Does Code Generation Promote or Prevent Optimizations?

A. Charfi, C. Mraidha, S. Gerard, F. Terrier, Pierre Boulet

Does Code Generation Promote or Prevent Optimizations?

Asma Charfi, Chokri Mraidha, Sébastien Gérard, François Terrier
CEA, LIST, Laboratory of model driven engineering for embedded systems, Point Courrier 94, F-91191, Gif sur Yvette, France.
{asma.charfi, chokri.mraidha, sebastien.gerard, francois.terrier}@cea.fr

Pierre Boulet
Université Lille 1, Sciences et Technologies
Cité scientifique, 59655
Villeneuve d'Ascq Cedex, France
Pierre.Boulet@lifl.fr

***Abstract*—This paper addresses the problem of code optimization for Real-Time and Embedded Systems (RTES). Such systems are designed using Model-Based Development (MBD) approach that consists of performing three major steps: building models, generating code from them and compiling the generated code. Actually, during the code generation, an important part of the modeling language semantics which could be useful for optimization is lost, thus, making impossible some optimizations achievement. This paper shows how adding a new level of optimization at the model level results in a more compact code. It also discusses the impact of the code generation on optimization: whether this step promotes or prevents optimizations. We conclude on a proposal of a new MBD approach containing only steps that advance optimization: modeling and compiling steps.**

Keywords-UML, Optimization, Compilation, code generation, RTES

I. INTRODUCTION

Real-Time and Embedded Systems (RTES) have become more complex with an increased number of product features. Their achievement has to satisfy the demand for shortened development times and higher expectations of product quality. Model-Based Development (MBD) [1] is an approach that takes RTES design into a higher level of abstraction, by using models at the center of the development process. Thus as illustrated on Fig.1, using a MBD approach, designers have to perform three steps: building models, generating code from them and compiling the generated code. Actually, a lot of systems are designed using the Unified Modeling Language (UML) [2] which is a general purpose language not specific to a domain. Thus, to model RTES, UML needs to be specialized. For example, the UML profile for MARTE [3] extends the capacities of UML for the modeling and analysis of RTES. MARTE provides facilities to annotate models with information required to perform specific quantitative analyses and allows to capture detailed design models of the application. Thus, models designed with MARTE, contain all the information about the system and its behavior, making possible an automatic code generation.

For several years, the code generation step was not fully automatic: designers had to complete the generated code by hand to get the ready-to-compile code. Currently, UML2 specification provides Actions and Activities packages that allow the full modeling of behavior making code generation a fully automatable step. Fully automatic code generation from high level specifications offers many advantages to RTES developers, including increased productivity, enhanced source code consistency and intends to improve performance of the generated system. Among those advantages, performance improvement is the most difficult to achieve. In fact, RTES are often constrained by their environment and the resources they own. Hence, an important problem to deal with in RTES development is linked to the optimization of their software part. Usually, MBD approaches rely mainly on compiler optimization frameworks to perform optimizations automatically. Although the enhancement of code generators and the use of optimizing compilers bring some answers to application optimization issue, most optimized compilers are still unable to perform optimizations related to the modeling language semantics. Actually, during the code generation, an important part of the modeling language semantics which could be useful for optimization is lost, thus, making impossible some optimizations achievement. We will show in this paper that compiler optimizations are not sufficient and that code generation is not a useful step for optimization issues. In this article, we will focus on memory optimization: an optimized code for us is a code that has the smallest size.

The paper is organized as follows. Section 2 presents compiler optimizations and their limits. The code generation impact on optimization is discussed in Section 3. Section 4 introduces a new approach to design RTES that consists in compiling optimized models directly to binary code. Section 5 discusses some related works and Section 6 concludes and presents some perspectives.

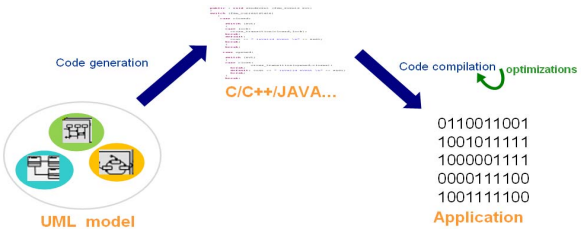

Figure 1. Classic Model Based Development approach

II. OPTIMIZATION IN THE COMPILING PROCESS

We are interested in C/C++ since this language is the most used language in the RTES design. The most widespread, well known and open source compiler is the Gnu Compiler Collection (GCC)[1].

