UniComp: a semantics-aware model compiler for optimised predictable software

Federico Ciccozzi
Mälardalen University, IDT
Västerås, Sweden
federico.ciccozzi@mdh.se

ABSTRACT

In Model-Driven Engineering, executables are generated from domain-specific modelling languages (DSMLs) through two steps: generation of program code in a third-generation programming languages (3GLs, like C++ or Java) from a model, and compilation of the generated code to object code. 3GL code generation raises three issues. (1) Code generators are DSML- and 3GL-specific, hence they can not be used for other DSMLs or 3GLs than those they were designed for. (2) Existing code generators do not exploit model semantics; hence, 3GL programs do not always semantically reflect models. (3) Existing 3GL compilers are unable to exploit model semantics; hence, they are not able to operate model-specific optimisations. (2) and (3) seriously threaten predictability of the generated executables.

We advocate the need and provides a solution proposal for an innovative model compilation framework based on model semantics to produce executables without translations to 3GLs. Model compilation will be based on a common semantics, the Semantics of a Foundational Subset for Executable UML Models (fUML), and will semantically underpin any DSML whose execution semantics can be specified with fUML.

KEYWORDS

UML, ALF, fUML, compilation, model-driven engineering, predictability, semantics.

ACM Reference Format:
Federico Ciccozzi. 2018. UniComp: a semantics-aware model compiler for optimised predictable software. In *ICSE-NIER'18: 40th International Conference on Software Engineering: New Ideas and Emerging Results Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183399.3183406

1 PROBLEM AND VISION

Predictability is paramount for embedded systems with real-time and safety-critical requirements since failures or simply unexpected behaviours can lead to harmful situations, even threatening human life (e.g., airbag not deploying in time). An executable software (hereafter simply 'executable') is predictable when its actual runtime behaviour matches the foreseen one. Predictability does not only encompass functionality but also execution properties such as timeliness and memory management. Currently, executables are often generated from software models through a two-step translation (i.e., to program code first and object code after). This process does not entail model execution semantics[1], hence producing executables that may not be semantically consistent to source models. This leads to unexpected behaviours, which represent a major threat to predictability.

We advocate the need for novel techniques for compilation of software models (hereafter simply 'models') directly to predictable executables. Compilation is based on model semantics and *bypasses* translation to program code. Our hypothesis is two-fold: (1) a model compiler that exploits model semantics to produce semantically consistent executables can be built, and (2) bypassing translation to program code and directly compiling to object code gives better control on compilation and optimisation, thus allowing more predictable executables.

1.1 Motivation

Model-Driven Engineering (MDE) can effectively tame complexity, express domain-specificity, and support human communication [27]. Domain-Specific Modelling Languages (DSMLs) allow domain experts (who may not be software experts) to express complex functions in a more human-centric way than if using traditional programming languages. The Unified Modeling Language (UML) is the most widely used architectural description language [20], the de facto standard in industry [17], and an ISO/IEC (19505) standard. UML is general-purpose, but it provides powerful profiling mechanisms to constrain and extend the language to define DSMLs.

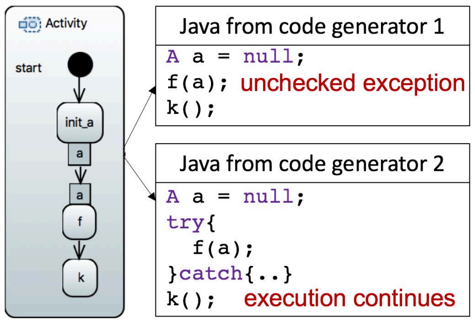
The Semantics of a Foundational Subset for Executable UML Models (fUML)[2] is a standard to define the execution semantics of UML (e.g., how control structures conditionally execute statements). fUML is intended for the definition of the execution semantics for any DSML based on UML profiles [32] or MOF [21]. In this paper we focus on DSMLs based on UML profiles (hereafter simply 'DSMLs').

In MDE, the **translational approach** is the current state of the art and practice for producing executables from DSMLs. By the translational approach we refer to a two-step translation: (i) from a model to program code conforming to a high level third-generation programming language (3GL), like C++ or Java, done by a so called **code generator**, and (ii) compilation of the generated 3GL program to object code, done by a so called **3GL code compiler**. The translational approach may seem to simplify the generation of executables by allowing reuse of existing 3GL code compilers/interpreters, but in truth it induces several issues [10]. Let us focus on the three most prominent.

