Towards Verifying Model Compilers

Matteo Bordin[1], Franco Gasperoni[2]

1: AdaCore, 46 rue d'Amsterdam 75009 Paris (FRANCE), bordin@adacore.com
2: AdaCore, 46 rue d'Amsterdam 75009 Paris (FRANCE), gasperoni@adacore.com

Abstract: In this paper we propose a methodological and technical approach to develop model compilers which provide formal proof of semantic preservation and complement model-level verification activities with formal analysis on the generated source code. Our paper also details which characteristic a programming language shall exhibit to be used as target of a model compiler. Finally, we evaluate the impact of using formal programming languages on the development and qualification of model compilers.

Keywords: Code generation, formal methods, model-driven development.

1. Introduction

Model-driven development is a major innovation vector for the construction of high-integrity embedded systems. The paradigm has been successfully applied to several large projects in the last decade, showing the advantages of shifting from source-centric to model-centric processes. Regardless of success stories, model-based approaches have failed to completely replace source code in the application life cycle. Why?

The first observation is that some model transformation chains have failed to provide clear added-value for the modeling process. These model translators treat the model just as a blueprint of the source code providing no meaningful abstraction gain. For example, several UML2 tools implement a one-to-one mapping between model and source code elements: modeling is thus just "graphical coding". In this case, there is no advantage in choosing a programming language exhibiting particular properties: the generated code is used simply as an intermediate representation between model and object code and no additional verification activities are performed on it.

The second observation concerns the preservation of model-level properties at source-level. This has been done by providing empirical evidence that the code generator has been developed with care and has undergone adequate testing. By contrast, formally proving that properties are preserved from model to source increases confidence in the model translator and in verification activities performed directly on the model.

The third observation is that going from model to object-code is done in two explicit steps: first the source is generated and then it is compiled to object code. In other terms, source code is an explicit artifact of the development process. This requires explicit configuration management of source code, even if it could be desirable to treat it like any other intermediate representation generated during the compilation process.

In this paper we propose a methodological and technical approach to circumvent some of the limitations described above. In particular, we focus on end-to-end property preservation and cohesion of the whole model compilation chain. While we fully acknowledge the importance of promoting model-based development as a mean for abstraction, we intend to address this specific topic on a separate work.

The approach described in this paper focuses on finding a formal and executable intermediate representation for a model. Such intermediate representation is used (1) to demonstrate that model properties are preserved; (2) to complement verification activities for the target platform; and (3) to compile the model to object code. Within this paper, we intend to explain why this approach would be of benefit in the development of model compilers in the high-integrity domain, with particular attention for application certification and tool qualification in the context of the DO-178 standard [18].

2. Overview of Modelling Languages

In this section we provide a brief overview of some important modeling languages for high-integrity embedded systems and evaluate whether their compilation chains suffer from the limitations described in the introduction.

2.1 UML2, SysML and MARTE

UML2 [1] is a general purpose graphical modeling language for object-oriented systems. The language is particularly suited to describe structural properties of systems – like components structure, services and interactions, state machines and deployment. In particular UML2 provides a graphical representation of a program's architecture (i.e. the packages or namespaces, the types or classes, and the program operations).

This graphical representation is a blueprint where relationships among types and classes have been made clear. This representation is more human-readable than its textual counterpart ("a picture is worth a thousand words"). When used in this fashion UML2 is often complemented with textual representation for operation implementation (here a short text is worth a thousand pictures). The one pragmatic drawback when switching between the graphical and textual paradigm is at the environment level: UML2 tools are usually better at handling diagrams than editing text.

To increase the expressiveness of UML2 diagrams, it is possible to use an abstract action language: a high-level and platform-independent textual language to describe algorithmic behavior. The OMG is currently standardizing such a language. UML2 also relies on OCL (Object Constraint Language) to express constraints such as methods pre/post conditions, type invariants, and assertions.

UML2 has a well defined platform-independent semantics. The UML2 standard defines the intended static and dynamic semantics and deliberately gives no indication on how such semantics should be implemented. Presently, the limitation of UML2 is its uncontrolled and complex multiple-view modeling space. A system may be modeled in UML2 from different overlapping viewpoints or, in more common terms, diagrams. Unfortunately, the standard does not provide clear indications on how to guarantee consistency among views. To guarantee view consistency, one limits the views used to non-overlapping ones. See [2] for a discussion on the UML2 diagrams needed to model a complete system while avoiding view overlap.