A. GCC optimizations

GCC implements a large number of optimizations which all interact in complex ways and have a different impact on compilation time, code size, energy consumption, etc. It offers different optimization flags (-O1: first level of optimization, -O2: second level…). In our case, since we deal with RTES design and especially with code size concerns, we are interested in –Os flag (s listed for size). GCC optimizations are also classified according to their level of abstraction (Fig.2). There are low level optimizations (e.g., register allocation, peepholes optimizations, etc) that perform in RTL (Register Transfer Level) form and high level optimizations (e.g., dead code elimination, constant propagation, etc) that perform in SSA (Static Single Assignment) form [4].

Although GCC offers a lot of optimization passes (more than 100), RTES designers are still not satisfied with the quality of the compiled code. They still go through to hand-tune their code. In the next paragraph, we will present an experiment that exhibits one of the GCC optimization limits.

B. Compiler optimizations limits

We introduce an example of UML state machine diagram with unreachable state. UML state machine diagrams are used to specify the dynamic behavior of active objects that are widely used for RTES design. To build our state machine diagram, we have used Papyrus[2], an open source tool for graphical UML2 modeling. Our diagram (Fig. 3) contains 3 states, 2 pseudo states (initial and final states) and 5 transitions. We can notice that S2 is an unreachable state because it has no incoming transitions. Obviously, the code related to this state will not be executed and can be considered as a dead code. We said in Section 2 that one of the SSA optimizations is called *dead code elimination*. This optimization should be able to remove all dead code. Is GCC able to detect an unreachable state as a dead model part?

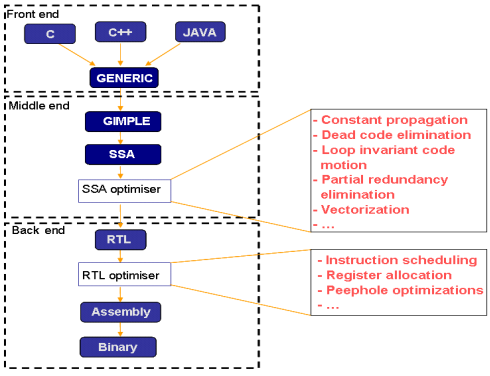

Figure 2. Current GCC optimization level

[1] http://gcc.gnu.org

[2] http://www.papyrusuml.org/

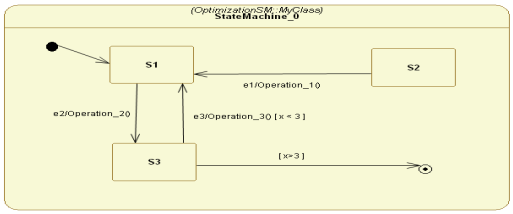

Figure 3. State machine diagram with unreachable state (S2)

Before answering this question, we have to generate C++ code from our state machine. There are numerous patterns that may be used to implement UML state machines. The most popular ones are: the State Pattern [5], the State Table Transition (STT) [6] and the Nested Switch Case statements [7]. The latter consists in having an outer case statement that selects the current state and an inner case statement that selects the appropriate state behavior given the type of the received event. Our code generator is based on the Nested Switch Case statements. However, in our previous work [8], a generation of C++ code from the 3 patterns was discussed. We have compiled the generated code using GCC 4.3.2 with –Os flag to get the smallest code size. For each optimization pass, GCC generates the correspondent file. For example, for the dead code elimination, we get file_name.dce (dce stand for dead code elimination). In the last dead code elimination pass (some optimizations are executed more than once to avoid conflict that may be created by other passes), we found that code related to the unreachable state is not removed. The size of the generated assembly code is 12669 bytes.

