Research Issues with Application Generators

Premkumar T. Devanbu
Software and Systems Research Laboratory,
AT&T Bell Laboratories
2b417, 600 Mountain Ave, Murray Hill, NJ 07974
Tel: (908) 582-2062
Email: prem@research.att.com

Abstract

Application generators, which generate domain-specific systems from specifications formulated in a special-purpose high-level formal language, can achieve very high levels of software reuse and impressive gains in productivity. Several tools available in the market have greatly simplified the construction of application generators. There still are, however, significant difficulties in the practical use of application generators. I'm particularly interested in the pursuit of methodological and technical solutions to the following problems: *designing domain-tailored specification languages*, *re-engineering domain specific frameworks out of existing systems*, and *assistance for debugging specifications.*

Keywords: domain analysis, application generators, reverse engineering

Workshop Goals: Keep in touch with the state of the art in reuse.

Working Groups: Domain analysis/engineering, Design guidelines for reuse - general, Ada, and/or C++, Reuse and OO methods, Reuse and formal methods, Useful and collectible metrics

1 Background

An application generator (AG) produces applications from high level specifications (See Figure 1.) The specification is written in a language that is likely to have powerful, compact, domain-specific constructs; the application generator will lex and parse this specification into an internal parse tree representation; this parse tree will then be checked for semantic correctness, consistency etc; from this validated parse tree representation of the specification, code is generated; this code can be linked up with an architecture, library and/or run time environment (which we collectively call a *domain specific framework* (DSF)) to implement the application. Application generators have become increasingly popular in industry, particularly with the development of tools to build application generators such as MetaTool [1], CENTAUR [2]. Commercially available tools such as MetaTool provide facilities for generating parsers, built-in data structures that can represent parse trees, and various traversal operators for traversing the parse tree. MetaTool provides a "template driven" approach to code generation that is particularly useful for code generation. Newer tools such as CENTAUR provide (in addition to parser generators) semantic checkers that are driven by formal specifications.

Application generators improve productivity in several ways. First, because of the power of a well-designed specification language, the size of a high-level specification to implement a given application is likely to much shorter than a corresponding implementation in a conventional programming language. Secondly, a high level of reuse can be achieved, more or less automatically; the applications re-use all of the domain-specific framework. In a recent survey paper [3], Biggerstaff cites some impressive gains in software productivity achieved through the use of application generators.

There are hower, some significant technical issues with application generators, which form the main concern of my research and this position paper.

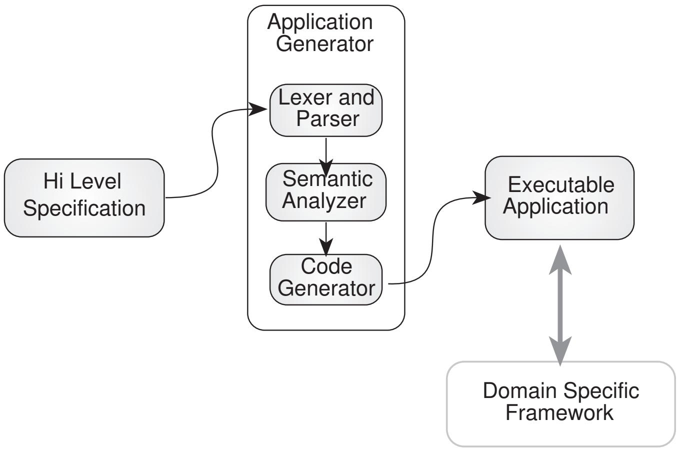

Figure 1: A Typical Application Generator

2 Position

There are several issues that arise in the practical deployment of AG technology:

- *Domain-Tailored Specification Languages* How does one design a specification language for a given application domain ?
- *Scavenging the Domain Specific framework* Can an existing application, system or library be adapted to serve as the framework for the application generator ?
- *Developing/Maintaining Specifications* How does one ensure that a specification is correct and/or consistent ? If there are errors/bugs in the application, how can they be identified in terms of the specification language ?

2.1 Specification Languages

The most important and difficult part of creating an AG is the design of a suitable high level specification language. Little or nothing is known about how to go about creating such a language.

Clearly, most applications in an application domain have a lot in *common*; this common part is of little concern to the application creator; thus the specification language doesn't need elements to describe this part. The purpose of the specification of an application is to describe what is *different and special* about a particular application. The design of the language should include features necessary to characterize the *variation* in the domain. Consider the case of LALR parser generators, the *function*, or *algorithm* is the same for all applications (parsers); the difference between applications is solely the *grammar* that is being parsed; hence the specification language for `yacc` [4] looks a lot like a context free grammar specification. Thus, the *type of variation* in the domain should drive the design of the specification language.

I am interested in developing an methodologicalapproach to designing a specification language; a first step, we are examining numerous examples of successful application generators to determine broad classes of specification languages, and the linguistic features particular to each class. We hope to develop a characterization of the relationship between the type of variation in the domain of application, and the type of the applicable specification language and its features. With this characterization, we hope to develop a *prescriptive methodology* that would enable a domain analyst to use a more systematic approach to design a specification language for a given application domain.