Code generation strategies for UML2 usually treat the model as a blueprint of the code: model elements are mapped to source via a one-to-one mapping. From this perspective, the target programming language is not particularly important since no methodological added value comes from targeting a specific language. Property preservation at source-level is difficult to demonstrate as some of these languages are hard to analyze formally and because most programming languages have no direct way to map OCL constraints, making it difficult to prove that dynamic and static model-level constraints are respected in the translated sources.

SysML [3] is a UML2-derived language used to describe industrial systems with particular emphasis on system-level components interaction and hybrid systems modeling. MARTE [4] describes the concurrent execution in both platform-independent and platform-specific terms. The platform-specific semantic elements of MARTE can be used for schedulability and other types of concurrent system analysis. This subset of MARTE integrates AADL [5] via the UML profiling mechanism.

2.2 Simulink and Stateflow

Simulink and Stateflow [6] are used to model control systems via block diagrams (for data flow) and hierarchical state machines (for state-based behaviour) with an emphasis on mathematical equations. Differently from UML2-based tools, Simulink and Stateflow raise abstraction to the level of control theory and thier model translators employ a more complex code generation strategy translating a set of equations into a list of sequential operations with causal and temporal dependencies. Simulink and Stateflow model translators, however, do not provide evidence of property preservation and elide differences in mathematical semantics between model-space and target hardware.

3. Introducing Verifying Model Compilers

To improve the state-of-the-art a *verifying model compiler* would: (1) provide formal evidence that model properties are preserved in the model-to-source translation; (2) perform target-dependent verification activities which cannot be easily done at model level; and (3) wrap the whole translation chain in a single logical tool.

3.1 Property Preservation

Modeling languages are particularly well suited for the formalization and verification of structural properties (safety properties of state machines, determinism of concurrent models, ...). A limitation of today's model translation technologies is the lack of formal evidence when it comes to property preservation in the refinement process from model to sources. The core reason for such limitation lies with target programming languages which are executable but not formally analyzable. In practice, the target programming language cannot formalize the properties expressed at model-level: such properties are "lost in translation".

The approach chosen for verifying model compilers is to target formal programming languages which are both executable and analyzable. Such programming languages should be able to represent the properties formalized at model level and permit their proof at source-level. Some of the properties we are interested in are:

- Information and data flow contracts: how components share data and information.
- State machine properties such as safety, completeness, and determinism.
- Design-by-contract in the form of pre/post-conditions, invariants, and assertions.
- Determinism and analyzability of concurrent execution.

In section 4 we will analyze several target languages (SPARK/Ada, ACSL/C, JML/Java, Spec#/C#) and evaluate them in terms of property preservation, soundness of proving technology, efficiency, and interfacing abilities with other languages. This last point is significant as some components of the final application may be handcrafted possibly because of reuse or outsourcing.

3.2 Complementing Verification Activities

Verifying model compilers complement model-level verification with formal proofs at source-level. In fact, several crucial characteristics of a complete system can be proved only at source-level. For example, consider platform-specific properties such as absence of overflow, array-out-of-bound, or stack overflow errors: proving such properties requires knowledge of the implementation semantics and may not be provable at model-level – even in the presence of fully expressive abstract action languages. This need for source-level analysis to verify target-specific properties requires targeting a programming language for which sound proof techniques exist.

Errors detected at source-level should be either fed back into the model (e.g. an overflow may imply that a computation in the model diverges) or are due to errors in the model compiler implementation. Absence of run-time errors in the generated sources reinforces confidence in the model compiler. In both cases the modeler need not understand the intermediate language representation used for formal verification. In section 4 we evaluate possible target programming languages from this perspective.