(1) Code generators are DSML- and 3GL-specific. Currently, given a DSML, tailor-made code generators need to be provided in order to generate 3GL programs from it. This leads to the following issues. If different 3GLs are targeted, multiple code generators are

[1]Execution semantics of a language defines how the various constructs produce an executable behavior. In the paper we use *execution semantics* and *semantics* as synonyms.
[2]http://www.omg.org/spec/FUML/

required. Code generators can not be used for other DSMLs than those they were designed for. When a DSML evolves, related code generators need to individually co-evolve and that is a complex manual process, very costly and time-consuming in case of certified code generators [14].
(2) Existing code generators do not exploit model execution semantics [29]. Models are mostly considered by code generators as syntactical blueprints and execution semantics is inferred when translating them to 3GL programs. This semantics is 3GL-specific rather than DSML-specific. **This leads to 3GL programs whose behaviour does not always semantically reflect the modelled one.** Consider the example of a fUML model and the related generated Java code, in Figure 1. According to fUML semantics, if the input parameter `a` of an action `f` is `null`, the action is not called and the execution stops. Existing code generators do not consider this since they are not aware of fUML semantics. Simple code generators (1 in Figure 1) make a syntactical translation, which leads to `f(a)` being called with a potential unchecked exception (not caught by the 3GL code compiler!) and thus an erroneous behaviour. Smarter code generators (2 in Figure 1) enclose the method call in a `try/catch` statement for avoiding unhandled and unchecked exceptions; however, model semantics is not preserved since `f(a)` is called and the execution continues. Both cases result in unpredicted, potentially hazardous, behaviours.



Java from code generator 1

```
A a = null;
f(a);
k();
```

Java from code generator 2

```
A a = null;
try{
  f(a);
}catch{..}
k();
```

Figure 1: Semantic inconsistency

(3) Existing 3GL code compilers/interpreters take full advantage of the 3GL semantics for optimisation purposes, but are unaware of model semantics [7]. Consequently, optimisations on generated 3GL code, besides potentially violating model semantics, are unable to perform model-specific improvements. An example is compilers' dead code elimination, which cannot detect certain unreachable states in 3GL code generated from state-chart models, as shown in [7]. Optimisations based on model semantics and model-based analysis would be able to do this. Similar issues have been experienced for 3GLs, as in the Cfront translational approach for C++. The idea was to translate C++ to C to use existing C compilers. Cfront was abandoned in favour of real compilation of C++ because it resulted in suboptimal executables [23].

1.2 Vision

In the state of the art and practice, 3GL code generation from models is improperly referred to as "model(-driven) compilation" and 3GL code generators as "model compilers" (e.g., [3]). Differently from current approaches, we envision a real model compiler, which is DSML-independent.

Such a model compiler would semantically underpin and compile any DSML whose execution semantics is specifiable with fUML. Bypassing translation to 3GLs, we will target a single compiler internal language[3], from which different machine-specific executables can be generated. By exploiting model semantics instead of 3GL semantics, this model compiler will provide executables that are semantically coherent to the source models. The model compiler's ability to exploit model semantics will enable new kinds of optimisation based on such semantics, driven by model-based analysis, and performed at compile-time.

2 STATES OF THE ART AND PRACTICE

2.1 Most relevant related work

The only documented research effort related to our work is represented by [30], where the authors propose compilation of UML by defining a front-end for GCC and enhancing dead code elimination and block merging. The approach does not entail (i) a solution for different DSMLs through a unified execution semantics, nor (ii) complex behaviour definitions; in our approach we exploit the standard Action Language for Foundational UML (ALF, http://www.omg.org/spec/ALF/), which gives us an edge in comparison to this approach. While that work leverages only a limited subset of UML (simplified state-machines only), we aim at providing a solution based on the execution semantics brought by fUML.

2.2 Other relevant approaches

Generation of executables. Software modelling languages rely on translation to 3GLs as execution-enabler. This practice regards most well-known modelling languages, HRT-HOOD [22], SDL [18], Simulink/Matlab [11], Modelica [12], SysML [34], and EMF-based languages [13]. Even though providing 3GL code generation, most approaches improperly claim to provide model compilation. UML has witnessed the same trend. Generation of executables from UML, both in research and practice, has focused on translation to 3GLs. A typical example is[3], where authors provide translation from UML to C++ and, according to our definition, improperly call it "model-driven compilation". Executable UML (xUML) [26] also focuses on code generation to various 3GLs, similarly to Executable Translatable UML (xtUML), where model compilers are in fact translators from UML models to various 3GLs. This practice spanned over the most widely adopted UML modelling tools as well, among which the Mentor Graphics BridgePoint, IBM RSA family, Telelogic Tau, Sparx Enterprise Architect, Microsoft Visual Studio, MagicDraw UML, and Borland Together Architect.

