial partners of Project P. As the projects are currently at an early stage, the detailed selection of language features is still ongoing. The subset selection process is guided by two major needs: (i) to guarantee that the intended system functionality can be correctly implemented in an executable run-time system which is verifiable and (ii) to assure that the qualification of the code generator as a development tool at level A (DO-178B) or at TQL1 (DO-178C) is feasible.

The languages of interest are divided by modeling viewpoints, assuming the Globally Asynchronous Locally Synchronous (GALS) computational model:

- System/Software architecture: SysML/MARTE and AADL
- Software architecture: UML
- Behaviour: Simulink, Stateflow, Scicos/Xcos, Embedded Matlab, Scilab and UML

The two first viewpoints are both concerned with the system and software architecture and the borderline between the two is not always very clear in real life. The system/software architecture view is concerned with the split-down of a larger system to functional subsystems, their topology and interfaces. This view allows also to map software components to physical artifacts like processors, buses and operating systems elements like tasks and threads. The SysML[7] and UML MARTE[8] profile that are standardized by OMG and the Architecture Analysis & Design Language (AADL)[9] standardized by SAE, are all addressing these aspects in slightly different ways and are more and more finding usage in the industry. The pure software architectural view is more concerned with the decomposition of the software components themselves and UML is most widely used for that purpose. The selection of the necessary and sufficient subsets of the aforementioned languages for usage in the embedded high integrity domain represented by the Project P industrial partners is currently ongoing.

The behavioral modeling is concerned with providing actual functional algorithms for the components of the developed systems. Depending on the kind of the system and modeling practices there are many choices. For a more mathematically oriented control algorithm a synchronous dataflow representation can be appropriate. Simulink, Scicos[10], Xcos[11] and SCADE are some of the tools that provide such formalisms. For more imperative algorithms a state chart or an activity diagram can be more suitable. One can use, for instance, UML activity diagrams, UML state machines or Stateflow[12] charts. At other times it might be more convenient to write an algorithm in a textual language like Matlab or some tabular formalism. In the projects P and Hi-MoCo the Simulink, Stateflow, Scicos/Xcos, Embedded Matlab, Scilab and UML (activity and state diagrams) languages have been chosen as the supported input languages for behavioral modeling. For Simulink, Stateflow and Scicos/Xcos we are relying on the language subsets and associated modeling rules identified by Gene-Auto [1]. These subsets have been shown to be sufficient for modeling real industrial control applications [2]. Some extensions, like supporting some additional Simulink blocks and a subset of the textual Matlab language have been planned in the current projects.

The supported subsets of the input languages are accompanied with a set of modeling rules, which are primarily meant to guarantee that the models are appropriate for embedded software and do not contain error-prone or overly complex constructs and also allow a relatively straightforward translation from models to source code. Of course, one major challenge is not to impose modeling rules which are too constraining. Some examples of reasonable modeling rules are:

- The model must contain enough type information to allow unambiguous typing of the complete model
- Cycles in dataflow models must involve a unit delay (the causality rule)
- State charts are not allowed to generate local events – due to the run-to-completion semantics such events trigger a recursive evaluation of the chart (and possible generation of new events), which is not a desirable behavior in an embedded application
- An OR decomposition of a state chart must always have an unguarded default transition to avoid an inconsistent state at run-time.

[7] www.sysml.org
[8] www.omgmarte.org
[9] http://standards.sae.org/as5506a
[10] www.scicos.org
[11] www.scilab.org/products/xcos
[12] www.mathworks.com/products/stateflow

Overview of the toolset architecture

The P toolset greatly benefits from the experience obtained during the GeneAuto project [1]. It is fair to say that the contents of this section are an evolution of those contained in [1] and describing the GeneAuto architecture. The main novelty introduced on this paper is the framing of the tool architecture into the DO-178C context, in particular by taking into account the Object-oriented design supplement and the Tool Qualification Document.