3.3 A Single Logical Tool

From a technical perspective, current model translation techniques separate source code generation from traditional source-to-object compilation. This strategy forces the modeler to regard source code as an explicit product of the development process and thus subject to configuration management, verification and traceability. To remove these source-centric activities, model-to-object translation should be wrapped in a single logical tool. A verifying model compiler would hide intermediate source code generation and proofs from the modeler in the same way a conventional compiler hides semantic analysis and intermediate code generation from programmers. Furthermore, by wrapping the whole model compilation chain in a single logical tool, it is easier to cross-compile a model, execute it, and debug it on the target hardware (or simulator). This approach is orthogonal to model execution/debug via interpretation on an abstract (UML) virtual machine.

As a final note, a verifying model compiler should generate appropriate hooks where foreign code may be interfaced to incorporate existing or sub-contracted components.

4. Evaluation of Possible Target Languages

As stated in the previous section, the intermediate representation generated by a model compiler shall:

1. Be unambiguous.
2. Guarantee that the model semantics is preserved in the transformation to ensure that verification activities performed at the model level apply also to the final executable program.
3. Permit additional target-dependent static verification activities to be performed, such as: checking for absence of run-time errors, verifying conformance to design contracts (pre/post conditions, invariants, etc.), analyzing stack size requirements, and computing worst-case execution time.

Point 1 says that the static and run-time semantics of the target language shall be well defined. Point 2 says that the chosen target programming language must be rich enough to represent most (if not all) semantic constructs of the input model, possibly including formally expressed properties. Point 3 says that verification tools for the chosen programming language must exist to complement the verification activities performed at the model level.

Contrary to the common practice, the developer of the model compiler, and not the end users, should choose the target programming language of a model compiler. This choice has a great impact on the development of a sound model compiler and on the possibility of qualifying it with respect to appropriate industrial standards (ex. DO-178). We return to this specific point in section 5.

Considering the input modeling language of interest (the UML-centric family and Simulink/Statflow) and due to the requirements above, our candidate target programming languages are:

1. SPARK: SPARK [8] is a subset of Ada augmented with annotations to describe contracts for information/data flow and behavior (pre/post conditions, invariants). Contracts can be formally proved through automated tools.
2. C and the ANSI C Specification Language (ACSL): ACSL is an annotation conceptually similar to SPARK annotations. Contracts can be formally proved via automated tools, for example in Frama-C [9].
3. Java and the Java Modeling Language (JML). JML [10] is an annotation language to specify the behavior of Java programs. It is used to improve both static and dynamic checks for Java programs. Contracts can be formally proved using the ESC/Java2 theorem prover [10].
4. Spec# [11], an extension to the C# programming language to describe and formally prove behavioral contracts of C# programs.

The choice of the target programming language is strongly connected to the availability of a conventional compiler presenting the appropriate characteristics. In particular, the compiler shall:

- Target the chosen hardware target.
- Produce efficient object code.
- Require minimal run-time libraries. The time or space overhead induced by the run-time support may cause a discrepancy between the results of analysis performed at the modeling or programming language level and empirical evidence. For example, consider a compiler targeting a virtual machine that includes a thread to perform garbage collection: this thread is present neither in the model nor in the intermediate source-level representation, complicating the timing analysis.
- Guarantee some form of traceability between source code and object code: this requirement derives from international standards for high-integrity embedded systems development such as DO-178B for commercial avionics.

We cannot thus evaluate programming languages and their compilers separately, but we shall instead treat them a whole.

The fitness of a programming language as target of a model compiler is evaluated based on three criteria: (i) the programming language coverage of the semantic needs for modeling language described above, (ii) the kinds of static verification supported at source code level and (iii) the availability of compilers for the domain of interest.

SPARK supports the description and proof of data/information flow and behavioral contracts. Concurrency is supported with first-class language features and annotations to prove absence of deadlocks, race conditions and priority ceiling violations. It has a focus on proving absence of run-time errors such as numeric overflow or out-of-bound array indexes. It has a limited support for object-oriented semantics: polymorphism is outside the language semantics. The SPARK theorem prover is

sound. SPARK is compiled to object code using standard Ada compilers: conventional compilers with the properties described above are available and all Ada compilers generate equivalent object code starting from the same SPARK program. Since SPARK is based on Ada, its interfacing with other languages is pretty straightforward.

