Towards a model compilation framework based on a unified model execution semantics

Federico Ciccozzi
School of Innovation, Design and Engineering
Mälardalen University, Västerås (Sweden)
Email: federico.ciccozzi@mdh.se

Abstract—Due to the increasing complexity of software systems, model-driven engineering has been introduced to shift the developer’s focus from machine-centric program code to human-centric models of the software under development. In model-driven approaches, program code in conventional programming languages (e.g., C++, Java) is commonly generated from models and then compiled or interpreted. Intermediate translation of models to program code raises two fundamental issues: 1) semantic inconsistency and information loss between an executable and its source model, and 2) suboptimality of executables, since compilers are unable to exploit model semantics. These issues are not tolerable in embedded real-time and safety-critical applications. To tame them, we propose direct compilation of models bypassing intermediate translations to conventional programming languages.

Index Terms—UML, ALF, fUML, compilation, model-driven engineering.

I. INTRODUCTION

Model-Driven Engineering (MDE) represents an effective engineering approach to tackle the difficulty of third-generation programming languages (3GLs) to effectively tame complexity, express domain-specific concepts and support human communication [1]. Domain-Specific Modelling Languages (DSMLs) allow domain experts to develop complex functions in a more human-centric way than if using 3GLs.

The Unified Modeling Language (UML) is the most widely used architectural description language [2], the de facto standard in industry [3], and an ISO/IEC (19505-1:2012) standard. UML is general-purpose, but it provides powerful profiling mechanisms to constrain and extend the language to achieve DSMLs, so called UML profiles. The Semantics of a Foundational Subset for Executable UML Models (fUML)\(^1\) is a recent standard which defines the execution semantics of UML, e.g. the strategy by which modelled expressions are evaluated to values, or the way for modelled control structures to conditionally execute modelled statements. fUML is intended for the definition of execution semantics for any UML profile [4], but can be used for any DSML based on the Meta-Object Facility standard [5].

In MDE, the translational approach, also known as code generation, represents the current states of the art and practice for producing executables from models based on DSMLs. By the translational approach we refer to a two-step translation: (1) from a model to code conforming to a third-generation programming language (3GL, like C++ or Java), and (2) compilation of the generated 3GL program to executable form (or interpretation of the program).

\(^1\)http://www.omg.org/spec/FUML/.

Currently, given a DSML, ad-hoc code generators need to be provided in order to generate 3GLs programs from it (left-hand side of Figure 1). If different 3GLs are targeted, multiple DSML-specific code generators are required (e.g., dashed arrows from DSML\(_1\) to 3GL\(_1\) and 3GL\(_2\)). The three most prominent issues with this practice are:

- Code generators are DSML-specific. This leads to two issues. (i) When a DSML evolves, all related code generators need to individually co-evolve; co-evolution of code generators is a complex manual and error-prone process. (ii) Code generators can not be reused for other DSMLs than those they were designed for.
- Existing code generators do not exploit model execution semantics [6]. 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. This leads to 3GL programs which semantically do not always reflect the models.
- 3GL compilers/interpreters exploit 3GL semantics for optimisation purposes but are unable to entail model semantics [6]. Consequently, optimisations based on generated 3GL code are unable to perform model-specific optimisations [7].

Contribution. Our goal is to provide an innovative model compilation framework, specifically designed for leveraging model execution semantics to produce object code directly from models *without* translations to 3GLs (right-hand side

Fig. 1. State-of-the-art compared to our compilation framework
of Figure 1). Compilation will be based on a common execution semantics, fUML, rather than on specific semantics of single DSMLs. Hence, all DSMLs whose semantics is defined or definable through fUML will be directly compilable by our approach. Being able to exploit model semantics, the model compiler will provide executables which are semantically coherent to the source models. Moreover, it will enable a new kind of optimisation based on model semantics, driven by model-based analysis and performed at compile-time.

**Related work.** The documented research effort most related to our work is represented by [8], where the authors propose compilation of UML by defining a front-end for an existing compiling system (i.e., GCC) and enhancing dead code elimination and block merging. The approach does not entail compiling system (i.e., GCC) and enhancing dead code elimination and block merging. The approach does not entail complex behaviour definitions; in our approach we entail the elimination and block merging. The approach does not entail ing compiling system (i.e., GCC) and enhancing dead code elimination and block merging. The approach does not entail.