One of the main challenges when developing qualifiable development tools lays in defining an architecture that is flexible enough to allow a certain degree of configurability for the user while still not increasing the complexity of qualification to prohibitive heights. Even in the case of a model compiler for a single language, aspects such as the deployment of reusable components, the exploitation of external libraries and the mechanism chosen for assuring configurability have all a significant impact. Imagine for example to qualify a simple code generator for UML class diagrams. How would you gain confidence on the use of external XML libraries to parse input models? How would you guarantee the user can customize the code generation strategy without requiring a new qualification process to be deployed from scratch? In the case of a heterogeneous model compiler, the amount of such questions simply explodes due to the exponential number of combination of possible input modeling languages, code generation strategies and target platforms. Given the complexity of qualifying model compilers, the impact of a loose tool architecture would be simply devastating.

To minimize the risks associated to the possible combinations of input languages and code generation strategies, we are adopting the following architecture. The tool, which we will call from now on the P toolset, is delivered to applicants as a *tool collection* comprising several distinct executable programs: several Importer Tools and a single Model Compiler Tool. Note that the notion of tool collection is introduced by the Tool Qualification Document of DO-178C itself. The executables are bound together by the qualification data accompanying the toolset and its components are not intended to be qualified separately, but rather as a complete chain. The Importer Tools transform user languages into their representation into the P formalism, while the Model Compiler tool transforms models in the P formalism into source code. Such a distinction has two main advantages:

a) It allows developing several different and simpler importers, so as to take into account different serialization formats and technologies. For example, UML modeling tools are usually able to serialize models in XML files following a precise standardized schema. At the same time, Simulink uses a proprietary textual format which may change in each release of the tool. By decoupling the Importer Tool from the Model Compiler tool, we allow a separate evolution of those two components, so as to easily support new languages or follow the evolution of serialization formats without requiring to fully re-develop the qualification data for the entire model compilation chain.

b) It enables to deploy different qualification strategy for the Importer and Model Compiler tools. At least two possible choices exist for the Importer tool: (1) we develop the tool following a strict development process corresponding to the safety level of the future target of code generation or (2) we develop a separate verification tool to check *a posteriori* the equivalence between the input models and their representation in the P formalism. The option (1) is not practical for all types of importers foreseen for project-P, because their implementation may require languages not appropriate for TQL1 qualification or have significant run-time dependencies. This is for example the case of a Simulink importer written in Matlab: the major problems would be the lack of appropriate tools for the Matlab scripting language (for example, to measure structural coverage) and its interpreted nature, which would require at least some form of verification on the Matlab interpreter. Considering that the first representation in the P formalism and the user models are pretty close in terms of abstraction level, we strongly believe that development of a verification tool that enables to qualify importer as a “black box” is also technically feasible. The option (2) is also a much more lightweight solution because it does not require qualification of the whole development process of the tool, including the development of tests to achieve MC/DC.

The reason why the option (2) is not applied also for the Model Compiler tool is twofold. First of all, as clarified by the Tool Qualification Document, it won't permit to gain full certification credit from the tool qualification: we will return on this point on the section about the qualification approach. On a second instance, and on a more technical level, in the case of the Model Compiler tool the semantic distance between the input (model in the P formalism) and the output (source code) is so wide that establishing evidence of equivalence would be rather difficult (not to say impossible). One additional limitations of this approach is the difficulty associated to the introduction of any form of optimization in the model compilation process. An optimizing model compiler would require a more and more sophisticated construction of traceability data to provide evidence of semantic equivalence. However, as demonstrated by the Gene-Auto project, model compiler optimizations are quite important to ensure high-performance (in space and time) executable object code, unless rather advanced (conventional) compiler optimizations are activated that may require additional verification activities on the generated binary code.

By splitting the P toolset into a collection of several tool executables, we ease the support for new input languages and allow for separate production of qualification data for the Importer and Model Compiler tools.

