ITEA

Information Technology for European Advancement

WP 1.5 - Code Generation Synthesis of Different Approaches and Corresponding Issues (D1.5.1)

Version 01 - Public

Edited by Anita Orhand

Software Development Process for Real-Time Embedded Software Systems (DESS)

ITEA COMPETENCES involved:
1) Complex Systems Engineering
2) Communications
3) Distributed Information and services

June, 2000

Table of Contents

1 INTRODUCTION **3**

1.1 PURPOSE OF THIS DOCUMENT 3

2 WHY USE CODE GENERATION **4**

2.1 CONTEXT 4

2.2 MOTIVATIONS 4

3 ISSUES FOR REAL TIME EMBEDDED SOFTWARE **6**

4 RELEVANCE TO COMPONENT ORIENTED PROGRAMMING **7**

5 CONCLUSION **8**

1 Introduction

1.1 Purpose of This Document

The DESS work package numbered 1.5 is dedicated to study the impact on the efficiency of code generation technology when using high-level abstractions, such as UML architecture diagrams or state machine charts in order to develop real-time embedded systems. This document is the first deliverable of this work package and it is intended to synthesize the issues faced by the DESS partners when putting into practice code generation techniques.

In order to track the achievements and pitfalls of code generation, the partners of this work package were asked to describe the following points:

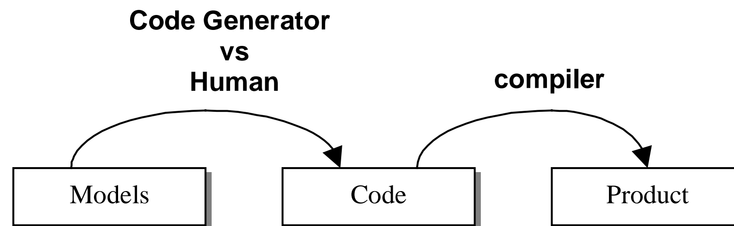
- The context in which the code generation is put into practice in their organization,
- The development method (process) followed when using the code generation techniques
- The pros and cons the described approach

This document is the synthesis of each of partners' contributions, focusing on the real time and embedded aspects. After a description of the motivations for code generation, we list the critical issues related to this technology for real-time embedded software. Then, current solutions adopted by partners are presented with their benefices and problems, in particular relatively to the topics listed above.

2 Why Use Code Generation

2.1 Context

Code generation is a great challenge for today developers since it will allow locating most of their job on the modeling of the system they have to develop. Modeling is an exciting and a valuable activity for both the developers and the company since it increases knowledge of individuals about the systems produced.



Between the models and the products exists an intermediary state which is the source code. While the transformation of the code into the product is a mature technology (e.g. the compilers) the transition from the model to the code is still an immature technology. Maybe this phenomenon is a direct consequence of the too recent adoption of a standardized modeling formalism such as UML? However, this question remains out of the scope of this study.

Relying on an automated tool to go from one form to another form (such as the code to the product) allow to ensure that the resulting form is still consistent with its source. Indeed, most of the issues with all the artifacts produced during the development lifecycle are related with the consistency between then. The typical example is the consistency between the documentation of the code with the code itself. It has been proven that the better solution for this problem is to rely on an automated generation of documentation from the source code.

Obviously, the transition from one form to another one requires most of the time a human intervention in order to insert the additional value the new form is supposed to provide. Back to the documentation generation example it means that the generator may extract from the source code only the prototypes and that a human intervention should complete the document by providing textual explanation of the semantics of each prototype. In the case of the code generation this additional information may be defined either in the generated code after the generation occurs or in the model itself using some dedicated facilities of the modeler and exploited by the code generator.

Even if the code generation is not a mature technology a lot of on going efforts promise to achieve very interesting results in the near future. It should be noted that the code generation is not restricted to generation of code from abstract models (UML, SDL, etc) such as the one described in most of the partner's contributions. Indeed, the GUI generators and all visual programming environments, such as LabView for digital signal processing, are tremendous contributions in the domain of code generation.

2.2 Motivations

What leads to the use of code generation technology is mainly the necessity to have a development process fast and flexible.

Reduce Development Time and Cost

Speed is required because markets are moving very rapidly, and thus products have to get out quickly. The automation from design to code, the use of high level model allowing a large cut in the testing stage by model checking, allow speeding up the whole development time.

Support for Iterative Process

Flexibility, because the processes also have to handle very late design changes. The use of high-level simulation models that are transformed into executable code allows this flexibility. Changes in the

system requirements can be incorporated in this formal model and quickly propagated to the implementation.

Robustness against Hardware Changes

Flexibility also because more and more embedded applications run in an open environment, where the change of target hardware must be handled with a minor impact on the logical design. A development environment with a clear separation between system design and hardware configuration allows to work easily on new devices.