We can then conclude that GCC is unable to detect an unreachable state as a dead model part. Now, let's optimize the model by removing the unreachable state. From the new optimized model, we generate C++ code and recompile it. We obtain an assembly code measuring only 11393 bytes. It is true that the gain in term of code size is not significant (only 10.07 %), but this gain is proportional to the number of removed states/transitions. It depends also on the state machine kind. In [8] we have generated code from non optimized and optimized model with hierarchic state machine. We have found that optimization gain could exceed 45%.

C. Synthesis

GCC gets all the information about the system from the generated code. However, some optimizations need higher level information that is very difficult to obtain from SSA. These pieces of information exist in models but were lost by the code generation step. For example, the information that says "a state with no incoming transition is an unreachable state, so its code is a dead code" is lost when we move from model to code. Therefore, we have to exploit UML models semantics information before their loss. An alternative consists in exploiting this semantics information in the code generation step by implementing optimizations related to UML semantics information in the code generator.

III. OPTIMIZATION IN THE CODE GENERATION PROCESS

Section 2 has shown that the code generation step is the cause of the loss of some UML semantics information, preventing the compiler from performing some optimizations.

This loss of semantics is mainly due to the fact that programming languages, used as target of the code generation, are at a lower level of abstraction and therefore they do not provide the same concepts as the modeling languages. For example UML state machine diagram concepts (e.g., state, transition) are not directly represented in programming languages. For this reason, it is impossible to directly map modeling language concepts to the target programming language concepts. However, to perform optimizations related to UML semantics, code generator developers can incorporate the UML semantics information in the code generator. In this case, code generators do not only generate code from UML structural and behavior models, but also incorporate a set of rules to respect UML semantics.

A. Code generation optimizations

Most advanced UML tools provide template based code generators (e.g., Papyrus and Acceleo[3]). Acceleo is a MBD tool that permits to write code generators from UML to several languages: C/C++, Java, C#, etc. It was used to generate C++ code from our state machine diagram (fig.3). Thus, our code generator is a set of Acceleo templates. Some templates are written only for optimizations purpose. For example, the compiler optimization that consists in removing dead code (called dce for GCC) will be replaced by a template that completely inhibits the dead code generation. The optimization presented in section 2 (removing code related to unreachable state) will be performed by calling an Acceleo template that first detects unreachable states and then prevents the code generation from those states. It should be noted that to implement optimizations in the code generation process, we have to analyze the model. While, to implement optimizations in the compiling process, we have to analyze the code generated from the model. Since the model is more compact and does not contain parasite sequential code found in the third generation language, implementing UML optimizations during code generation is easier than implementing them in the compiler. It is true that such alternative could enhance optimizations in a MBD approach, but it also introduces defects into code generators.

B. Code generation optimization limits

Implementing optimizations in code generators will make them hard to maintain, more complex and hard to certify. Indeed, some sectors such as avionic and automotive sectors require certified code generators that produce certified code.

Moreover, one of the interesting code generators features, other than the fully automatic code generation, is the model debugging. Some code generators such as BridgePoint[4] provide a model debugger that enables the developer to debug an application using model-based breakpoints. However, if we decide to implement optimizations related to the UML semantics in the code generator, model debugging will not be an easy task. In fact, optimizing during the code generation is likely to widen the gap between the initial model and the generated code. For example, we consider the optimization that consists in removing the unreachable state. If we implement this optimization in the code generator, the generated code will not contain a trace of the unreachable state, while this same state still exists in the model. Thus, model debugging will be a tricky task just like debugging the code that has been optimized by any of the programming language compilers. Aside from all these drawbacks, implementing optimizations in code generators is dependent from the target language as well as the model implementation pattern. Thus, if we change the target language (from C++ to Java for example) or the model implementation pattern (from State Pattern to STT in case of state machine diagram for example), we have to re-implement those optimizations. Even if we do not implement optimizations in code generators to avoid all drawbacks listed above, generating code from UML models can influence as well the compiler optimizations process that relies mainly on the control flow graph (CFG) of the generated code. In fact, given the C++ representation (sequential form) of the code to be compiled, GCC has to build the CFG of this sequential form to perform SSA optimizations. However, in the model level, we have already the CFG expressed by the state machine diagram. Thus, instead of generating code from model (moving from CFG to a sequential form) and then moving from the sequential form to the CFG to implement the optimizations, we would better save the first CFG provided by the state machine diagram and perform on it compiler optimizations. We have to mention here that CFG generated from code respects UML model semantics and did not undergo any transformation issues. On the contrary, most of traditional compiler and analysis tools have been based on CFG generated from code to perform optimizations. However, moving from UML CFG to programming languages sequential form and then rebuild CFG from code is still a redundant and time consuming step.