However, this high-level architecture alone does not make the toolset flexible enough to allow the customization of the Model Compiler from the P formalism to support user-specific needs with a reasonable impact to tool qualification. This task requires providing mechanisms to override specific aspects of the model compilation chain or even to add new transformation steps into the chain. To achieve this result, we extend the notion of tool collection. The Model Compiler tool is an *ordered* composition of *elementary tools* implementing an externally observable transformation in the model compilation chain. Examples elementary tools are: scheduling and type inference of data-flow models, parallelization and creation of synchronization/protection mechanisms in architectural models and printing of the refined P formalism to source code. Each elementary tool is contained in a reusable object-code library, making it a *de facto* re-usable software component, of course shipping with qualification data. Elementary tools are integrated into the *backbone model compiler architecture*, which is a generic architecture for model compilers offering just model reading/serialization and registration of model transformation steps. Users can thus decide to include new transformation steps simply by registering them to the backbone architecture. Overriding the model compilation steps can be achieved by the classical inheritance/overriding mechanism of object-oriented design. We map each elementary tool to a class implementing a precise interface: user-specific needs can be implemented by re-implementing such an interface or by overriding some services of the default implementation

One last core aspect for the toolset architecture is related to the use of external libraries, for example to read/write XML data. The Tool Qualification Document acknowledges the notion of external component as software components, which are not *"not under control of the developer of the tool (such as operating system functions or an external software library)"*. When using external components, the applicant is not required to provide full qualification data or to achieve MC/DC coverage on them. It is however required to identify them in the design phase and to ensure that requirement-driven tests actually exercise the interfaces and functionalities of each external component. This new clarification of the Tool Qualification Document allows for much more flexibility in development tool qualification.

A manifesto for the P formalism

As shown above, the P formalism is the cornerstone of the entire architecture of the P toolset. The main principles behind its design are best summarized by the following manifesto:

- The P formalism is a pivot formalism encompassing functional behavior and architectural aspect for describing high-integrity real-time embedded systems.

- The P formalism is a not visible to the final user and doesn't have any concrete syntax. The concrete syntax is provided by the standardized languages accepted by the importers of the P formalism. It is also void of any syntactic sugar to ease model editing: it is thus minimal.

- The P formalism is partitioned because it is composed of a collection of interconnected domain-specific formalisms, each of which corresponding to a precise modeling viewpoint. One key design decision is to separate the generated code in two aspects: sequential and concurrent/distributed. In that purpose, we split the P formalism in functional behavior whose generated code will be sequential, and architectural behavior whose generated code will be concurrent and distributed.

    1. The P formalism *for synchronous functional behavior* to represent synchronous behavior which will be executed in a sequential manner;
    2. The P formalism *for asynchronous functional behavior* to represent asynchronous behavior which will be executed in a sequential manner;
    3. The P formalism *for real-time architecture* to represent both synchronous and asynchronous, static and dynamic, concurrent, distributed architectures;
    4. The P formalism *for datatypes* to describe data dictionaries and their raw representation in memory;
    5. The P formalism *for integration* to create links across the domain-specific formalisms above.

- The P formalism has three main layers:

    1. The design layer, which is close to the concepts of the input modeling languages. Typical elements of the specification layer are those found in input modeling languages, such as state, action, task, block, port or flow. One key point is that the models are deterministic on the functional side: given the same input values, they will produce the same output values. But, there can exist several execution paths that will lead to the same result and all of them are contained in the model. Thus, model execution is not deterministic even if its results are always the same.
    2. The software mapping layer, which refines elements of the specification layers into an explicit implementation that is void of all non-determinism and compliant with the requirements of the analysis tools. For example, a dataflow model with parallel branches coming from the design layer may be refined into a sequenced model on the software mapping layer; or as another example, synchronization/protection mechanisms might be generated for an application in the real-time architecture if required.
    3. The code model layer, which refines elements of the software mapping layer into elements close to the elements of conventional imperative programming languages extended with domain specific data structures like array and vectors and the associated operators. Typical elements of the code model layer are: statement, variable, function or module. This last layer is the one used to print the source code.

- The P formalism has well-defined static and dynamic semantics. To ensure this rather obvious requirement is correctly implemented, we plan to rely on the Gene-Auto System Model for the behavioral viewpoint and on AADL for the real-time architecture viewpoint.

