Chapter XIV

Practical Considerations in Automatic Code Generation

Paul Dietz, Motorola, USA
Aswin van den Berg, Motorola, USA
Kevin Marth, Motorola, USA
Thomas Weigert, Motorola, USA
Frank Weil, Motorola, USA

Abstract

Model-driven engineering proposes to develop software systems by first creating an executable model of the system design and then transforming this model into an implementation. This chapter discusses how to automatically transform such design models into product implementations for industrial-strength systems. It provides insights, practical considerations, and lessons learned when developing code generators for applications that must conform to the constraints imposed by real-world, high-performance systems. This deeper understanding of the relevant issues will enable developers of automatic code generation systems to build transformation tools that can be deployed in industrial applications with stringent performance requirements.
Background

Commercial software development still largely follows processes based on the software development life-cycle paradigms that were introduced in the late sixties as software engineering faced the *software crisis*: the methods and skills employed for designing software did not match the emerging hardware capabilities. While many variations to the original *Waterfall* paradigm have been put into practice, the poor performance of software engineering processes continues to undo the advances in processing capabilities due to Moore’s law. The primary criticisms that have been lobbied against the software life-cycle model still hold today: software development begins with an attempt to recognize and understand the user requirements and then proceeds to implement a software system that satisfies those requirements. The requirements specification is formulated in a dialogue between users and system analysts. Typically, the requirements definition reflects the developers’ interpretation of the users’ needs. Where the communication of these needs has been distorted, either by preconceptions or by general unfamiliarity on either side, it is unlikely that misunderstandings become apparent until the user tests (examines) the near-ready product. Thus, maintenance is performed at the implementation level. At this point, the programmers have applied considerable skill and knowledge to optimize the code. However, optimization spreads information: it takes advantage of what is known elsewhere in the system and substitutes complex but efficient realizations for the simple abstractions of the specification. The result is a system more difficult to understand due to increased dependencies among its components and scattering of information. Correcting errors deriving from the requirements phase during software maintenance becomes exceedingly expensive.

As soon as these problems became apparent, new methodologies were proposed; see Agresti (1986) for a collection of seminal papers criticizing the waterfall model and presenting alternative approaches. An *operational specification* is a system model than can be evaluated or executed to exhibit the behavior of a software system at an early development stage. *Transformational implementation* is an approach to software development that uses automated support to apply a series of transformations that change a specification into an implementation. Together these techniques promise to overcome the limitations of the Waterfall model: an operational design specification avoids errors in the requirements phase by demonstrating to the users the system behavior during an early stage of the software development process. Transformational implementation enables the rapid realization of the specification and the maintenance of the product at the level of the specification. Together these techniques have become known as *model-driven engineering*. Weigert and Weil (2006) discuss the benefits achievable when applying model-driven engineering to the development of real-time embedded systems and give data from industrial experience.

Requirements and design verification through operational specifications became feasible by the advent of standardized executable specification languages such as SDL, or more recently, UML. These languages have been given operational interpretations and are widely supported by tools that allow users not only to develop the model of the software system, but also to exercise it against test cases, and observe and analyze the resultant behavior of the software system. Tools are able to transform the high-level specifications into implementations in popular target languages: see Czarnecki and Eisenecker (2000) or Tsai and Weigert (1993) for an overview of approaches to system implementation through program transformation. Nevertheless, commercial software development has not been able to capi-
Related Content

CoPBoard: A Catalyst for Distributed Communities of Practice
Gilson Yukio Sato and Jean-Paul A. Barthès (2010). *International Journal of Software Science and Computational Intelligence* (pp. 52-71).
[www.igi-global.com/article/copboard-catalyst-distributed-communities-practice/39105?camid=4v1a](www.igi-global.com/article/copboard-catalyst-distributed-communities-practice/39105?camid=4v1a)

Relevant and Non-Redundant Amino Acid Sequence Selection for Protein Functional Site Identification
[www.igi-global.com/article/relevant-non-redundant-amino-acid/43896?camid=4v1a](www.igi-global.com/article/relevant-non-redundant-amino-acid/43896?camid=4v1a)

Artificial Intelligence Techniques for Unbalanced Datasets in Real World Classification Tasks
[www.igi-global.com/chapter/artificial-intelligence-techniques-unbalanced-datasets/56154?camid=4v1a](www.igi-global.com/chapter/artificial-intelligence-techniques-unbalanced-datasets/56154?camid=4v1a)
Case Technology for Psycho-Diagnostic System Generation
www.igi-global.com/chapter/case-technology-psycho-diagnostic-system/38489?camid=4v1a