Electronic Notes in Theoretical Computer Science 25 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume25.html 10 pages

Formal Methods in Practice

Wolfgang Polak

Consultant
Sunnyvale, USA
wp@pocs.com

Abstract

Technology transfer from academic research to industrial practice is hampered by social, political and economic problems more that by technical issues. This paper describes one instance of successful technology transfer based on a special-purpose language and associated translation tool tailored to the customer's needs. The key lesson to be learned from this example is that mathematical formalisms must be transparent to the user. Formalisms can be effectively employed if they are represented by tools that fit into existing work processes.

It is suggested that the model of special-purpose, domain-specific languages and their translators are an important vehicle to transition advanced technology to practice. This approach enables domain experts to solve problems using familiar terminology. It enables engineers of all disciplines to utilize computers without becoming software engineers. In doing so we not only mitigate the chronic shortage of qualified software personnel but also simplify the problem of requirements analysis and specification.

1 The Problem

The ultimate purpose of software engineering and computer science is to produce better, cheaper software. In this context *software* refers to a running system. The production of high-level source code is a possible but not necessary intermediate step. *Better* encompasses all qualitative aspects such as correctness, efficiency and so on. *Cheaper* refers to the overall cost of a software system including production, deployment, and maintenance.

Theoretical problems such as models for component composition, better theorem proving technology, formalized requirements analysis and the like, are important elements of a solution. The question is how best to make them practical.

Software engineers, desperate for automation, often create ad-hoc solutions without any formal basis. For example, the need to structure and organize complex software systems has lead to the creation and success of UML. Putting

©2000 Published by Elsevier Science B. V.

such tools on a rigorous formal basis is an important first step.

There is an impressive list of projects that use formal methods [1]. Yet most of the examples required extensive hand-holding by researchers and do not represent successful theory in wide-spread use. Examples of formal methods in common use are more modest and include grammars, supported by parser generators, and finite state machines [7].

Why does computer science not have a larger impact on software engineering practices? First, there is a big communication gap between theoreticians and practitioners. For the theoretician programs are mathematical objects that never fail if we can just get their specification right and verify the code. For the practitioner formal methods use obscure notation, deal with toys examples, and will never scale. Software engineers are faced with daunting management, version control, and similar problems and must constantly make engineering tradeoffs to meet tight deadlines and market windows – computer scientists know little of that. Computer scientists create wonderful theories, concepts and abstractions – software engineers understand little of that. Transitioning science to engineering is not just a technical problem but is mainly an educational, social, managerial problem.

Educational: Software engineers could make use of many theoretical results if they knew how to do so. But we don't speak the same language. The presentation of research results is geared toward peer review not towards technology transition.

Social: Software engineers are reluctant to take outside advice. After all they manage to build complex systems. Who likes to be told that some of his expertise can be replaced by a program?

Managerial: Processes and procedures for software construction have evolved over many years and are firmly entrenched in organizations. Any change will be perceived as risky and is likely rejected.

This paper describes an example of successful technology transfer based on an intelligent translator for a domain-specific specification language and lessons learned in the process. The formal systems used in this project are rather modest. The point here is that translators for very high-level languages provide an effective vehicle for making complex, formally-based tools accessible to the engineering community. Indeed, special-purpose languages suggest a new paradigm of software development by empowering engineers in other disciplines to describe (aka program) solutions to their computational and control problems.

2 An Example Of Technology Transfer

2.1 The Problems of Technology Transfer

For several years the author was involved in a research project at a major aerospace corporation. The project studied techniques for program synthesis, automatic code generation, very high-level languages, graphical design tools and similar topics. The goal was to simplify specification of software systems and to make code synthesis practical by working in a restricted domain.

As in most industrial research laboratories there was the pressure to show practical relevance of the work. To that end, the project developed a number of prototype tools that were considered practical and useful by academic standards (e.g.[3,2,8,4,5]).

But academic standards are not good enough to be accepted by those responsible for real products. Several attempts to transition some of the lab's technology to product divisions were met with universal rejection. There were several reasons for this rejection, most of them non-technical in nature.

