Science of Computer Programming 77 (2012) 81–82

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Editorial

Introduction to the Special Issue on Automatic Program Generation for Embedded Systems

This special issue contains three papers devoted to the automatic generation of programs for embedded systems. *Automatic program generation* is a form of *meta-programming*, where a high-level program is automatically generated for the programmer using some program generator tool. Embedded systems are a natural target for automatic program generation. Program generator tools may range from simple pre-processors through to complete implementations of new languages, or to tools that build a *software product line* from libraries of components, tailoring the final software to the specific needs of the product. The special issue arose out of a workshop on automatic program generation for embedded systems, held in Salzburg, Austria in late 2007. Following the workshop, an open call was issued for extended, journal quality papers to explore key issues associated with this emerging and increasingly important research area.

As a form of machine translation, automatic program generation is related to compilation. It differs in a number of respects. Firstly, the generator produces code in a high-level programming language, such as C, C++, Java, ML or Haskell. This must then be compiled or interpreted so that it can be executed on the target platform. Secondly, automatic program generation often involves programmer intervention to direct the generator tool. In some approaches, such as the staged meta-programming used in Meta-ML, it may even involve the programmer writing rules that govern how meta-level source is translated to the high-level target.

The main motivation for using automatic program generation is to provide abstractions/features that are not present in the target language, so reducing the cost of code development/maintenance, enhancing flexibility and improving time to market. As a way of translating high-level programs, automatic programming has a number of advantages over conventional compilation. The main advantages are that program generators are much simpler than full compilers, and so can be written much more rapidly; that full advantage can be taken of an existing language implementation, so existing front-end and back-end tools, code generators, optimisers etc. can all be exploited without needing to be re-developed; that generated code automatically complies with the code standards or language requirements that may be enforced by some companies or other organisations; and that the developer is not bound to a (possibly small) language developer, but can continue to use generated code even if the generator tool is no longer supported or available. The main disadvantages are that errors are often reported in terms of the target language and it may also be necessary to modify/specialise the generated target code. It follows that the programmer must have a working understanding of both source and target language, and be able to relate errors and code from the two levels.

The boundary between program generation and compilation becomes a little blurred when very high-level languages use a high-level language as a compilation target. This is becoming an increasingly important technique, since it allows reuse of code generators and other parts of the compiler tool-chain without needing to integrate with the internals of a specific compiler. Done properly, this considerably reduces the effort required to produce a compiler for a new high-level language, and can dramatically enhance both reliability, performance and portability. A related approach is that of *Embedded Domain Specific Languages*, where the compiler for the new source language is developed in a host language, essentially extending the host language constructs to cover those of the new language. This brings additional advantages, since all host features can be used directly by the new domain-specific language. In this special issue, we consider these and similar techniques to be part of the broad spectrum of automatic program generation.

Embedded systems form a rich target domain for automatic program generation techniques. The embedded systems landscape is presently dominated by the use of C. This ensures maximum portability for what may be very obscure architectures and platforms, and also provides good access to low-level operating system and hardware features, without significantly impacting performance. However, the use of C has its own drawbacks, notably to do with memory management, use of pointers, the ease of introducing bugs, and the relatively low level of abstraction, meaning that much repetitive or long-winded code may be necessary. Deploying program generation techniques means that high-level programming abstractions

0167-6423/$ – see front matter
doi:10.1016/j.scico.2011.11.001

may be used without sacrificing any of the many advantages of C. Indeed, tools may include cost or memory information that would take some effort to derive by hand.

The special issue contains three papers. Each of these papers throws new light on some specific aspect of this increasingly important area. In their paper on "Automatic Library Migration for the Generation of Hardware-in-the-loop Models", *Ryssel, Ploennigs and Kabitzsch* show how a generative programming technique can handle the diversity of test libraries, simplifying their management by the programmer. The approach is illustrated using MATLAB/Simulink examples taken from the automotive domain, where a software product line is built from systems components. The authors have been able to demonstrate both good performance and good quality in the examples that they have studied. Because the technique is based around a functional building-block approach, these results should extend beyond the automotive sector to a variety of other domains. This clearly demonstrates the generality of generative programming approaches.

In their paper on "Separate Compilation of Hierarchical Real-Time Programs into Linear-Bounded Embedded Machine Code", *Ghosal et al.* consider how to compile a hierarchical extension of Giotto so that it targets a virtual machine for embedded systems, the E-Machine. This involves flattening the hierarchy. However, if this is done naively, there may then be a code explosion. Here, the use of automatic program generation is as part of the compilation process for the hierarchical code. The technique produces a version of the source code that eliminates the hierarchical structure, generating E-Machine code that is linear in the source input.

Finally, in their paper on "Compositional Design of Isochronous Systems", *Talpin et al.* introduce an approach to balance the tradeoff between performance and precision by introducing a formal design methodology that supports non-blocking composition of a specific class of processes, those that support weak *endochrony*, that is processes with an internal notion of time. Importantly, the approach is able to reuse most of the Signal tool suite, which has been commercialised by TNI. This gives an easily exploitable tool set, a classic outcome from a generative approach.

We hope that the reader derives as much pleasure in reading these papers as we had in editing this volume.

Kevin Hammond
School of Computer Science,
University of St Andrews,
St Andrews, Scotland, UK
E-mail address: kh@cs.st-andrews.ac.uk.

Paul H.J. Kelly
Department of Computing,
Imperial College,
London, UK
E-mail address: P.Kelly@imperial.ac.uk.

Available online 6 November 2011