C and ACSL present characteristics similar to SPARK, with the main difference being the lack of support for concurrency. The typing system for C is much less expressive than the SPARK one: the C and ACSL representation of a given semantics can be much more verbose than the corresponding SPARK one. For example, SPARK can directly express type properties such as the minimal and maximal value for an integer; while ACSL requires annotations (logic formulae) for each single variable or formal parameter. This difference is caused by the different roots of the two languages: SPARK derives from Ada, which has an expressive typing system; ACSL is based on C which is extremely poor (and platform-dependent) when describing types. Sound theorem provers for ACSL exist, such as Frama-C [9]. C and ACSL are compiled to object code using C compilers: conventional compilers with the properties described above exist. Interfacing C with other languages is straightforward.

Java and JML support formal specification and proof of properties on object-oriented programs using polymorphism. ESC/Java2 is, by its own design, an unsound theorem prover. The language does not provide advanced support for numeric types and their analysis. The language can rely on the concurrency features described by the Real-Time Java and Safety Critical Java initiatives, which are conceptually similar to those present in SPARK; however, no specific JML annotations exist for such concurrency features. A Java implementation may include a virtual machine to execute, increasing the semantic distance between the input for analysis and the actual executable.

Spec# supports formal specification and proof of properties for object-oriented programs using polymorphism. The language does not provide advanced support to describe and analyze numeric types, and it lacks concurrency features appropriate for the domain of interest. At the moment of writing, Spec# requires a compiler targeting a .NET implementation, making it unsuitable for high-integrity embedded systems.

Language	Evaluation
SPARK	*Expressivity:* ++
	Static verification: ++
	Compilation: +++
C/ACSL (Frama-C)	*Expressivity:* +
	Static verification: ++
	Compilation: +++
Java/JML	*Expressivity:* ++
	Static verification: +
	Compilation: ++
Spec#	*Expressivity:* ++
	Static verification: ++
	Compilation: +

The summarized results of our evaluation are:

- SPARK supports design-by-contract, numerics and concurrency but has a limited support for object orientation. It uses a sound theorem prover. It is compiled to efficient object code.
- C and ACSL support design-by-contract, but have a limited supports for numerics and object-orientation; they do not support concurrency. A sound theorem prover is available. It is compiled to efficient object code.
- Java and JML support design-by-contract and object orientation. They do not support advanced numeric types or concurrency. Their theorem prover is unsound. The compilation process comes with a significant run-time system and does not lead to very efficient code.
- Spec# supports design-by-contract and object orientation. It does not support advanced numeric types nor concurrency. Its theorem prover is sound.. At the time of writing, no compiler apt for the domain of interest exists.

The result of our analysis does not identify a clear winner, even if SPARK and C/ACSL are probably the most suitable solutions for the domain of interest. The main limit of these languages is the lack of support for object-oriented features.

4.1 Examples of a SPARK Verifying Model Compiler Usage

In this section we provide some simple examples on how the verifying model compiler philosophy can be of benefit in the development of high-integrity

software. The first two examples target Stateflow and Simulink models, while the third and fourth focus on Executable UML models.

The first example is taken from the February 2010 issue of "Communication of the ACM" [12]. In [12] the authors use a *separate* formal specification language to model the behavior of a microwave oven and prove a set of properties on it via model checking. A typical property is: "if the oven is running, then the door is closed". The control algorithm of the microwave oven is modeled in Stateflow. The proof technology used in [12] is NuSMV [13]. To demonstrate that the property was *not* satisfied by the input model and to produce a counter example, the NuSMV theorem prover produced and evaluated $9.8x10^6$ states.

The second example is taken from a Simulink model to implement the control algorithm for a window manager to be used in heads-up and heads-down displays for next-generation commercial aircraft [19]. Among the properties proved on the Simulink model, the most interesting are related to the logic of selecting the most appropriate unit to display the information to the user. The properties were again proved at model level by using model checking techniques.

Our main goal was to evaluate if we could translate the model *and* properties of interest in SPARK so as to use a *unique* representation to feed to the proof engine *and* the traditional compiler. Following our verifying model compiler vision, we wanted to (i) decrease the semantic distance between the executable and its formal specification used to prove properties and (ii) increase the confidence that the source code representation implements the model semantics by proving that the properties proved at model level still hold at source level. It must be noted that the SPARK proving technology is based on abstract interpretation and deductive verification rather than model checking.