3 Issues for Real Time Embedded Software

Code generators have been for a long time used for prototyping a system under development. But for targeting, the code generated does not meet some issues that are of first importance for real-time embedded software:

Resource Constraints

- Time: code generation includes frequently an overhead that leads to a loss of efficiency during execution.
- Memory: memory is an important constraint when addressing mass-market products and memory blueprint must be reduced as much as possible.

Hardware Configuration

- Consideration of specific features of the target hardware (limited processing power, peripheral module, etc): the code must be used on the target hardware without manual adaptation and optimization.
- Support for multiple hardware: the architecture of the hardware for which code generated should not be hard-coded within the tool but should be used by description templates, which prevents from having a specific tool for each micro-controller used.

Model Checking and Analysis

- Model: use of a high-level specification language (e.g. StateChart, ESTEREL) which is not specific for the code generation tool.
- Analysis: an analysis of the timing behavior of models can allow guaranteeing real time property. Worst Case Execution Time can be computed, properties such as deadline respect, mutual exclusion respect, etc can be checked.

4 Relevance to Component Oriented Programming

The code generation technology is a somewhat a tool allowing to build systems by emphasising the modelling of the systems and relying on automatic transformation of the model into code to be compiled. One side effect of an effective code generation technology is that the reusability issues are shifted from the programming step to the modelling step.

As a consequence, the unit of reusability changes in a model compared to what is done in a program. Reused artefacts in a program are libraries and source-code whereas it becomes portion of model such as frameworks and components in model-oriented programming.

The effective reuse of models by means of self-contained components is a very complex task. Code generation is just one means that will enable this approach, but it turns out to be an important one.

Code Generation is relevant to component oriented programming because code generation produces systems from models and models are made of architectured components.

Components and Statecharts

Using statecharts, the functional structure of a system can be expressed by so-called *activity charts*, which are basically blocks communicating with the environment by signals flowing into and out of them. Although activity charts do not comply in every detail with the elaborate definition for components as given in WP1.4 of the DESS project, it should be possible to reuse previously defined activity charts.

At this point, however, we are facing several (partly conflicting) aspects:

- Components that are not only delivered as binary code but also equipped with an abstract model could easily be incorporated into the above mentioned design flow.
- If code generators can also be used for components then these are (at least partly) independent from the target hardware and could be reused in several environments.
- Protection of the intellectual property of a component from an external supplier restricts the introspection of the component.

Despite the promising approaches using model based design flows in conjunction with code generators for abstract models, an effective reuse strategy following this approach still needs much more attention and further research.

As statecharts are one of the languages of the UML notation one might assume that any code generation solution for statecharts could directly be used for UML as well. Although there is some truth in this observation, there are still a number of limitations.

- Statecharts do not have a standardised semantics (as opposed to e.g. SDL). The semantics associating with the statecharts is much more provided by the supporting tool set than a clear semantics of the graphical language. A formal and standardised semantics is not available for UML so far.
- Statecharts are only one of several diagrams within UML. Thus, they only cover a subset of the complete description of a system. An advanced code generation should also take the other diagrams into account, e.g. sequence diagrams for scheduling purposes.

Again, although the path towards a successful use of code generation in the scope of UML is visible, further research efforts and standardisation activities have to be performed.

5 Conclusion

The purpose of this document is to make a synthesis about the issues faced by methods, techniques and tools when it comes to using code generation from high-level model. This synthesis aims to be based on DESS partners' practical experience through their use of commercial or in house developer approaches.

The commercial offers related to code generation remain both emerging and immature or dedicated to a specific application domain (SDL, Esterel, etc.). Therefore, a common trend among partners seems to be either to create their own framework or to extend existing toolset. Concerning the critical issues for real-time embedded systems, they focus particularly on:

- taking into account the specific features of the target hardware,
- using a high level specification language relevant in the domain of embedded systems (Statechart, ESTEREL language),
- analyzing the timing behavior of models.

Nevertheless, we have little feedback in term of efficiency of the generated code. A widely spread idea about the code generation effectiveness is that it depends not only on the performances of the tool but it depends also on the expressiveness of the models and the design of the processed models. Indeed it seems doomed from the start to expect tools that will be able to produce very efficient code, or at least effective, from badly designed models and/or poorly expressive model formats.

Therefor, as preliminary conclusion for this report, we believe that we should define very precisely the scope of responsibility for the code generation tool in the goal of producing efficient code. Besides, we should also define what we may expect from modeling language and design guidelines in order to craft convenient code generator.

To end with this preliminary study we have found that there is a strong connection between code generation and component notation. Indeed, component notation impacts very much on the way a system's model is built and the model is the input of code generation. Consequently, the component notation promoted by the partners in the context of this project should also take into account code generation issues.