Generation of Correct-by-Construction Code from Design Models for Embedded Systems

Federico Ciccozzi, Antonio Cicchetti, Mikael Krekola, Mikael Sjödin
School of Innovation, Design and Engineering - Mälardalen University, MRTC
Västerås, Sweden
Email: [federico.ciccozzi, antonio.cicchetti, mikael.krekola, mikael.sjodin]@mdh.se

Abstract—In a model-driven engineering development process that focuses on guaranteeing that extra-functional concerns modeled at design level are preserved at platform execution level, the task of automated code generation must produce artifacts that enable back-annotation activities. In fact when the target platform code has been generated, quality attributes of the system are evaluated by appropriate code execution monitoring/analysis tools and their results back-annotated to the source models to be extensively evaluated. Only at this point the preservation of analysed extra-functional aspects can be either asserted or achieved by re-applying the code generation chain to the source models properly optimized according to the evaluation results. In this work we provide a solution for the problem of automatically generating target platform code from source models focusing on producing code artifacts that facilitate analysis and enable back-annotation activities. Arisen challenges and solutions are described together with completed and planned implementation of the proposed approach.

Keywords—embedded systems, code generation, model-driven engineering, back annotation, model transformations

I. INTRODUCTION

Model-Driven Engineering (MDE) aims at easing the system development by using models as main artifacts. Providing abstractions of the real world, models can reduce the complexity of the problem and allow the developer to focus on the concepts that matter in the design of the application by reasoning in terms of domain-specific concepts [2]. The common goal of MDE approaches is to refine models among levels of abstraction through model transformations until target platform code is generated. Nevertheless, generation of code cannot be considered the final step of an MDE process, especially when extra-functional aspects of the system modeled at abstract levels are meant to be preserved at code level for achieving a correct-by-construction product. Especially in the embedded systems domain with its timing constraints and resource limitations, many of these aspects can only be analyzed using results gathered at execution time; the resulting computed values are back-annotated to the source models for further extra-functional evaluation and optimization [4].

Proper description, verification and preservation of such properties throughout the development process help in reducing final product verification and validation effort and costs by providing a generated code which is consistent to the modeling artifacts for further maintainability issues. In fact, managing extra-functional properties by means of design and evaluation at early development phases allows the developing team to have control over the preservation of such properties at product level [18], [20]. In order to preserve extra-functional properties and achieve correctness-by-construction, the generated code is not edited after its generation and therefore particular attention to the code generating transformation process is needed in order to produce code that facilitates analysis and enables back-annotation of computed values to source models.

With this work we propose a solution for automatic generation of target platform code from source models tailored to MDE processes that aim at guaranteeing that extra-functional concerns modeled at design level are preserved at platform execution level; the code is meant to be generated in a way that facilitates execution analysis by allowing controlled code injections for such purpose and enables back-annotation capabilities by maintaining explicit traceability and consistency between source models and generated code. The solution is easily maintainable, reusable and adaptable thanks to its multi-step structure and the introduction of intermediate artifacts for mitigating the differences in expressiveness between source and target languages.

The remainder of the paper is structured as follows. Section II identifies motivations and contribution in relation to those aspects of the proposed approach that have already been partially explored in the current state-of-the-art. In section III the actual proposed approach is unwound in all its steps and described in detail together with arisen challenges, adopted solutions and planned implementation; the intended validation process is also described. The current status of the implementation and planned next steps conclude the paper in section IV.

II. RELATED WORK

Automating the code generation phase is a task as delicate as important in an MDE process for embedded systems; if successfully achieved, it can positively impact economic factors, such as time-to-market as well as overall
costs and risks. Qualitative factors are also improved, such as correctness-by-construction of the generated application code and consistency between the different artifacts along the entire development process (i.e., models at different abstraction levels and generated code) [2].

An increasing number of European projects focuses on automatic software code generation for embedded real-time systems and is devoted to producing correct-by-construction code with preservation of extra-functional requirements, such as CHESS (cf. Acknowledgments) and Gene-Auto ITEA2 [9]. In fact, despite the numerous attempts in the literature to solve such problem, it still represents an open research issue.

In [3] the authors propose a code generation approach from AADL to tailored C for real-time embedded systems focusing on the need of flexibility in such generators. This supports the reasoning about our proposed multi-step solution, thought to be highly flexible and adaptable to different target platform languages.

The usefulness of introducing intermediate artifacts (i.e., intermediate meta-model in our solution) for mitigating the differences in expressiveness between the modeling and the target platform languages is confirmed by Alras et al. in [1] where a synchronous formal program by means of the Lustre++ programming language is used as intermediate representation of the system. In our solution we prefer to place such artifact at the same abstraction level of the design models in order to maintain domain-independence and enhance reusability. Moreover, in that work the authors state the ability of the process in guaranteeing the faithfulness of the generated code, although no actual proof is given. Our solution aims at asserting such faithfulness by generating correct-by-construction code which, enabling back-annotation of execution analysis results to source models, allows the developer to evaluate extra-functional properties preservation at modeling level.

Several works, such as [8], [10], [12], [19], [11], provide a similar solution to ours from an abstract perspective (i.e., using UML profiles, state-machine diagrams as source artifacts), but none of them focuses on generating code which enables back-annotation capabilities for a better evaluation of the extra-functional preservation from models to code.