We translated both the Stateflow model of [12] and the Simulink model of [19] in SPARK and added the properties of interest as post conditions of the subprogram implementing, respectively, the state machine logic and the control algorithm. For the Stateflow model, we were also able to fully model the state machine logic as a concatenation of logic expressions (again, as part of the post condition of the subprogram implementing the state machine logic). We used a prototype of the Simulink and Stateflow Ada back-end based on the Gene-Auto technology [15] and manually added SPARK annotations *a posteriori* on the generated code. The process of manually adding annotations was straightforward and we expect their automatic generation to be likewise. The SPARK code was translated into, respectively, 50 and 38 verification conditions. It is included in an extended version of this paper on the AdaCore web site [26]. We used a development version of the SPARK technology which can be coupled with the SMT solvers through the Victor translator [29].

The SPARK technology was able to prove the same properties and detect the same violations detected by the technology used in the original papers. SPARK was also able to provide precise conditions under which the violated properties do not hold. Such conditions are equivalent to the counter example produced by the NuSMV model checker. In addition, we were also able to prove that the properties of interest hold when the input model is slightly changed, again as suggested in the original papers. SPARK was of course able to prove also the absence of run-time errors, a task at which model checkers are usually not particularly effective.

The last two examples we list focus on the proof of absence of run-time errors in the code generated from an Executable UML model. Such experiments were successfully carried out by AdaCore clients using model compilers targeting SPARK on commercial applications in the defense [16] and energy domains [17] using different Executable UML modeling environments (respectively Mentor Graphics BridgePoint and Kennedy Carter iUML). In the cited projects, the generation of SPARK from Executable UML models permitted a zero-cost formal verification of absence of run-time errors in the functionalities encoded using an abstract action language which manipulates Executable UML elements. In particular, no specific constraints were imposed on the modeling standard to accommodate the generation of SPARK code and all required annotations were automatically generated. We acknowledge that this approach defeats the use of SPARK as a design-by-contract tool: the contracts are inferred automatically from the implementation in Executable UML. However, the purpose for the use of SPARK in this context was the proof of absence of run-time errors, rather than partial correctness: from

this point of view, the SPARK model compilers were extremely successful in complementing model-level verification activities with source level formal verification without requiring any modification to the users' modeling practice.

5. Developing Verifying Model Compilers

In the previous section we provided evidence that the verifying model compiler vision is feasible and has already been applied in industrial projects. We now focus on the evaluation of the impact of using formal programming languages as target for the model compilation chain in terms of:

- Development of the model compiler itself.
- Qualification of the model compiler in a DO-178 context.
- Certification in a DO-178 context of an application including generated code.

Our feedback derives from our involvement with the Gene-Auto consortium (ITEA 05018, [14]) for the development of a SPARK back-end for a sound subset of Simulink and StateFlow.

5.1 Impact on Model Compiler Development

While being constrained by SPARK semantic and syntactic limitations increased the initial definition of the code generation strategy, the use of a formal programming language as target for a model compiler caused surprising benefits during the development of the latter. The generated code produced by the prototype SPARK back-end for Gene-Auto had to be compilable and pass a small formal verification test. This requirement led to the discovery of several inconsistencies within the whole model compilation chain, including the part shared with the back-end for the C programming language. The errors we found included uninitialized or useless variables, dead code and unnecessary statements (for example, an if statement statically evaluable to False or True). Such discoveries permitted to greatly improve the Gene-Auto model compiler, with benefits for both the SPARK and C back-ends. In this particular case the same errors could have been caught by less formal verification tools (for example CodePeer [27] and Coverity Static Analysis [28]) which do not require the use of stringent semantics as for SPARK. However, the use of a formal language is still of benefit because it leaves the door open for more advanced formal analysis, in particular related to the absence of run-time errors and partial correctness (see the following sections).