Supported verification and analysis techniques

Early validation and verification plays a key role in the adoption of model driven engineering for the development of safety critical systems. Many kinds of model verification and analysis activities can be conducted depending on the kind of data expressed by the model. These activities can focus on the structure or syntax of the model or on its behavior or semantics. The P formalism is a collection of interconnected modeling languages that allows expressing different views. It is thus mandatory to check both properties internal to views, and properties of the whole model combining all the views. As example, properties must be checked both on the internal properties of functional and architectural models, and on the external allocation from functional to architecture models. Another kind of properties is fundamental inside the P project: the correctness of the various model transformations involved in code generation. These properties can be also both structural and behavioral.

Regarding structural properties, the OMG proposal for Model Driven Architecture (MDA), modeling languages are defined using the MOF meta-modeling formalism and structural constraints can be expressed using the OCL specification language. OCL is based on First Order Logic extended with model manipulation operators. It allows expressing a wide range of properties except the ones related to the evolution of models during time. Project P will rely either on OCL or on similar languages in order to express model consistency properties both intra-view and extra-view. These properties will either be generic or user domain, industrial context, or project specific. Following early experiments in the OPEES and quarteFt projects, project P will rely on OCL to express and verify the correctness of model transformations relying on explicit trace links between the source and target languages.

Behavior verification or analysis requires more complicated tools that cannot be developed for each new language. The P toolset will rely on existing external tools in that purpose. For example, the TOPCASED, SPICES [3], and quarteFt projects have designed the FIACRE verification dedicated modeling language, as an intermediate between several model checking toolsets such as TINA and CADP. FIACRE will be used as support for the verification of temporal behavioral properties in P.

The definition of these different kind of structural and behavioral verification and analysis toolset based on the common P formalism will then enable end users to build their own domain, enterprise or project specific system development toolchain like it was advocated for example in TOPCASED and MEMVATEX [4].

Envisioned qualification process in an open-source context

Qualifying software according to strict industry standards, like DO-178 in the avionics, is massive work [5]. Furthermore, it enforces strong constraints on the development process and requires keeping the end product's functionality and variability to a strict minimal. This poses a challenge for production of flexible tools that are customizable for end users with different needs and/or are developed by a community. The overall toolset architecture introduced in a previous section explains how the tool can be tuned to meet the users' needs by providing new elementary tools and registering them into the backbone. In this section, we explain how we plan to put to practice this mechanism and frame it in the context of the DO-178C Tool Qualification Document and Object-Oriented Design annex.

One major point of qualifying a component-off-the-shelf (COTS) tool lies in clearly identifying the responsibilities for producing qualification data. Again, the Tool Qualification Document comes to help, in particular by identifying three different levels of requirements:

- Developer Tool Operational Requirements (developer-TOR) which describes the expected functional behavior of the tool with respect to the input and output formalisms.
- User Tool Operational Requirements (user-TOR), which are used to map the user needs to the developer-TOR.
- Tool Requirements, which are a refinement of the developer-TOR.

In the general case, the developer and user –TOR would be quite close, with the user-TOR simply providing a refined vision of the developer-TOR for the qualification of the tool in a precise environment. In the context of the P toolset, the situation is slightly different because it is a tool collection rather than a single tool. Each executable tool of the collection is qualified against its own developer-TOR. Notably, such tools are the Importer Tools that map the particular input languages to the P formalism, and the Model Compiler Tool, that maps the P formalism to source code. Tool users do not have direct visibility on this internal split, but are instead required to express their tool operational requirements with respect to the mapping between the input languages and source code *without explicitly mentioning the intermediate transformation to the P formalism.* This is in line with the positioning of the P formalism as a hidden intermediate language without any concrete syntax. To simplify the production of the user-TOR, we plan to provide a standard user-TOR, which directly abstracts the coupling of the developer-TOR for the different executable tools included in the collection.

The tuning of the tool is still possible for advanced users. Instead of qualifying a complex tool supporting several configuration options, we opted for a simpler architecture but let the user add and/or override model transformation steps by registering new elementary tools into the backbone. Given then commercial open source license of the tool *and* of the associated qualification material, this approach is feasible. However, the modification of some particularly critical elementary tools would invalidate a significant amount of qualification data and the developer-TOR. For example, modifying the elementary tools that calculates the sequencing of blocks of a synchronous data flow model would be quite risky because a significant part of the remaining model transformation assumes that sequencing was correctly done. For this reason, only a limited subset of model compilation steps can be overridden and, when registering new elementary tools to the backbone, it must be ensured that they preserve some basic properties of the previous transformations such as, for example, the partial ordering of blocks.