III. THE PROPOSED SOLUTION

Starting from design models as source artifacts, our solution aims at providing automatic generation of target platform code through appropriate model transformations. Such models are designed using the CHESS Modeling Language (CHESS-ML), defined and currently under-refinement within the CHESS project (cf. Acknowledgments). According to the CHESS methodology, the system models are composed by a set of different enriched UML diagrams both for static (e.g., component, class, composite diagrams) and behavioral (e.g., state-machine, activity, sequence diagrams) design of the system under development. The actual behavior of the system is modeled in terms of UML state-machine diagrams enriched with action code that conforms to the Action Language for Foundational UML (ALF) [15]. Given the source models, our solution aims at providing a full code generation that entails both static and behavioral description of the system. While the translation of the static aspects of the system from models to code can be considered a trivial task, the same cannot be asserted for the translation of behavioral aspects; in this work we focus on the transformation process from behavioral design of the system to the corresponding C++ code specifically enhanced for enabling back-annotation activities and facilitating controlled code injections for analysis purposes.

The task of automating the generation of platform-tailored code does not only concern the actual transformation from design models to code since tracing information between model elements and generated code segments has to be defined to enable following back-annotation activities. As a consequence model transformations in charge of code generation must be properly defined by encoding apposite rules for the generation of explicit traceability links between source and target. Appropriate structures, such as an ad-hoc tracing model [13], provide concepts for maintaining traces between source models and corresponding generated code. The proposed solution, depicted in Fig. 1, relies on models and transformations as main artifacts for the generation of target code and is composed by four main tasks: (1) transforming the CHESS source model to a flattened ALF model (Fig. 1a), (2) generating the intermediate model from the ALF model (Fig. 1b), (3) generating the target platform code from the intermediate model (Fig. 1c), and (4) creating explicit traceability links between source model elements and generated code (Fig. 1d).

![Figure 1. From CHESS Source Models to Target Platform Code](image-url)
Transforming the CHESS source model to a flattened ALF model.

As previously mentioned, the CHESS model contains information on both structural and behavioral design of the system together with extra-functional properties definition and other complex decorations. In order to ease the code generation, such models could be lightened by taking into account only the needed information that affect the target code, while the ALF code used for the behavioral description is unwound into a flattened ALF model. Complex constructs typically exploited to improve code readability and compactness are mapped toward equivalent simple sequences of basic statements more suitable from a code generation perspective. This task is performed by an in-place model-to-model transformation (M2M) [6] using the Operational QVT (QVTo) [14] under the Eclipse Modeling Framework (EMF) [17]. Such transformation takes the CHESS source model as input and, after unwinding and pruning operations, gives a flattened ALF model as output.

Generating the intermediate model from the ALF model.

Since the ALF meta-model is syntax-driven, while usually target platform languages are semantics-driven, an intermediate model is needed to alleviate the expressiveness' differences between source and target languages and hence achieve a smooth and sound transition, in this case from the ALF model to C++ code. The intermediate model has to conform to a meta-model that we specifically defined for such bridging purposes. In Fig. 2 a portion of the intermediate meta-model is depicted and entities for specific purposes can be noticed: Process, for deployment issues, Class, Interface, Operation, for translation to the target object-oriented programming language, InjectionMarker, representing controlled injection of code for analysis purposes. The intermediate model is generated as output of a QVTo M2M transformation, that takes the flattened ALF model as input.

Generating the target platform code from the intermediate model.

Once the intermediate model is created, the final step of generating C++ code can be carried out by taking into account specific needs to be addressed such as enabling specific code injection for back-annotation and analysis issues. In Fig. 3 a small portion of the model-to-text transformation (M2T) [6] that takes as input the intermediate model and generates the actual C++ code is depicted; the transformation is defined by using the Xpand language [21] under the Eclipse Modeling Framework (EMF) [17].

![Figure 3. Model-to-Text Transformation in Xpand](image)

Creating explicit traceability links between source model elements and generated code segments.

Tracing information between model elements and generated code segments has to be defined for back-annotation, i.e. fundamental activities in an MDE process that aims at achieving correctness-by-construction [4]. Explicit trace links between model elements and code are created by defining a set of ad-hoc transformation rules within the transformation chain responsible for the code generation. Such rules create a mapping between each model element to its respective generated execution unit or code block by creating and populating a traceability model according to the constraints defined in the traceability meta-model; related artifacts and transformations are described in [5].

Instead of having a multi-step approach and hence introducing additional artifacts along the transformation chain, a solution consisting of a single direct transformation from model to code could have been adopted. While it might sound as a better and faster solution, a single and direct transformation negatively affects several aspects: (i) maintainability and reusability, since if any change is made at any level of the chain the whole transformation...
C++ code is partially implemented and already able to CHESS source model to the intermediate model, passing the intermediate meta-model has been defined and currently elements and generated code is still under development. Ability rules for creating explicit links between source model traceability meta-model has been defined while trace-back-annotation activities. Also been described with particular focus on the need of distribution taking into account those aspects which have already been partially explored in the current state-of-the-art have been described including those aspects which have already been partially explored in the current state-of-the-art have been described. The research work presented in this paper has been supported by the CHESS project (ARTEMIS JU grant nr. 216682 - http://chess-project.ning.com/).

ACKNOWLEDGMENT

The research work presented in this paper has been supported by the CHESS project (ARTEMIS JU grant nr. 216682 - http://chess-project.ning.com/).

REFERENCES