2.2 Creating a domain specific framework

Tools such as MetaTool, CENTAUR, etc, have simplified parsing, semantic analysis and even code generation. However, one element of the application generator, the domain specific framework, must still be built the old fashioned way. This framework must capture all the most general aspects of the domain in a library and/or architecture.

This framework must contain much of the domain software functionality required by the applications that the AG produces. For example, in a process control domain the domain specific framework must contain all the necessary drivers for the various sensors, actuators, etc; also, the scientific subroutine libraries for the necessary physical/chemical computations. GENOA [5] is an application generator that produces source code analyzers. Source code analyzers created by GENOA can extract useful information from source code. For this system, we needed a framework that lexes, parsed, and semantically analyzed source program code and builds an *annotated parse tree*, which can be traversed and analyzed.

Generally speaking, a framework for an application generator would have to be implemented from scratch. However, an *existing application may contain within it the seeds of a domain specific framework*. For example, an existing control system for a thermostat may contain driver software for various types of sensors and actuators, day/night timing event generators etc. Likewise, in the language tool domain (relevant to GENOA) an existing compiler may contain a lexer, parser, semantic analyzer etc. In such cases, it may be possible to scavenge the domain specific framework out of the existing system for more general use.

We have used this approach successfully in the GENOA system. GENOA is a *portable* analyzer generator, that can be attached to any existing lexer/parser/analyzer for a given programming language; this attachment is performed by writing a formal specification in a companion system, called GENII. A GENII specification describes the linguistic elements of the language to be analyzed (Expressions, Declarations, Statements etc) and their implementation in the specific front end (data structures, operations, etc). Using this specification, GENII basically places a standard "wrapper" around the existing implementation. This "wrapped" implementation, and associated (generated) tables are used by GENOA as the domain specific framework for generated application. In this way, we have created an analyzer generator for C++, called GEN++, based on the Cfront compiler. Thus in the case of GENOA, the "framework scavenging" process has been formalized and automated to some extent. Because of the complexity of the C++ language, and the attendent difficulty of parsing it, this approach is particularly advantageous. In fact, we believe that GEN++ is the first and only available[1] tool of this kind for C++.

We believe that this approach to extracting a domain specific framework out of an existing application could be viable if the existing application is implemented in a modular fashion, and the "generic elements" within it are of high quality, and if the cost of implementing a DSF from scratch are quite high. If a well understood, formal (algebraic) model of the domain specific framework is available, this can be used to automate the frame work scavenging process, as in the case of GENOA/GENII.

2.3 Debugging High Level Specifications

A specification for an application is a program like any other, and is consequently subject to various types of error. Some of these could perhaps be identified and corrected by inspection. However, particularly when specifications get large, a *debugger* of some sort would be needed. However, application generators typically don't come with any debugging support.

The current approach to debugging specifications written in domain-specific languages is quite unpleasant: the code generated by the AG (often in a language such as C) forms the basis of the debugging efforts. A conventional symbolic debugger such as SDB or GDB is used to debug the generated application; the user has to manually perform the mapping between the generated C source code displayed by the symbolic debugger and the original specification. This certainly involves mapping from a particular line in the generated source code to a particular line (or part there of) in the original specification. In addition, the data models for the specification language and C are probably quite different; the specification language may provide data abstractions and operations that are suitable for the application domain, but are quite different the data types of C. The user debugging at the C level has to learn how the AG implements these data abstractions in C. To complicate matters, AG may also "mangle" the identifiers used in the specification language in various ways to avoid name conflicts and/or encode different types of information; this is an

---

[1] A beta version is available free of charge for academic/non profit use. Contact the author for more information.

additional burden. The drudgery of this type of debugging represents an important challenge for AG technology.

One solution to this problem is to implement a customized domain-specific debugger, from scratch, specifically designed for the particular specification language; such a debugger would be congnizant of the syntax and semantics of the specification language. The user would be able to debug a specification with the application in terms of its particular execution and data model. The cost of building just a debugger, however, makes this approach suitable only for AG's that have wide market penetration, and are used to build very large and complex applications.

I am interested in developing systematic methods of *adapting existing symbolic debuggers* such as GDB to create domain-specific debuggers. Tools such as GDB have embedded in them many of the basic functionality of debugging: user interface, reading symbol tables from object files and executables, operating in a "master/slave" configuration with another process, reading/modifying/-displaying data, setting up various types of breakpoints etc. The research challenge here is to develop tools and techniques for the systematic adaptation of existing symbolic debuggers to create debuggers for particular domain specification languages.

References

[1] J. C. Cleaveland and R. Tatem, "METATOOL specification driven tool, System Overview," Tech. Rep. 193010-211, AT&T Bell Laboratories, North Andover, MA, 1990.

[2] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pasual, "CENTAUR: The System," in *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1988.

[3] T. Biggerstaff, "An Assessment and Analysis of Software Reuse," in *Advances in Computers, Vol 34*, Academic Press, 1992.

[4] A. Aho, R. Sethi, and J. Ullman, *Principles of Compiler Design.* Addison-Wesley, 1991.

[5] P. Devanbu, "GENOA/GENII - a customizable, language- and front-end- independentcode analyzer," in *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992.

3 Biography

Premkumar T. Devanbu is a member of the Software and Systems Research Laboratory at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include software development tools and methodologies, application generators, formal methods, and metrics.