The use of object-oriented design (OOD) for the mechanism used to extend and/or override the model compilation strategy of the P toolset deserves a separate discussion. While OOD has been used in the mainstream software engineering for decades now, its application to safety-critical software is pretty limited, mostly because of the complexity of satisfying a consistent verification strategy when using dynamic dispatching (and potentially, depending on the language technology, garbage collection and/or virtualization). The application of OOD in the context of Project P will be the central topic of a future paper, in particular on how we managed to marry the flexibility requirement with the DO-178C Object-Oriented Design supplement.

From a high-level perspective, we thus propose a double strategy for qualification. First of all, we package the P toolset as a tool collection, which brings the advantages previously described, while still allowing the user to map their own user-TOR onto the developer-TOR. Moreover, we allow selective overriding of model transformation steps and registration of new elementary tools into the backbone and, thanks to the open-source nature of the tools and qualification data, allow also a user re-production of the qualification data with minimal effort.

Conclusion

In this position paper we have illustrated the main motivations and challenges for developing an heterogeneous model compiler. Despite the fact that the two major technical characterizing factors, openness and flexibility, are quite uncommon in high-integrity software, the main challenge rather lies in the qualifiability of the toolset. For this reason, and due to the early state of the project, the main focus of this paper is on the overall tool architecture and the qualification approach.

What we have presented in this paper is nevertheless a manifesto for developing heterogeneous model compilers and designing intermediate modeling languages intended to be visible to tools rather than the final users. While deeply leveraging on the results and experience of the Gene-Auto project, Project P promises to ease the deployment of a cross-department model-driven engineering process which complies with the upcoming DO-178C standard.

References

[1] Toom, A.; Izerrouken, N.; Naks, T.; Pantel, M.; Ssi Yan Kai, O. (2010). Towards reliable code generation with an open tool: evolutions of the Gene-Auto toolset. In: 5th International Congress and exhibition ERTS2 2010 : Embedded Real Time Software and Systems, 19-21 May 2010, Toulouse, France: SIA/3AF/SEE 2010. http://www.erts2010.org/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010%202/ERTS2010_0129_final.pdf

[2] Rugina, A.-E., Dalbin, J.-C.: Experiences with the GENE-AUTO Code Generator in the Aerospace Industry. In: In: 5th International Congress and exhibition ERTS2 2010 : Embedded Real Time Software and Systems, 19-21 May 2010, Toulouse, France: SIA/3AF/SEE 2010. http://www.erts2010.org/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010/ERTS2010_0026_final.pdf

[3] Berthomieu, B., Bodeveix, J.-P., Dal Zilio, S., Dissaux, P., Filali, M., Gaufillet, P., Heim, S., Vernadat, F. Formal Verification of AADL models with Fiacre and Tina. In: 5th International Congress and exhibition ERTS2 2010 : Embedded Real-Time Software and Systems, 19-21 May 2010, Toulouse, France: SIA/3AF/SEE 2010. http://www.erts2010.org/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010%202/ERTS2010_0107_final.pdf

[4] Albinet, A. et al. The MeMVaTEx methodology: from requirements to models in automotive application design. In: 5th International Congress and exhibition ERTS2 2010 : Embedded Real-Time Software and Systems , 19-21 May 2010, Toulouse, France: SIA/3AF/SEE 2010. http://www.syndex.org/publications/pubs/erts08/erts08.pdf

[5] Bordin, M.; Gasperoni F.: (2010). Towards Verifying Model Compilers. In: 5th International Congress and exhibition ERTS2 2010: Embedded Real Time Software and Systems, 19-21 May 2010, Toulouse, France: SIA, 2010. http://www.erts2010.org/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010%202/ERTS2010_0134_final.pdf