- Academics tend to develop tools in the abstract, i.e., they solve an intellectually interesting problem without regard to actual applications. When scientists talk about concepts such as "completeness of decision procedures" of "expressiveness of languages," their value will not be apparent to decision makers. Technology must be sold by describing the concrete problems being solved, how much time is saved, and how quality is improved. The technology is irrelevant, it is its impact that matters.
- People in charge of software projects are extremely concerned about schedule risk. Even if a new tool promises great time savings, it will be rejected if there is even minimal risk that it might negatively impact the schedule. Large potential time savings are often not realistic due to a steep learning curve.
- Researchers tend to build tools in isolation without consideration of the environment and the work process of software production. Tools that require changes in an established software development process are difficult to sell.
- An important reason for rejection is the perceived and often real lack of maintenance and support for systems that come out of research labs.
- One frequent objection to the use of machine generated code was readability. From the academic point of view, machine generated Ada code is no different than compiler generated assembly code. But the programmer in the field will be skeptical of the new technology and will want to inspect and understand the code. As a consequence significant effort was spent on generating human readable, commented code.

2.2 A Breakthrough

In early 1995, the company was preparing a proposal for a new NASA satellite program. To justify a low project cost an experiment was proposed that would demonstrate and measure the cost reduction possible through automatic code generation.

We were given an existing satellite software system that was operational in a simulator environment. The task was to generate from specifications one key module to achieve a different functionality. The generated code was to be tested and validated in the existing simulation.

After many failed attempts to introduce our technology into the product divisions we had finally generated some visibility and interest. There were a few major problems though. None of the lab's researchers had any experience with the satellite domain; we did not even understand the new requirements. We had no domain-specific specification language and no idea what one should look like. And we were only given four weeks to complete the experiment. The task was close to impossible. A cynic might think that we were deliberately setup for failure. More likely, the problem was of our own making since we had created misconceptions and wrong expectations in our earlier attempts to "sell" our technology.

After some fight, we convinced management to allocate a full-time aerospace engineer to the project. He was our domain expert and brought the specification language. As it turned out, aerospace engineers specify and test their control laws in MatLab [1]. These MatLab specifications with some additional information became the input to our new tool.

Using extensive parsing, pretty printing, and tree manipulation tools that the project had developed over the years, we managed to build a prototype system that generated usable code – at least for one example. The experiment was successful and the data gathered was used in the proposal. Ironically, the proposal was not successful: its cost did not fit within the parameters considered reasonable by NASA and it was rejected as unrealistically cheap.

3 Successful Automatic Code Generation

Even though the satellite proposal was not successful, the experiment was and it demonstrated the utility of our approach and gave the lab some credibility. The aerospace engineer that participated in the experiment became a very strong advocate for the technology. By necessity (e.g., lack of time), we had created a solution that was simple and fit into the existing development process with minimal impact. As a result the initial crude prototype was further developed into a usable system, the Flight Code Generator (FCG), that is now actively used on several programs. The current version of the system employs

[1] MatLab and all other product and company names mentioned in this document are used for identification purposes only, and may be trademarks of their respective owners.

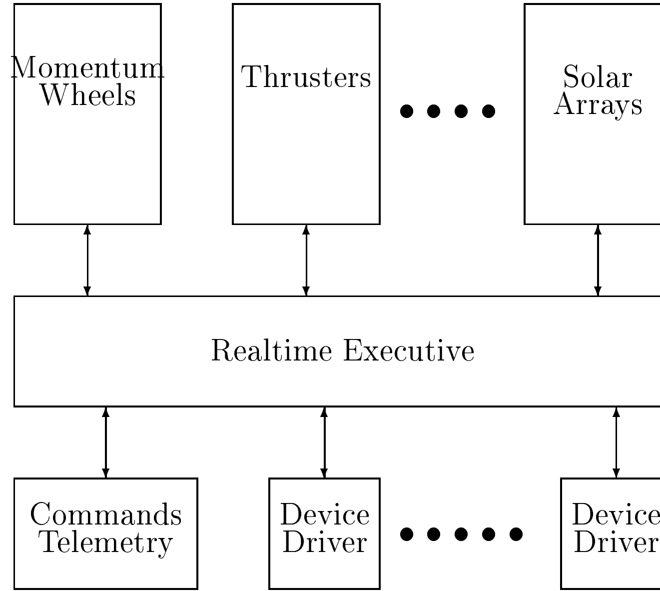

Fig. 1. Satellite software architecture with multiple functional modules that are connected to the realtime executive through standard interfaces.

dataflow analysis, various code optimization techniques, type inference, and analysis of finite state machines.