C. Synthesis

The UML ability to express the data flow as well as the control flow can be considered as an argument to defend the approach of compiling directly models (get the binary code from models by avoiding the code generation). In addition to that, the bad impact of code generation step in optimization process (mainly the prevention of some optimizations, the loss and the rebuild of the CFG which is very useful to perform high level optimizations) leads us to adopt a new approach of MBD that skips the code generation step and generates directly binary code from optimized models (Fig. 4).

IV. Optimized Model Compiling Process

A. Model compiler architecture

Several existing works such as [9] and [10] encourage skipping the code generation step in a MBD approach. According to Jacobson, one of the founders of UML, this approach brings the advantages to avoid the unnecessary use of two higher level languages: UML and a third generation language [10]. The code generation step has long been used with great success to make UML models productive. But, now, UML is becoming so expressive (with profile and action semantics) that it can be considered as a "programming"

[3] http://www.acceleo.org/

[4] http://www.mentor.com/products/sm/model_development/bridgepoint

language. Thus, nowadays, the only difference between an executable UML model and code generated from it is that the generated code can be compiled (using one compiler among a huge list of existing compilers) and subsequently executed. While for the executable model, although some works are interested in model simulation and execution [11][12], we can not find until now a tool that compiles directly UML models and can be compared to third generation language compilers in terms of analysis, debug and optimizations. The lack of tool support is the former limitation that can hamper the widespread adoption of skipping the code generation step. The latter is a social barrier. Indeed, programmers will probably be very reluctant to use model compilers. They will not accept the idea of programming in UML. It is just like the difficulty that inventers of high-level programming languages have been faced to when they have tried to convince assembly programmers that there is no more need to program directly in assembly code.

In this paper, we try to avoid the first limitation: the lack of model compilation tools support. To do that, we will not build a model compiler from scratch, we will instead, reuse existing compiler intermediate forms. Thus, we will be able to exploit their optimizations and their common features like transforming assembly to binary code, common analysis, debug etc. As a consequence, our model compiler can have comparable performance characteristics as existing third generation languages compilers. Since we have studied GCC compiler architecture and identified intermediate forms devoted for optimizations issues (Section 2), our model compiler will reuse the GCC back end. In other words, we will try to build an UML front end for GCC (Fig. 5). A *front end* purpose is to read the source file, parse it and then convert it to an intermediate representation. Fig.2 shows 3 different GCC front ends: C, C++ and java front end. The transformation from an intermediate form to assembly is called *back end*. The *middle end* encapsulates all other transformations between intermediate representations mainly for optimization purpose. In GCC, the first intermediate form is called GENERIC [4]. It is a generic AST (Abstract Syntax Tree) for all front ends. GIMPLE is a simplified form of GENERIC to facilitate optimizations and SSA (Section 2) is a modified GIMPLE that is suitable for most GCC optimizations. To build our UML front end (Fig.5), we can target one of the 3 intermediate forms listed above. We can instead, transform our UML model to another intermediate form. But to reuse all pertinent GCC optimizations, we have to join at the end one of the GCC intermediate forms. Obviously, this target form could not be the RTL form, because if we move from our optimized model directly to the RTL form, we will escape all the interesting SSA optimizations. Moreover, Since GENERIC, GIMPLE and SSA have the same metamodel and the same structure (tree), we would better target the GENERIC form, and GCC will perform for us the two other transformations: GENERIC to GIMPLE to SSA.