5.2 Impact on DO-178 Tool Qualification

DO-178 does not discuss directly if/how the qualification of a model compiler could alleviate the verification activities on the generated code. An accepted approach in industrial practice (see for example [32]) is however to formalize low-level requirements in a model and rely on a model compiler qualified as development tool to:

1. Perform some verification activities (for example testing and structural coverage) on the model, typically via simulation: verification at a higher abstraction level is expected to be less costly.
2. Skip some verification activities on the source code (for example compliance with low-level requirements) because the generated code is expected to be faithful to its model-level specification.

In order to rely on model simulation instead of testing of executable code for verification activities, it is also usually necessary to produce evidence that the generated code is compiled to object code preserving the same functional properties. The whole qualification process of a development tool is extremely costly: for Gene-Auto the estimated qualification cost is around eight person years. The high initial investment in qualifying a model compiler as a development tool has been so far a barrier to the commercial availability of qualified model compilers.

In this paper, we propose an alternative path which could be considered to alleviate the qualification costs of a model compiler. Our approach does not consider for the moment how to provide evidence of property preservation from the generated source code to the (cross-)compiled object code. It is however worth noting that it is usually simpler to provide such evidence for formal programming languages because they employ a much simpler semantics and thus do not exercise particularly advanced or obscure features of a language/compiler.

The strategy we propose here to alleviate the cost of qualifying a model compiler is conceptually similar to

the "Unit Proof" methodology promoted by Airbus [25], which permits to eliminate most verification activities on source code if (i) all low-level requirements are expressed as formal properties, (ii) the formal verification framework is qualified accordingly to DO-178 as a verification tool. In a DO-178 context, qualifying a verification tool (like a theorem prover) is much less costly than qualifying a development tool (like a model compiler). The same applies also to Tool Qualification Levels in DO-178C.

The strategy we propose requires to:

1. Express low-level requirements at model-level, potentially using formal specification of properties. Formal specification of properties is not always necessary: for example, the logic of a state machine by itself formalizes low-level requirements without requiring the explicit production of formal properties. This point requires a modeling language able to model formal properties: this is the case of both Simulink/Stateflow and Executable UML (via OCL).

2. Verify that all model-level properties are correctly translated to the formal programming language targeted by the model compiler. Additional properties may be produced in the source code, for example to represent the logic of a state machine as a post-condition (see also the source code available in the extended version of this paper [26]).

3. Prove that all model level properties hold at source level. As show in section 4.1, we were able to formally express and prove model-level properties on SPARK programs.

4. Qualify the formal verification technology used at point 3 as a verification tool.

If the conditions above are all met, we can formally demonstrate that the generated source code complies with the low-level requirements formalized at model level, *without* qualifying the model compiler as a development tool. We thus believe that the verifying model compiler vision could permit to benefit of the advantages typical of a qualified model compiler (reduction of verification activities on source code) without actually requiring the qualification of the model compiler itself.

5.3 Impact on DO-178 Application Certification

The use of formal programming languages as a mean to decrease the cost of unit testing is increasing in the high-integrity domain: consider, in addition to SPARK, the use of tools like Caveat [20] and Astrée [21]. By targeting a formal programming language, we guarantee that the generated source code can be safely integrated with other (legacy) high-integrity components written in the same formal language. For example, the generation of SPARK assures that the same level of safety can be maintained across different software components and that global, application-level analysis can be performed. If we targeted classical programming languages, we would have limited the formal verification to a component-by-component basis and required manual analysis for the integration of manually written SPARK and generated Ada code. The model compiler vision fully guarantees that the proven advantages of using formal programming languages in a DO-178 certification context can be applied even when the application code is partially generated from modeling languages.

6. Related work

The use of intermediate (pivot) languages for formal verification has been already proposed and implemented in TOPCASED with Fiacre [22]. However, Fiacre is a verification-oriented pivot language: its use consists in being generated from modeling languages like AADL to verify some specific properties. The implementation of the source model into source code is then produced with a separate code generator which has neither visibility nor knowledge of Fiacre. Proof of semantic consistency between the generated source code and the formal model used for verification thus remains an open point. With verifying model compilers the verification and implementation-oriented views are consistent *by construction* because they collapse to the same representation. Such representation, for example a SPARK program, is indeed analyzable, executable and can be compiled to object code with mature technologies.