FCG is successful because it (i) is specialized to a narrow domain, (ii) generates code that fits into an existing architecture, and (iii) fits into an established development process. The following is a brief description of these technical aspects of FCG.

3.1 Building Satellite Control Systems

Figure 1 shows a reusable software architecture for satellite control systems. The realtime executive provides an infrastructure that is independent of the particular system requirements and can be reused across multiple spacecraft. It connects spacecraft specific device drivers and functional modules. These modules perform such functions as rotating solar arrays, moving momentum wheels, determine position based on various sensors and so on. The code of each module is executed sequentially at an appropriate clock rate. For each clock cycle the module performs the appropriate computation which includes reading ground commands and sending telemetry information. Modules communicate by shared variables which require no synchronization if the reader and writer modules run at the same clock rate.

During the design process aerospace engineers (AE) develop the control laws for each functional module. Typically a single engineer works on a module. The control laws are coded and evaluated in MatLab to determine proper behavior. The result of these tests are plots that show various responses to control inputs.

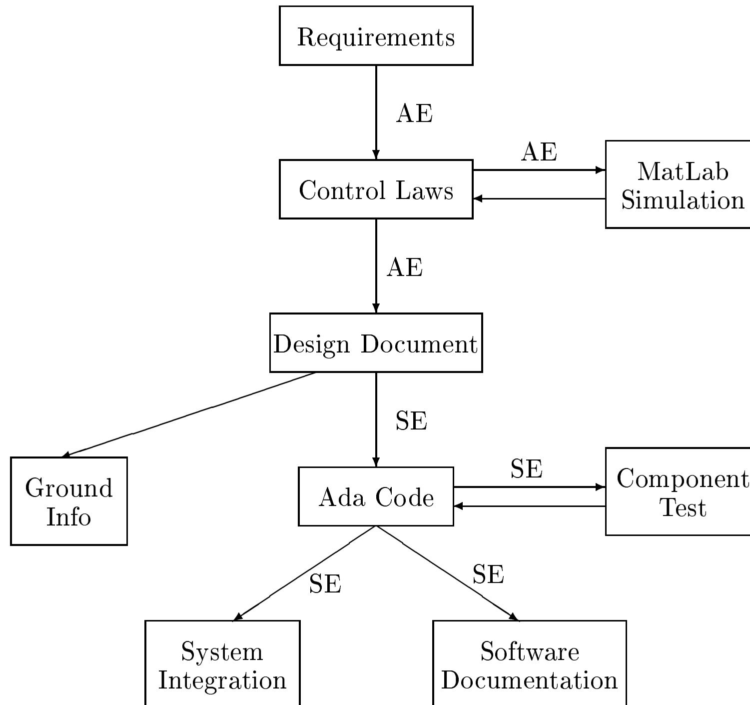

Fig. 2. Aerospace (AE) and software (SE) engineers cooperate to develop functional modules

Figure 2 show the development process for an individual module. A software engineer (SE) takes the design document produced by the aerospace engineer and develops appropriate Ada code. This code is unit-tested and later integrated into the system. Separate software documentation is produced for the hand-written Ada code. The design document is also used as a basis for developing ground software that needs to interpret telemetry information and generate commands.

3.2 Tool Support

It is apparent that the process of Figure 2 is inefficient and error prone. But it leaves plenty of room for automation and the experiment described in section 2.2 would not have possibly succeeded without the reusable architecture and the process being in place.

First, the process suggests a natural specification language: MatLab. While the MatLab source contains all necessary equations and formulas as well as test code to produce various plots, it does not contain information about the kind of telemetry to send, the commands and their parameters that are to be received, and how to respond to a particular command. Thus the specification language was defined as an extension to MatLab that includes the following additions:

- Optional type information can be added to determine precision of data and

to select specific Ada types (e.g. the support infrastructure contains a 4-element float vector type as well as a quaternion type which are structurally equal but have different associated operations).

- Telemetry is specified by listing those variables whose values are to be included in the telemetry stream.
- Commands are defined by a name and possible parameters.
- A hierarchical finite state machine (essentially a textual version of state charts [6]) specifies the actions to be taken in response to a clock tick or a command.
- Special comments were added that can be included in generated Ada code and documentation.

In addition, it was necessary to mark certain inputs (e.g., test code that generates plots) so that it can be excluded from processing by FCG. All extensions were added to MatLab using special comment characters such that a source file of the extended language can still be processed by MatLab. The resulting language is ugly by any measure. But that problem was far outweighed by the benefits of having a single representation of the design. Engineers found surprising ways to make their specifications readable.