Interpretation mechanisms for UML exist too [32], but they rely on resource-demanding middleware, thus falling short when deploying on embedded real-time systems.
Timing and memory analysis for predictability. In real-time and safety-critical embedded systems, optimisations for maximising timeliness and minimising memory failures are vital. Regarding timing, there is a fair body of research in the analysis and optimisation of models. Early timing analysis is crucial to shorten time-to-market, reduce costs, and increase quality of real-time applications [9]. Model-based timing analysis can be found in several domains: in automotive, with EAST-ADL [5], and AUTOSAR [2], in avionics, with AADL [31] and UPPAAL [1], in other domains,

[3]A compiler internal language, so called intermediate language or representation, is a low-level language similar to assembly.

MARTE [9] and SWEET [19]. However, results of analysis are only used to make model changes at design-time. Compile-time optimisation of generated 3GL code is not based on model-based analysis results nor aware of model semantics; hence, it is unable to make model-specific optimisations [7].

Research results in model-based memory analysis are scarce since modelling languages abstract from specific policies for the allocation of objects in memory at runtime; meaningful analysis is hard to achieve. In UML for instance, any object can be dynamically allocated on the heap by default to avoid disruptions of the type system. This is not in line with the best practices for safety-critical code; "The Power of Ten – Rules for Developing Safety Critical Code" by NASA [16] explicitly encourages minimisation of dynamic memory allocation. A smarter way to distribute objects between heap and stack is clearly a must in safety-critical applications. In the literature related to object-orientation, escape analysis has been proposed to check which non-escaping objects can be allocated on the stack [25]. Unbounded structure analysis identifies unbounded objects to bound to a fixed size for minimising dynamic memory handling [15]. However, the application of these methods to models has been neglected to date. Memory optimisation is based on 3GL semantics and done at compile- (e.g., [24] for C++) or at run-time (e.g., RTSJ [4] for Java).

3 UNICOMP: A SEMANTICS-AWARE MODEL COMPILER

We envision UniComp as a compilation framework applicable to any DSML whose execution semantics is definable through fUML. We use UML composite structures for describing structural information; fUML's annex "Precise Semantics of UML Composite Structures" prescribes the semantics for that. ALF is the standard textual notation for expressing algorithmic behaviours in UML; ALF's semantics is given by fUML. For modelling behaviours, we will use UML state-machines together with ALF; fUML's annex, "Precise Semantics for UML State Machines", prescribes the execution semantics for state-machines. MARTE, de facto standard UML profile for modelling timing and memory constraints, will be used for describing real-time and memory constraints at model level for model-based analysis. We will provide a proof-of-concept on two existing DSMLs: UML-RT [28] and CHESS-ML [9]. Both leverage UML composite structures and state-machines in different customised manners.

We build our model compiler on top of the Low Level Virtual Machine (LLVM) compiler infrastructure. LLVM is preferred over its competitors (e.g., GCC, Elsa or PCC) due to its API-based nature, which makes it very extensible and customisable. LLVM permits to define specific front- and middle-ends. Through a front-end, we will provide an efficient transformation from fUML to LLVM's low level compiler internal language (hereafter 'intermediate language') following fUML semantics. Through the middle-end we will enact compile-time optimisations, such as SSA-based optimisations and register allocations, based on model-based analysis results. Back-ends that produce executables for different targets (e.g., ARM, MBlaze, PowerPC) from the intermediate language will be reused since independent of the source language (fUML).

3.1 Providing a solution

In order to achieve our unified compilation framework, we need to perform the following three steps.

(1) Definition of compilation rules for fUML. We define compilation rules in terms of semantic mappings between fUML and the intermediate language as follows. Recall that we focus on *execution semantics*, as previously defined, and its preservation.

Consider fUML as a 5-tuple $L_m = <A_m, C_m, S_m, Msem_m, Msyn_m>$. A_m represents fUML's abstract syntax, C_m the concrete syntax, S_m the semantic domain, $Msem_m$: $A_m \rightarrow S_m$ the semantic mapping which gives semantics to syntactical concepts, and $Msyn_m$: $C_m \rightarrow A_m$ the syntactical mapping assigning abstract syntax elements to corresponding ones in the concrete syntax.