Another interesting approach closer to the verifying model compiler vision is the one proposed by ClawZ [23]. It provides formal evidence of semantic preservation of the refinement process from Simulink models to SPARK source code. This evidence is

provided by translating both the Simulink model and SPARK program into Z schemas and formally proving their equivalence. To achieve this result, it was necessary to define a formal Z representation for a subset of Simulink; a Z representation for SPARK is, on the other side, already available [24]. The main advantage of ClawZ is its verification tool nature, meaning that its DO-178 qualification would cost less than qualifying a code generator (which is a development tool). In addition it does not need to have any internal knowledge of the code generator, as it just copes with the initial input (the model) and the final output (the SPARK code). The approach proposed by ClawZ is conceptually similar to the one we propose, in the sense they both require formal programming languages to be the target of code generation: it is not a coincidence that SPARK is the target language of ClawZ. At this stage, the main advantage of the model compiler vision is the avoidance of using additional explicit formal representations (like Z) to prove property preservation. Recent advancements [33] on this technology suggest that the Z representations can be hidden.

Formal evidence of property preservation can also be obtained by construction if the model compiler itself is formally developed/verified. This is the approach applied for the CompCert [30] compiler and for the block sequencer in Gene-Auto. Both technologies were developed using Coq [31]. The main limitation of this approach is that the complexity and cost of the formal development may drastically limit the scope of the technology. CompCert for example has only PowerPC and ARM back-ends, has limited optimization features and contains several functionalities (for example I/O and parsing) which are not formally specified/verified. The Gene-Auto block sequencer is specified with 4500 SLOC of Coq (including 130 theorems) [34] and represents just a single step in the model compilation chain.

The last alternative solution is to consider just verification of absence of run-time errors on the generated code using tools like Polyspace or CodePeer. This approach clearly does not provide any evidence of property preservation. The same of course applies when trying to infer the faithfulness of the generated code to the model by simply comparing test results obtained via model simulation and execution of the generated code.

7. Conclusion

In this paper we have discussed the vision for verifying model compilers. Verifying model compilers promise improvements over the state-of-the-art in code generation by targeting a formal programming language which (i) can automatically provide evidence that property proved at model level still hold at source level and (ii) can complement model-level verification activities with source-level formal verification of platform-specific properties. We supported our point-of-view with limited experiments for Simulink and Stateflow models and with industrial-scale applications for Executable UML models. All experiments used SPARK as a target language.

In this paper we also evaluated different programming languages with respect to the requirement of being used as targets of a verifying model compiler. The SPARK and Frama-C frameworks emerged as the most effective choices.

Finally, we discussed the impact of targeting formal programming languages from several points of view, in particular (i) development process of a model compiler, (ii) DO-178 tool qualification and (iii) DO-178 application certification. The deployment of verifying model compilers brings benefit for all aspects above.

From a conceptual standpoint, the most valuable outcome of our discussion is the positioning of traditional programming languages within a model-centric development process. In contexts outside the high-integrity domain, source code may well be considered as a derived artifact with no interesting properties. On the contrary, if property preservation and platform-specific verification are significant concerns, the formal programming language plays a pivotal role within a model-based compilation chain, even though such intermediate representation may not be directly visible to the modeler. This requires finding a semantic mapping for the modeling language which lends itself to formal verification via automated theorem proving. Considering this aspect, it is important to note that the dominant needs when developing a (verifying) model compiler shall be those expressed by the developers of the model compiler rather than those of the final users. Given the cost of qualification/certification and that mature technologies exist to easily integrate modules written in different programming languages, there is no

reason to target a less-than-ideal formal programming language.

Formal programming languages and proof technologies may of course need to be enhanced to extend the set of properties they can prove and to ensure that properties can be preserved from model to sources. If this technical development is accomplished, verifying model compilers will ease the adoption of the model-driven paradigm, increase the confidence in modeled application, and improve the overall development process.

8. Acknowledgements

This work was carried out within the context of the LAMBDA project, part of the System@tic cluster. The views presented in this paper are those of the authors' only and do not necessarily engage those of the other members of the LAMBDA consortium. The authors gratefully thank the Gene-Auto consortium.

9. References

[1] OMG: "*UML2 Metamodel Superstructure*" http://www.omg.org/cgi-bin/doc?ptc/2004-10-05.