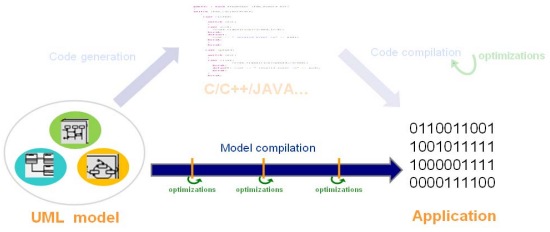

Figure 4. New Optimization Based Approach for RTES

Although compiling directly UML models to binary code overcomes the bad impacts of the code generation on optimization process, this approach did not resolve compiler optimizations limits (Section 2). In fact, since we will reuse the GCC middle and back ends, only optimizations that are not related to UML semantics can be performed (SSA and RTL optimizations in case of GCC). So, the compiler still unable to perform optimizations related to UML semantics. An alternative consists in optimizing the model before compiling.

B. *Optimization in the modeling process*

A model optimization is a kind of model refactoring [13]. It is a model transformation that guarantees the transition from non optimized model to an optimized model by keeping unchanged the behavior of the model. Several model transformation engines have been developed such as [14]. Ref. [15] is a new proposed open source component under Eclipse EMFT for model refactoring. It applies some state machine refactorings (e.g., removing isolated state, removing redundant transition). Those refactorings are similar to the transformations provided by our optimization tool. There is not yet a corresponding tool support in Eclipse for creating and applying refactorings. Thus, we have decided to build our own optimization tool (developed with java). In its current version, the users choose manually the optimizations to perform. We plan to improve it in a way that it automatically executes optimizations that correspond to the UML model.

C. *Synthesis*

We propose an optimization-based approach for RTES design that contains only steps that are useful for optimizations issues: modeling and compiling steps. Since code generation does not contribute in optimization issues, skipping this step in an optimization-based approach for RTES design is natural. This results in a new MBD approach that compiles directly models into binary code (Fig.4). Our model compiler will be a UML GCC front end that reuses as they are existing GCC optimizations. Writing a GCC front end is not a complicated task. However, building a UML front end which is

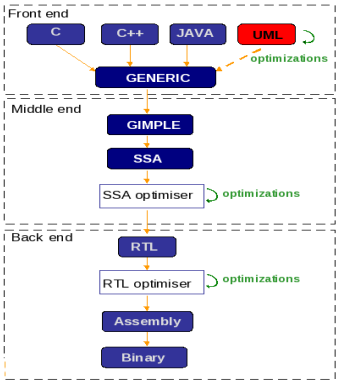

Figure 5. UML GCC front end

able to deal with the language in its entirety is not an easy task, especially if we consider all concepts of the UML that is always qualified as non formal and ambiguous language. To specify a formal semantic for a subset of UML, OMG has defined fUML[16]. fUML (foundational UML) includes only class diagram to model structures and activities diagrams to model behaviors. Thus, transforming our UML models to fUML can be an interesting step since fUML is a well formed subset of UML and it defines a virtual machine (not code generation) process to execute models. We can also use it to simulate our models before compiling them. As future work, we intend to explore the benefit of transforming our models to fUML for optimizations issues.

V. Related Works

There are other approaches that are not satisfied by compiler optimizations such as [17] and [18]. In [18], authors argue that optimizations in modern compilers are constructed for a predetermined set of primitive types. Thus, programmers are unable to exploit optimizations for user-defined types. They are also unable to incorporate new optimizations in the compiler set of optimizations since it is a fixed set. As a solution, authors propose to use the code reuse mechanisms of generic programming to make compiler optimizations easier to write and to apply them more broadly. Their framework uses ROSE to produce AST from C++ code. Then, they optimize the CFG produced from the AST. Based on this modified CFG, generic optimizations are written in the specification language Scheme [18]. Although this approach, like our approach, is not satisfied with modern compiler optimizations and intends to enhance them, we disapprove the idea of qualifying the compiler optimization set as fixed set. In fact, we have studied GCC optimization, and we are convinced that adding an optimization to the GCC optimizations set is feasible thanks to GCC plug-in structure.