FCG is a batch tool written in Common Lisp that takes specifications written in the extended MatLab language and generates the following outputs (controlled by command line options)

(i) Database records that describe telemetry and command information necessary for building ground software.

(ii) An Ada package that conforms to interfaces and conventions of the reusable architecture. While the code is commented and human readable it is ready for system integration and does not require human modifications.

(iii) A test environment that allows interactive or scripted unit testing of the generated Ada code. The test environment contains an interpreter that allows inspection and modification of all variables, calls to defined procedure, and the simulation of clock ticks and the arrival of commands. It also allows the generation of plots that can be compared with those generated by MatLab.

(iv) Documentation of both the design and implementation of the module. This information is based on the specifications, embedded comments, and decisions made by the Ada code generator

The new tool substantially simplifies the development process with only minimal additional work (see Figure 3). The aerospace engineer has to provide additional specifications in the MatLab source and is now performing unit tests of the generated Ada code. Any necessary code change is made in the MatLab source. Even with this additional work, the AE's job is simplified since the documentation requirements are reduced and the communication with

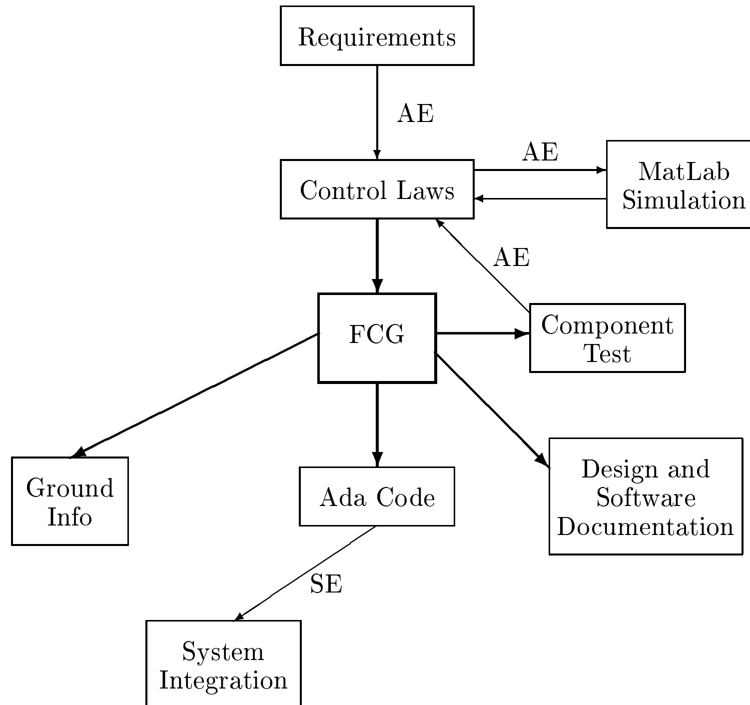

Fig. 3. FCG fits into the existing development process and eliminated virtually all manual handling of the Ada code for functional components.

the software engineer is eliminated. The SEs are left focus on infrastructure development and system integration.

3.3 A Recipe For Success

FCG is now used on three satellite systems. On one program FCG is being used both for the control and the payload software and almost half of the software is automatically generated. While this is significant, the system is not universally accepted throughout the corporation. Two problems dominate: The system lacks user support and maintenance. Many software designers refuse to work within the confines of a reusable architecture and insist on starting with a clean slate.

Why was FCG successful when much more elaborate earlier prototypes failed? Luck was an important part. The challenge experiment created the necessary visibility and convinced management and engineers of the value of the technology. Without the strong support of advocates from within the product division, insertion of new technology would not have been possible. Input from the user community is important. An internal advocate is ideal. Users that feel in control are very supportive. Interestingly, all support came from aerospace engineers whose jobs become more difficult with FCG. All resistance came from software engineers whose jobs were simplified by the tool.

Documentation is as important as code. Using a single source to generate code as well as documentation and other artifacts ensures consistency and simplifies maintenance. Being able to generate custom database records and documentation was a major selling point.

A critical reason for success is minimizing risk. In the FCG approach it is always possible to revert to the old ways if problems should arise. Several features of the system helped to minimize risk:

(i) The learning curve for the tool was very shallow. Initial use (e.g. unit testing) is possible using straight MatLab code.

