Enhancing the Generation of Correct-by-construction Code from Design Models for Complex Embedded Systems

Federico Ciccozzi, Mikael Sjödin
Mälardalen University, MRTC
Västerås, Sweden
{federico.ciccozzi,mikael.sjodin}@mdh.se

Abstract—Modern embedded systems are becoming more and more complex thus demanding for new powerful development mechanisms. Model-driven engineering has been recognised as a promising paradigm for the development of complex systems especially for its capability of abstracting the problem through models and then manipulating them to automatically generate target code. In our previous works, we presented mechanisms for the generation of 100% of the target code from UML models to be run on singlecore platforms.

In this work we provide possible solutions to enhance the generation process to entail a more complex set of platform configurations (i.e., multiprocess, multicore) as well as heterogeneous processing units (i.e., CPU, GPU).

I. INTRODUCTION

The research work presented in this paper focuses on the extension of a previously developed code generation process [6] in order to be able to produce target code to be run on complex platform configurations, such as multiprocess on singlecore, multiprocess on multicore. Moreover, possible further enhancements to entail heterogeneous platforms are introduced too. In fact, in spite of the positive preliminary results achieved in [6], the fact that simple “one size fits all” code generation does not work for complex platforms raised the need of adapting the generation process to entail these cases.

Model-Driven Engineering (MDE) aims at easing the system development by exploiting models as abstractions of the real phenomenon. In fact, the use of models can reduce the complexity of the problem and focus on the concepts that matter in the design of the system by reasoning in terms of domain-specific concepts [11]. The major objective of MDE is to provide automated generation of code to be executed on specific target platforms. Automating the code generation phase is a task as delicate as important in an MDE process for complex 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 code and consistency between the different artefacts along the entire development process (i.e., models at different abstraction levels and generated code) [2]. Correctness-by-construction is intended as the ability to demonstrate software correctness in terms of the approach exploited to generate it. In this respect the notion of correctness refers to the adherence of the generated code to what specified at model level, once the generation process has been validated [5]. Nonetheless the correctness of the user solution in the modelling space must be demonstrated for every model, e.g., by adopting model verification methods. Model-based analysis techniques may be employed for this purpose; anyhow the verification of models goes beyond the scope of this contribution.

Typically the code generation phase is considered as the conclusive step of an MDE approach [13]. On the one hand, in order for MDE to replace standard coding-based development processes, functional as well as extra-functional properties (EFPs) of the system modelled at abstract levels must be preserved in the generated code artefacts. On the other hand many EFPs, especially when dealing with limited resource systems (e.g., embedded systems), cannot be accurately predicted before code execution [16]. That is the reason why, in [6], we claimed that generation of code and its analysis should rather be seen as a possibly transitional step in the development. The values coming from executing or statically analysing the generated code are propagated back to the design models for a thorough evaluation of the system properties preservation from models to generated code. Possible editing of the models can then be performed and code re-generated until desired level of system properties preservation is reached and thereby correct-by-construction code is produced.

The two main activities collaborating for achieving preservation of system properties are: (i) fully automated transformation from design models to generate target code and (ii) back-propagation of code execution monitoring and/or static analysis results to the models. In [6] we provided

1In the remainder of this work we address the preservation of functional and extra-functional properties as “system properties preservation”.

2Throughout the paper, we intend design models as defined by means of the CHESS-ML, which is a UML-profile including a subset of the MARTE profile, and modelled following the component-based design pattern. Moreover, the behaviour specification is modelled in terms of UML state-machine diagrams enriched with Action Language for Foundational UML (ALF) code describing each operation’s behaviour (for more details please refer to [6]).
an automated round-trip engineering support for MDE of embedded systems with focus on ensuring system properties preservation throughout the entire development process, that is to say among the modelling levels down to the generated code. This automated process entails the generation of 100% of the source code from the design models, the creation of trace links between models and code, the monitoring of code execution and the propagation of collected values back to the models.

The outcomes of [6] have been extensively validated in industrial settings in order to evaluate its suitability as possible MDE replacement for standard coding-based development processes. This suitability has been asserted together with the necessity to extend the generation of code to be able to deal with complex platform configurations. In fact, in our previous works the focus was on the full\(^3\) generation of code and its consistency with the design models, while targeting simple platform configurations (i.e., singleprocess\(^4\) and singlecore). The ever increasing demand for computational power and parallelisation naturally leads to the deployment on more complex platforms and that is why the code generation has to be enhanced in order to fulfil these needs.

The remainder of the paper is organised as follows. Section II depicts the proposed extensions by means of three different platform configuration and deployment scenarios. A summary of the current state of the art related to the addressed problems is given in Section III. The paper is concluded in Section IV by the current status of the actual development together with the upcoming planned work.