Similarly, the intermediate language is another 5-tuple $L_k = <A_k, C_k, S_k, Msem_k, Msyn_k>$. Compiling fUML to intermediate language, that is to say L_m to L_k, means to provide a transformation T: $A_m \rightarrow A_k$, between the two languages' abstract syntaxes. To provide semantic mappings $Msem$ between fUML and the intermediate language, we enhance the concept of semantic anchoring. More specifically, we link the semantic mapping of L_k to the semantic mapping of L_m through T as follows: $Msem = Msem_k \circ T$, such that $\forall x \in A_m, Msem_m(x) \cong Msem(x)$. The latter indicates that for each well-formed model x conforming to A_m, its semantics in L_m given by $Msem_m(x)$ is *essentially the same*[4] as the semantics of the generated executable $T(x)$ in L_k given by $Msem(x)$ (that is to say $Msem_k(T(x))$).

(2) Model-based analysis for time and memory. Compilation rules are complemented with reliable mechanisms for analysing models and guiding optimisation. Here we define algorithms for models to be analysed in relation to time and memory, and for analysis results to be used for optimisation purposes.

Model-based timing analysis will assess real-time constraints, which are described as MARTE decorations on model elements. We will investigate existing timing analysis exploiting MARTE [9] and SWEET [19]. Specific front-ends for fUML will be needed to run existing analysis on our models. Model-based memory analysis will drive memory allocation in the generated object code. We will define novel model-based memory analysis based on state of the art techniques for object-orientation, such as escape [8] and unbounded structure [15] analysis.

Results of timing and memory analysis are reported back to the user with hints on detected issues and possible model optimisations; automatic optimisation of models is not in the scope of our solution. Moreover, we will define algorithms for the model compiler to exploit analysis results to optimise executables (e.g., optimise the tradeoff between allocations on heap and stack). Note that such algorithms will be independent of the methods used to reach analysis results. This enables the possibility to include other analyses, not covered in this project, as long as their results can be formatted to input to our optimisation algorithms.

(3) Automating analysis, compilation, and optimisation. We define novel automated mechanisms for models to be analysed and transformed into executables, which are optimised according to analysis results. Analysis and compilation are achieved by model transformations.

For analysis, the model-based analysis algorithms defined in (2) will be realised in terms of model transformations that take fUML in input and provide analysis results in output. These results are back-propagated to the models, for the developer to inspect them, by in-place model transformations. For compilation, two sets of

[4]Minor deviations are tolerable if they do not alter expected data and control flows.

model transformations will be realised. One set will be in charge of realising the compilation rules defined in (1). The other set will realise the optimisation algorithms defined in (2).

Transformation unit testing [33] will be used to test model transformations. *Comparison of simulation and execution traces* will be used to assess the effectiveness of the model compilation framework in maximising semantic preservation, minimising dynamic memory handling, and optimising the tradeoff between allocations on heap and stack. The interpretation-based simulation environment for fUML, MOKA [32], that reproduces fUML semantics, will be used for gathering simulation traces. Runtime logging/monitoring mechanisms for LLVM will be defined for gathering execution traces. For timing, we will compare results from model-based analysis with those from object code flow analysis through the SWEET[5] analysis tool. For a more *formal verification*, we will investigate the combination of statistical model checking (e.g., UPPAAL-SMC [6]) and symbolic execution (e.g., KLEE[6]).

4 ENVISIONED IMPACT

We provide an innovative solution in terms of direct model compilation according to fUML semantics. The goal is ambitious, but its achievement will advance the state of the art and practice as follows:
– Providing model compilation, reasoning will focus entirely on human-centric artefacts, models, which will replace machine-centric documents, 3GL code, as advocated by MDE.
– Generated executables will be semantically more consistent to source models than with any existing solution based on translation to 3GLs. This will lead to functional predictability, which is pivotal in safety-critical applications.
– Any DSML whose execution semantics is definable in fUML will be compilable.
– Compile-time optimisations will follow model semantics thus performing model-specific optimisations; this will lead to time and memory predictability.
– Ability to produce more predictable executables will boost adoption of MDE for modern safety-critical systems (e.g., cyber-physical systems, IoT, autonomous systems).
– Bypassing translation to 3GLs will dismiss the need of extensive testing on generated 3GL code, thus decreasing time-to-market and effort in software development through MDE.

All the above represents significant contributions to efficient development of predictable applications.

5 EARLY RESULTS

We have provided a working solution for static flow timing analysis of a small subset ALF through the SWEET analysis tool. Analysis results are back-propagated to the ALF models too. In order to do that, we had provide semantic mappings between a small subset of ALF and SWEET's internal language, very similar to a compiler intermediate language. This showed feasibility and usefulness of mapping ALF to a much lower language directly, without translations to 3GLs.