Our goal is to provide a model compilation framework for fUML, which is applicable to any DSML whose execution semantics is definable through fUML. We will provide a proof-of-concept on the UML profile for Real-Time (UML-RT) [9] since we focus on DSMLs in terms of UML profiles and real-time systems. UML-RT capsules, subcapsules, ports, connectors and protocols will be used for describing structure. State-machines together with ALF will be used for describing behaviours, and fUML for describing the execution semantics.

Instead of implementing an entirely new compilation chain from scratch, thus ditching years of community efforts in building reliable and flexible compiling infrastructures, we will leverage an established compiling system: the Low Level Virtual Machine (LLVM)\(^2\). Thanks to a language-agnostic design, LLVM allows us to define a specific front-end for fUML and at the same time exploit existing back-ends as well as defined compilation libraries. Doing so we can concentrate our efforts into providing an efficient compilation chain from source models to a low-level internal compiler language (i.e., intermediate representation language) supported by LLVM.

LLVM comes along with a single compiler language, LLVM-IR. Building a front-end for fUML in LLVM means to define how modelling concepts are meant to be syntactically and semantically mapped to the LLVM-IR. This step is crucial to minimise inconsistencies between models and the generated compiler language code. The challenge here is to properly tame expressive power and high dynamicity of fUML and to reflect it into a much lower level representation. Object-orientation, highly dynamic memory handling and object typing, implicit parallelism for statements and blocks of statements, sequence expansion mechanisms, are just few examples of the powerful features provided by fUML.

Once syntactical and semantic mappings from fUML concepts to LLVM-IR will be in place, we will define automated mechanisms for fUML models to (i) be analysed in relation to, e.g., timeliness and memory management, and (ii) be transformed into LLVM-IR, which can be compiled into executables. Model-based analysis is meant to drive compile-time optimisations.

Our compilation is meant to be based on automated model transformations, both endogenous in order to eventually prepare models for compilation and exogenous for achieving corresponding LLVM-IR. By exploiting model transformations we can also provide explicit trace links between source and target to be used for, e.g., model debugging, models@runtime, and to keep consistency between models and executables.

### III. Validation strategy

We plan to validate the compilation framework through the combination of (i) transformation unit testing, and (ii) comparison of simulation and execution traces. Model transformations from fUML to LLVM-IR will be validated through transformation unit testing [10], where defined test cases are run and actual generated output is compared to the expected one to check whether they match. Moreover, we will exploit the combination of an existing interpretation-based simulation environment for fUML, MOKA [4] in Papyrus, that accurately reproduces fUML semantics, and runtime logging/monitoring mechanisms. Through simulation, we aim at gathering model execution traces for our UML models. Runtime logging/monitoring mechanisms for the selected compiler will be defined (existing ones will be reused when possible) for gathering execution traces of the generated executable binaries. For a sample of representative models, model simulation traces will be compared to execution traces to assess the effectiveness of the compilation chain in maximising semantics-preservation, minimising dynamic memory handling and optimising the tradeoff between allocations on heap and stack.

Another essential aspect for a compilation chain to be practical is its compilation time and scalability. We will measure average compilation time on a significative number of runs, and compare it to state-of-the-art translational approaches; employing design models of varying size, we will be able to measure scalability too. Moreover, we plan to evaluate our solution towards multiple back-ends provided by the selected compiler, even in combination if possible.

### IV. Outlook

We have run preliminary experiments in manually mapping small portions of fUML and ALF to LLVM-IR with positive results. We manually reverse-engineered simple legacy C++ applications to ALF, mapped the resulted ALF to LLVM-IR and executed generated binaries obtaining the same functional execution traces as by executing legacy C++. Currently, we are working on static flow timing analysis of ALF through the SWEET\(^4\) analysis tool.

### ACKNOWLEDGMENT

This research work is partially supported by the MOMENTUM project (http://www.es.mdh.se/projects/458-MOMENTUM), funded by the KK-foundation.

---

\(^2\)http://www.omg.org/spec/ALF/

\(^3\)http://llvm.org/

\(^4\)http://www.mrtc.mdh.se/projects/wcet/sweet/index.html
REFERENCES