II. Enhancing the Code Generation

In this section the planned extensions to the code generation are presented by means of three platform configuration and deployment scenarios (each of which incrementally enhances the precedent). In order to better understand the needed effort for the achievement of the proposed extensions, in Fig. 1.0 the platform configuration entailed by the transformation chain in [6] is shown. Two components, namely A and B, are deployed on a singlecore CPU. Consequently, the code is produced to be run on a singleprocess and singlecore platform with neither virtual nor actual parallelisation. More specifically, the code is generated as C++ and the communication between components is always exploited by means of intraprocess communication mechanisms. Our goal is to enhance the transformation chain to enable the generation of code runnable on the platform scenarios presented in the followings.

\(^3\)For full it is meant code that can be executed directly after its generation without any manual manipulation.

\(^4\)We refer to process as an independent execution unit that only interacts with other processes via interprocess communication mechanisms (managed by the operating system). Moreover, if several processes are deployed on a singlecore CPU, even if virtually parallel, their execution is serialised by the operating system.

Scenario 1: Multiprocess-Singlecore

The first enhancement step is represented by the possibility to generate code that can be virtually parallelised on a singlecore platform. The ability to handle virtual parallelisation and therefore interaction among processes via interprocess communication mechanisms is considered as provided by the operating system. In our proposed solution we aim at executing the generated application on the OSE Real-Time Operating System [8]. In Fig. 1.1 the deployment configuration of this first scenario is depicted. The two components are deployed on two separate virtual processes, namely VProcess 1 and VProcess 2, which are in turn deployed on a singlecore CPU (CPU 1). On the one hand, being able to deploy the application on virtual processes does not produce major improvements in performances since the code would still be running on a singlecore CPU. On the other hand, it positively impacts the analysability of the system under development and therefore the accuracy with which the level of EFPs preservation from model to code can be evaluated. In fact, by exploiting our round-trip approach proposed in [6] together with the enhanced monitoring capabilities of OSE proposed by Saadatmand et al. in [15] we can extrapolate valuable information about the values assumed by several EFPs (e.g., CPU load, execution time, heap and stack memory) when running the application.

Monitoring information are calculated for each single process. Consequently, deploying the whole software system on one single process does not lead to accurate performances evaluation of the single components, but rather gives an overall view at system level. That is the reason why virtual parallelisation through processes, even though on singlecore, increases the accuracy in evaluating extra-functional preservation from model to generated code. In fact, being able to model different deployment configurations by means of processes, produce code for it, monitor its execution and propagate the values back to the model, allow to optimise the deployment configuration and directly measure the impact on the produced code.

In order to achieve this enhancement the transformation process has to be modified to account the deployment configuration information that, together with the design model, drives the generation by directing each component to the right process and generating the communication code in order to distinguish between:

- Intraprocess communication: communication between components deployed on a same process is achieved by function calls;
- Interprocess communication: components deployed on different processes communicate via signals across processes.

Scenario 2: Homogeneous Multiprocess-Multicore

The advent of multicore platforms has produced a dramatical improvement on performances in the execution of complex...
software systems. The ever increasing demand for computational power and parallelisation in the embedded domain naturally leads to the adoption of multicore solutions. That is why we propose to enhance the code generation in order to produce code for software systems to be run on multicore (number of CPU cores > 1) sockets. In Fig. 1.2 the two components A and B are deployed on a N-core socket, namely CPU 1..N. Concerning the preservation of EFPs from model to code, being able to generate, execute and monitor code on multicore platforms and therefore achieve actual parallelisation amplifies the benefits introduced in scenario 1. In fact, we would be able to monitor the timing behaviours of tasks, as well as other properties like CPU load, execution time, memory usage, and thereby extract accurate runtime information that can be propagated back to the design model. At this point the goodness of both system design and deployment can be more accurately evaluated and consequently, when needed, modified for producing a more efficient implementation.

Software applications running on multicore platforms usually account different design considerations than applications running on singlecore platforms do. This issue becomes crucial when comparing the generation of multithreaded applications for singlecore (i.e., scenario 1) and for multicore. In fact the assumptions made in coding a multithreaded application for singlecore may not be valid on multicore. Especially two are the areas in which these differences arise: memory caching and thread priority management. Moreover, the models will have to entail information regarding independent sets of tasks that could be executed in parallel, and also identify temporal dependencies between tasks; all this information will have to be properly employed by the code generator. Regarding the communication, the paradigms introduced in scenario 1 will be addressed and enhanced where needed.