[2] Bordin M., Panunzio M., Vardanega T.: "*Beyond ASSERT: Increasing the Effectiveness of Model-driven Engineering*", DASIA 2009.

[3] OMG: "*SysML specification*" http://www.omg.org/cgi-bin/doc?formal/2007-09-01.

[4] OMG: "*UML profile for MARTE*" http://www.omg.org/cgi-bin/doc?ptc/2007-08-04.

[5] SAE: "*AADL*" – http://la.sei.cmu.edu/aadl/currentsite/aadlstd.html.

[6] Simulink: http://www.mathworks.com/products/simulink/

[7] Papyrus: http://wiki.eclipse.org/MDT/Papyrus

[8] SPARK: http://www.adacore.com/home/products/sparkpro/

[9] Frama-C: http://frama-c.com/

[10] JML: http://www.eecs.ucf.edu/~leavens/JML/

[11] Spec#: http://research.microsoft.com/en-us/projects/specsharp/

[12] Miller S., WhalenM., Cofer D.: *"Software Model Checking Takes Off",* Communications of the ACM VOL.53 NO.02 02/2010

[13] NuSMV: http://nusmv.irst.itc.it/

[14] Toom A., Naks T., Pantel M., Gandriau M., Wati I.: *"Gene-Auto: an Automatic Code Generator for a safe subset of Simulink/Stateflow and Scicos"* ERTS 2010.

[15] Gene-Auto/Ada: http://www.open-do.org/projects/geneautoada/

[16] Wedin E.: *"Applying Model-driven Architecture and SPARK Ada – A SPARK Ada Model Compiler for xtUML"* Reliable Software Technologies – Ada Europe 2010 Industrial Presentation (to appear).

[17] Curtis D.: *"SPARK annotations Within Executable UML"* Reliable Software Technologies – Ada Europe 2006, LNCS

[18] EUROCAE*: "Software Considerations in Airborne Systems and Equipment Certification"* - DO-178B, 1992, 1999.

[19] Whalen M., Innis J., Miller S., Wagner L.: *"ADGS-2100 Adaptive Display & Guidance System Window Manager Analysis"* NASA Contract Report available at http://shemesh.larc.nasa.gov/fm/papers/ADGS-2100WindowManagerAnalysis.pdf

[20] Caveat: http://www-list.cea.fr/labos/fr/LSL/caveat/index.html

[21] Astrée: http://www.astree.ens.fr/

[22] Berthomieu B., Bodeveix J., Farail P., Filali M., Garavel H., Gaufillet P., Lang F., Vernadat F.: *"Fiacre: an Intermediate Language for Model Verification In the TOPCASED Environment"* ERTS 2008

[23] Arthan R., Caseley P., O'Halloran C., Smith A.: *"ClawZ: Control Laws in Z "* ICFEM'00

[24] O'Neill: *"The Formal Semantics of SPARK83"* available on request on http://www.sparkada.com/sparkTechnicalReferences.aspx

[25] Souyris J., Wiels V., Delmas D., Delseny H.: *"Formal Verification of Avionics Software Products"* FM2009, LNCS

[26] Bordin M., Gasperoni F.: *"Verifying Model Compilers"* ERTS 2010. Full paper available at http://www.adacore.com/category/developers-center/reference-library/technical-papers/

[27] CodePeer: http://www.adacore.com/home/products/codepeer/

[28] Coverity Static Analysis: http://www.coverity.com/products/static-analysis.html

[29] Jackson P., Ellis B., Sharp K.: *"Using SMT Solvers to Verify High-Integrity Programs"* 2nd international Workshop on Automated Formal Methods, AFM07, 2007.

[30] Leroy X.: *"Formal Verification of a realistic compiler"* Communications of the ACM 52(7), July 2009

[31] Coq: http://coq.inria.fr/

[32] SCADE KCG: http://www.esterel-technologies.com/products/scade-suite/do-178b-code-generation

[33] O'Halloran C.: *"Guess and Verify – Back to the Future"*, FM2009

[34] Izerrouken N., Pantel M., Thirioux X.: *"Machine-Checked Sequencer for Crititcal Embedded Code Generator"* ICFEM09