[5] http://www.mrtc.mdh.se/projects/wcet/sweet/index.html
[6] https://klee.github.io/

REFERENCES

[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Procs of FORMATS*, pages 60–72. Springer, 2003.
[2] S. Anssi, S. Tucci-Piergiovanni, S. Kuntz, S. Gérard, and F. Terrier. Enabling scheduling analysis for autosar systems. In *Procs of ISORC*, pages 152–159. IEEE, 2011.
[3] T. Beierlein, D. Fröhlich, B. Steinbach, et al. Model-driven compilation of uml-models for reconfigurable architectures. In *2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES'04)*, 2004.
[4] G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, 2000.
[5] A. Bucaioni, A. Cicchetti, F. Ciccozzi, R. Eramo, S. Mubeen, and M. Sjödin. Anticipating implementation-level timing analysis for driving design-level decisions in east-adl. In *Procs of MASE*, 2015.
[6] P. Bulychev, A. David, K. G. Larsen, M. Mikučionis, D. B. Poulsen, A. Legay, and Z. Wang. UPPAAL-SMC: Statistical model checking for priced timed automata. *arXiv preprint arXiv:1207.1272*, 2012.
[7] A. Charfi, C. Mraidha, S. Gérard, F. Terrier, and P. Boulet. Does Code Generation Promote or Prevent Optimizations? In *Procs of ISORC*, 2010.
[8] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM TOPLAS*, 25(6):876–910, 2003.
[9] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega. CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *Procs of ASE*, 2012.
[10] F. Ciccozzi, T. Seceleanu, D. Corcoran, and D. Scholle. Uml-based development of embedded real-time software on multi-core in practice: Lessons learned and future perspectives. *IEEE Access*, 4:6528–6540, 2016.
[11] S. Feng, C. Driscoll, J. Fevold, H. Jiang, and G. Schirner. Rapid heterogeneous prototyping from simulink. In *Sixteenth International Symposium on Quality Electronic Design*, pages 141–146. IEEE, 2015.
[12] M. Flehmig, M. Walther, W. E. Nagel, I. Gubsch, J. Schuchart, and V. Waurich. Exploiting repeated structures and vectorization in modelica. In *Procs of Modelica*, number 118, pages 265–272, 2015.
[13] L. Geiger, T. Buchmann, and A. Dotor. Emf code generation with fujaba. *Fujaba days*, pages 25–29, 2007.
[14] T. Glotzner. Iec 61508 certification of a code generator. In *Procs of SSCS*, 2008.
[15] I. Grabe. *Static analysis of unbounded structures in object-oriented programs*. PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), Faculty of Science, Leiden University, 2012.
[16] G. J. Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6):95–99, 2006.
[17] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In *Procs of ICSE*. ACM, 2011.
[18] H.-U. Krieger. Sdl: A description language for building nlp systems. In *Procs of HLT-NAACL*, pages 83–90, 2003.
[19] B. Lisper. Sweet–a tool for wcet flow analysis. In *Procs of ISoLA*, 2014.
[20] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Trans. Software Eng.*, 39(6):869–891, 2013.
[21] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. xMOF: Executable DSMLs based on fUML. In *Procs of SLE*, 2013.
[22] S. Mazzini, M. D'Alessandro, M. Di Natale, A. Domenici, G. Lipari, and T. Vardanega. HRT-UML: Taking HRT-HOOD onto UML. In *Procs of ADA-Europe*, pages 405–416. Springer, 2003.
[23] S. Meyes. The Most Important C++ Software...*Ever*, 2006. http://www.artima.com/cppsource/top_cpp_software.html.
[24] K. D. Nilsen. Reliable real-time garbage collection of c++. *Computing Systems*, 7(4):467–504, 1994.
[25] Y. G. Park and B. Goldberg. Escape analysis on lists. *ACM SIGPLAN Notices*, 27(7):116–127, 1992.
[26] C. Raistrick. *Model driven architecture with executable UML*, volume 1. Cambridge University Press, 2004.
[27] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, Feb. 2006.
[28] B. Selic. Using UML for modeling complex real-time systems. In *Languages, compilers, and tools for embedded systems*, pages 250–260. Springer, 1998.
[29] B. Selic. What will it take? A view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.
[30] A. C. Smaoui, C. Mraidha, and P. Boulet. An Optimized Compilation of UML State Machines. In *Procs of ISORC*, 2012.
[31] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of aadl models. In *Procs of IPDPS*, 2006.
[32] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier. Formalizing Execution Semantics of UML Profiles with fUML Models. In *Procs of MoDELS*. 2014.
[33] A. Tiso, G. Reggio, and M. Leotta. Unit Testing of Model to Text Transformations. In *Procs of AMT*, 2014.
[34] B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat. Model-driven engineering of manufacturing automation software projects–a sysml-based approach. *Mechatronics*, 24(7):883–897, 2014.