For these reasons, in addition to the information regarding deployment on process and communication methods introduced in scenario 1, the code generator has to be able to distinguish between single and multicore configurations and tailor the code accordingly.

**Scenario 3: Heterogeneous Multiprocess-Multicore**

In several domains (e.g., medical, automotive, aerospace) embedded systems are expected to process massive amounts of data, even in real-time. Aiding in fulfilling these expectations, the development of hardware technologies towards heterogeneous configurations makes embedded systems able to handle, e.g., very high input data rates. A typical scenario would be represented by input data coming into a multicore socket, which in turn may exploit one or more GPUs as coprocessors for parallel processing of large blocks of data. Such a technology shift from homogeneous to heterogeneous platforms raises a number of new research issues on both the modelling and coding of embedded systems. In Fig. 1.3 the deployment configuration of the third scenario is depicted. The two components are deployed on a heterogeneous platform composed by a multicore socket of CPU cores (CPU 1..N) and a number of GPU coprocessors (GPU 1..M).

On the one hand, the introduction of heterogeneity gives us the possibility to enable faster computation and generally increase the performances of the generated implementation. On the other hand it adds an additional level of complexity in the platform and deployment configuration and therefore complicates the code generation process. Due to the fact that CPUs and GPUs employ different formalisms and mechanisms for code execution, as well as different programming languages, the transformation process has to be enhanced in order to be able to map model entities to code artefacts written in different target languages (e.g., C++ to be run on CPUs and OpenCL to be run on GPUs). Moreover, the transformation will have to generate the communication code needed for the interaction between CPUs and GPUs.
the code generation to account these different levels of granularity, fine-grained deployment information has to be accurately modelled at design level. This could be done either by embedding tags within the ALF code defining the component’s behaviour or at component level by means of proper decorations of the design model, depending on the considered level of granularity.

III. RELATED WORK

Several different approaches aiming at achieving code generation for embedded systems can be found in the literature. 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 system properties, such as CHESS (cf. Acknowledgements). In fact, despite the numerous attempts in the literature to solve this problem, it still represents an open research issue. In [3] the authors propose a code generation from AADL to tailored C for real-time embedded systems focusing on flexibility of the code generator. This supports the reasoning about the multi-step approach proposed in our solutions, introduced in [6], thought to be highly flexible and adaptable to different target platform languages. The usefulness of introducing intermediate artefacts (i.e., intermediate (meta)model in our solutions) for mitigating the differences in expressiveness between modelling and target platform languages is confirmed by Alras et al. in [1]. In our solution we prefer to place intermediate artefacts at the same abstraction level as the design models in order to maintain domain-independence and enhance reusability. Several works, such as [9], [10], [12], [17], just to mention a few, provide solutions similar to ours from an abstract perspective (i.e., using UML profiles and state-machine diagrams as source artefacts), though not focusing on generating correct-by-construction code.

Currently, a number of different approaches have been proposed for the generation of multicore systems, starting from different abstraction levels, such as in [14], [4], [7]. Nevertheless, the input needed for these approaches is at a very low abstraction level and the output is meant to complement elsewhere generated or already existing code artefacts. In our solution the whole implementation is meant to be generated from the design models in one single artefacts. In our solution the whole implementation is meant to be addressed in a soon to come paper. The next step will consist in boosting the code generator, already upgraded to deal with scenario 1, to embrace actual parallelisation by producing code to be run on homogeneous multicore platforms (i.e., scenario 2). Regarding scenario 3, in [6] we presented an intermediate representation at modelling level (e.g., intermediate (meta)model) of the code to be generated. This multi-step approach enables the generation of different languages from the same intermediate model (currently only C++ is generated). The transformation process is meant to be extended in order to fully exploit this characteristic and, through design models and deployment configuration, produce multilanguage code for multicore.

IV. CURRENT STATUS & PLANNED WORK

Currently the development of the solution proposed in scenario 1 is at its final stages. The details on the improvements made are to be detailed in the future. The next step will consist in boosting the code generator, already upgraded to deal with scenario 1, to embrace actual parallelisation by producing code to be run on homogeneous multicore platforms (i.e., scenario 2). Regarding scenario 3, in [6] we presented an intermediate representation at modelling level (e.g., intermediate (meta)model) of the code to be generated. This multi-step approach enables the generation of different languages from the same intermediate model (currently only C++ is generated). The transformation process is meant to be extended in order to fully exploit this characteristic and, through design models and deployment configuration, produce multilanguage code for multicore.

V. ACKNOWLEDGEMENTS

This research work has been carried out within the projects CHESS (http://chess-project.ning.com/) and RALF3 (http://www.mrtc.mdh.se/projects/raf3/).

REFERENCES