There are a number of languages and tools that are interested in optimizing the code generated from UML models. For example, xtUML [12], which is a subset of the UML with defined execution semantics, offers the ability to translate UML model directly into 100 % complete and optimized code. In xtUML, the model is compiled using a set of design pattern "*archetypes*" to be applied in code generation. Archetypes coupled with a run time library and a translation engine form a model compiler. It should be noted here that the term model compiler did not refer to an engine that generate directly the binary code. It just means that the code (C, Java...) was automatically generated from the model. BridgePoint and iUML/xUML[5] are examples of model compilers. A model compiler accedes to a repository that captures the semantic representation of a model. Thus, UML semantics information, required by some optimizations, will be visible to the model compiler. Model compiler's archetypes are changeable, so programmers can optimize their code by manipulating them: if they find that some rules do not generate sufficiently efficient code, they can modify the rules. However, given that all optimizations are implemented in a single process (model compiling), model compilers are likely to become complex and hard to maintain systems.

[5] http://www.kc.com/products/iuml.php

VI. Conclusion and Perspectives

In this article, we have discussed the optimization issues in a MBD for RTES. The MBD approach consists in performing three steps: building models, generating code from them and compiling the generated code. In this classic approach, code optimizations are only done in the compiling process. Most of compilers offer a lot of optimization passes, but they are still unable to perform some optimizations.

The contributions of this paper are twofold: we have presented cases where we can not rely only on compiler optimizations and we have proven that code generation, although it is an automatic step in the MBD approach, could prevent performing some optimizations. This concludes on a proposal of new MBD approach that consists in compiling directly models to binary code. Our future works will be focused on how to build a model compiler that is as performing as a third generation language compilers.

References

[1] B. Schätz, A. Pretschner, F. Huber and J. Philipps, “Model based development of embedded systems,” Lecture Notes in Computer Science, Springer, vol 2426, 2002, pp.331-336, 2002.

[2] OMG, OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2, Beta 1, 2008, http://www.uml.org/.

[3] OMG, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.0, 2009, http://www.omg.org /spec/MARTE/1.0/PDF/.

[4] D. Noville, GCC—An Architectural Overview, Current Status, and Future Directions, Proc, the Linux Symposium, Vol2, Canada, 2006.

[5] E.Gamma, R.Helm, R.Johnson, J.Vlissides,Design Patterns: Elements of Reusable Object-Oriented Software,Addison-Wesley, 1995.

[6] R.C.Martin, UML Tutorial: Finite State Machines, C++ Report, 1998.

[7] P. Metz, J. O'Brien and W. Weber, Code Generation Concepts for Statecharts Diagram of UML v1.1, Object Technology Group, 1999.

[8] A. Charfi, C. Mraidha, P. Boulet, S. Gérard and F. Terrier, "Toward optimized code generation through model-based optimization", Proc, Design Automation and Test in Europe, DATE 2010., in press.

[9] B. Selic, New Methods and Tools for Developing Real-Time Software, Panel Position Statement, ISORC, 2009.

[10] I. Jacobson, keynote address to CMP Media's UML World conference, New York City, June, 2001.

[11] H. Wada, J. Suzuki, M.M.B. Eadara, A. Malinowski and K. Oba, Design and Implementation of the Matilda Distributed UML Virtual Machine, Proc, Software Engineering Applications, 2006.

[12] S.J Mellor, M.J. Balcer, Executable UML: A Foundation for Model Driven Architecture, Addison Wesley, ISBN 0-201-74804-5, 2002.

[13] Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley, 1999.

[14] F. Jouault and I. Kurtev. Transforming Models with ATL, Proc, Workshop Model Transformations in Practice Jamaica, 2005.

[15] E. Biermann, C. Ermel and G. Taentzer, "Precise Semantics of EMF Model Transformations by Graph Transformation", MoDELS, 2008.

[16] OMG, Semantics of a Foundational Subset for Executable UML Models, Beta 1, 2008, http://www.omg.org/FUML/1.0/Beta1/PDF.

[17] P. Gottschling, A. Lumsdaine, Integrating Semantics and Compilation Using C++ concepts to develop robust and efficient reusable libraries, GPCE, 2008.

[18] J.Willcock, ALumsdaine, D.Quinlan, "Reusable Generic Program Analyses and Transformations", GPCE, USA, 2009.