(ii) The generated code is human-readable. If necessary, the code can be maintained by hand.

(iii) The tool fits into an existing development process. I.e., while some of the steps of the existing process are automated, none of the manual steps need to change in a significant way.

(iv) The system adapts to an existing architecture and its interfaces. No software changes are needed to accommodate machine generated code.

3.4 Commercial Tools

There are several commercial systems that generate code. But business reasons dictate that these systems are rather general purpose. Developing systems that generate custom code for a narrow domain is not commercially viable unless we can greatly simplify the construction and configuration of such system.

Integrated Systems offers MatrixX, a system for graphically specifying control systems and for generating code from such specifications. The product is much more mature and feature-rich than FCG but suffers from the lack of customization of the target code. The generated code cannot easily be integrated into a given satellite architecture. MatrixX was actively considered but was perceived as much higher risk and more disruptive than FCG.

National Instruments' LabVIEW and BridgeVIEW are products for graphically designing data acquisition and signal processing applications.

Other examples of successful automatic code generators include parser generators and attribute grammar systems as well as numerous generators for graphic user interfaces.

4 Final Thoughts

Formal methods are a means, not an end. To become useful and accepted, computer science theory must be packaged and become invisible. Tool builders need to understand both the formalism and their end-users. Domain-specific tools provide a promising vehicle to deliver theory to practitioners.

Ever higher levels of specification provide increased opportunities for for-

mal methods. Specifications based on constraints can use theorem provers to generate suitable code. Most domains tend to have design rules that can be checked using deductive or model-checking techniques. Domain-specific languages appear to be an effective delivery vehicle for formal methods. This, in turn, should reduce the cost and improve the quality of software.

While the FCG experience provides only one data point, the existence of commercial tools (e.g. those cited above) is evidence that suggests that automatic code generation is accepted by practitioners. Domain engineers like to be in control rather than having to depend on software engineers.

Today software engineers are expected to play experts in all areas from human-computer interfaces to fluid dynamics to fly-by-wire systems. Software engineers cannot play all these roles and if they do, poor software is a necessity. Instead, software engineers should be tool builders. They are uniquely qualified to make computers accessible to other disciplines and to empower engineers in other fields to express their designs.

Maybe domain-specific tools will eventually lead to a new software development paradigm, one where software technology empowers everyone to become a programmer in her field.

We have already seen how spreadsheet programs have made almost every computer user into a programmer. Obviously, not everyone is successful in programming their spreadsheets. But for disciplines where spreadsheets are in common use, their programming has already become part of the standard curriculum. In the long term, engineers in many disciplines will become programmers: domain specific programming will become part of the curriculum and standard practice in their discipline. Given the increasing proliferation of software, this development seems inevitable.

There is a good chance that such a development will also alleviate some of the problems of requirements analysis and capture. Requirements are often the interface between practitioners is different disciplines that speak different languages use different defaults and different common assumptions. If the requirements analyst and the programmer are experts in the same discipline there is much less change of miscommunication.

Acknowledgement

Eleanor Rieffel and James Baker provided valuable comments on earlier drafts of this paper. Discussions at the Monterey Workshop were very helpful and affected my thoughts on technology transition.

References

[1] Edmund M. Clarke, Jeanette M. Wing, and et. al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4es):626–643,

December 1996.

[2] Henson Graves. Interactive design in LEAP. In *Proc. 91 AAAI workshop on Automating Software Design*, 1991.

[3] Henson Graves. Lockheed environment for automatic programming. *IEEE Expert*, 7(6):15–25, December 1992.

[4] Henson Graves, Joe Louie, and Tracy Mullen. A code synthesis experiment. In *7th Knowledge-Based Software Engineering Conference (KBSE-92)*. IEEE Computer Society Press, September 1992.

[5] Henson Graves and Wolfgang Polak. Common intermediate design language. In *Hawaii International Conference on System Sciences*, January 1992.

[6] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[7] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE; a working environment for the development of complex reactive systems. In *Proceedings of the 10th International Conference on Software Engineering*, pages 396–406, Singapore, April 1988. IEEE Computer Society Press.

[8] J. Williamson, P. Jensen, L. Ogata, and H Graves. Automatic programming technologies for avionics software (APTAS). In *Proceedings of the 9th Digital Avionics Systems Conference*, pages 101–106. IEEE, 1990.