Design Automation: Making Formal Methods Relevant

Lisa Walton and James Hook*
walton@cse.ogi.edu and hook@cse.ogi.edu†

Formal methods are easily dismissed as heavy weight and irrelevant. People legitimately ask to see what impact formal methods have had on software development practices outside of those areas where governments or regulatory agencies have mandated their use. We argue that design automation is an opportunity for the insertion of formal methods into software development practice. In particular, we advocate the development of small, domain-specific design languages, and the use of these languages as front-ends to program generation systems.

Software Design for Reliability and Reuse (SDRR) is a method developed at the Pacific Software Research Center to support the development of generators for domain-specific languages [1]. Our method, which utilizes a robust suite of reusable transformation tools, has been used to implement a software component generator for a message translation and validation problem domain identified by the Air Force. Independent contractors have used our system as part of an experiment comparing our generation-based technology to an existing solution that uses a program templates based reuse technology[16, 18]. Preliminary results show a productivity improvement factor of 2.9 over templates, which already represents a significant productivity improvement over ad hoc methods[12].

1 Formal Methods Successes

The formal methods community is currently in a phase of critical self-evaluation. At the recent 1994 Monterey workshop on formal methods, the emerging consensus was that there needed to be a new, more realistic vision for future research. Participants universally concurred that formal methods are critical to the emergence of software engineering as a well-organized discipline based, as are other engineering disciplines, on sound and well-tested mathematical models. To this end, future research must be focused on approaches that would have a direct positive impact on large-scale software development. Crucial to this effort are the following goals

- Incorporate formal models and algorithms in computer tools designed to solve problems of practical significance
- Recognize that practical software development is largely driven by user requirements changes

*Authors supported in part by Air Force Materiel Command.

†Pacific Software Research Center, Oregon Graduate Institute of Science & Technology, P.O. Box 91000, Portland, OR 97291-1000 USA, http://www.cse.ogi.edu/PacSoft.

- Recognize that human understanding and creativity play an important role in software development

As well as a call for change, there also emerged some important success stories that were greeted enthusiastically by the participants. At the conclusion of the workshop, an examination of the factors contributing to these successes suggested an emerging paradigm for applying formal methods. The following list, substantially due to Goguen, describes these factors.

Focused, well-defined, and well-understood problem domain Successful systems generally solve problems in tightly-constrained domains which had significant leverage from context and from prior work. A particular advantage is gained when a solution can build on or incorporate previously defined libraries of program modules in the domain.

Coherent User Community A significant amount of early and continuing user interest and participation contributes to the design and evolution of successful tools. Participation includes but is not limited to providing financial resources, contributing significant domain knowledge, and debating design issues within the user community.

Intuitive Interface Most systems described as successful have an intuitive user interface that is meaningful to experts in a domain. These interfaces often incorporate the domain experts' own conventions and notations.

Large-Grain Approach Successful systems deal with large grain system issues, not small grain coding issues. Domain experts use formal methods to generate or define significant components or whole programs rather than having software engineers verify code at the statement or expression level.

Reasonable Integration into User Community Formal methods concepts and algorithms encapsulated into powerful tools must still be integrated into user development process. Success in this area is generally achieved when users are not required to significantly change the way they work by learning radically new notations and techniques.

It should also be noted that successful, well-crafted systems are generally built on top of previously developed, flexible tools. Cited in the proceedings of the workshop are the following systems that incorporated the above factors: CAPS [17], ControlH and MetaH [22], AMPHION [25], Panel [20], and Software Design for Reliability and Reuse (SDRR) [13]. We will look at two of these systems, AMPHION and ControlH and MetaH, as well as our own SDRR method.

1.1 Amphion System

Waldinger and Lowry's Amphion system, developed at NASA Ames, is a creative system that generates Fortran programs to solve planetary geometry problems. By annotating a picture, a user specifies a problem statement such as "Where do I

point my camera on the Galileo spacecraft to take a picture of the comet impact on Jupiter?". Provided the problem is adequately specified, the system uses a deduction-synthesis technique to calculate a program that solves the problem. The generated program is essentially a series of calls to an extensive library of Fortran subroutines that NASA had previously developed to support planetary geometry. Prior to Amphion, this well-engineered library was not being used by the NASA scientists it was intended to support.

1.2 ControlH and MetaH

Vestal presented Honeywell's experience with tools formally capturing related designs in the domain of Guidance Navigation and Control. In this domain there is a well-established notation familiar to all of the control engineers which is not well-understood by software engineers instantiating the designs. They overcame this problem by developing tools that allowed a design to be represented in multiple, related specification languages. In particular, they ensure that the specification used by the software engineers is formally related to the diagrams drawn by the control engineers.

1.3 Software Design for Reliability and Reuse

PacSoft's SDRR method is based on design capture in domain-specific design languages (DSDLs) and automatic program generation using a reusable suite of program transformation tools. By capturing design requirements at the appropriate levels, the encapsulated designs (rather than the software components) become the reusable artifacts. When a subsequent application or version of a design is needed, design modifications are made to the specification given as input to the generator, and a new software component is generated automatically.

2 Formal Methods in SDRR

The design of an SDRR component generator begins with a multi-level domain analysis to determine the requirements of the users, characteristics of a component solution, and characteristics of the execution environment. It is here that SDRR makes a major contribution in the application of formal methods to real problems.

2.1 Design Requirements

The front-end to a software component generator is an effective DSDL tailored to the needs of the users. The back-end of a software component generator is parameterized with an implementation template (see below) tailored to the requirements of the existing system. A three-level domain analysis allows us to formally incorporate design requirements into these two interfaces as follows.

1. Design requirements for domain objects that need to be described by domain experts are captured by a direct semantics which structures and formalizes the collection of typically ad-hoc user notations. It is important that the semantics be expressive over domain entities without incorporating artificial encodings or over-specifying problem solutions. These operational details are more appropriately captured elsewhere. For the MTV generator, domain objects are message formats and constraints, and the ad-hoc notations are found in informal textual specification documents.
2. Design requirements of a software solution are captured in a formal computational semantics given to the DSDL in ADL, the algebraic design language used in SDRR [11, 13]. It is here that formal algorithms to solve domain problems should be captured. Note that these computational details will be largely invisible to the users of the DSDL. For the MTV generator, software solutions are modules that translate and validate incoming messages.
3. Design requirements of the environment in which the component will be used are captured in an implementation template [23, 24] characterizing the execution environment. These are a set of implementation primitives that specify how the computational semantics of the DSDL are to be realized in terms of a target programming language (Ada, for the MTV modules).

2.2 Automated Transformation Tools

Our suite of reusable tools optimize generated components through the use of automated program transformations that are applied during the course of program generation [14]. The transformations are mathematically based and are guaranteed to preserve the computational meaning of the programs specified in the design language. The transformation tools include: HOT, which applies higher-order transformations [21]; PEP, which performs lambda-lifting and higher-order removal [10, 8]; Firstify, an implementation of Reynold's algorithm for defunctionalization [19, 3, 2]; and Astre, a first-order transformation tool based on term-rewriting techniques [6, 4, 5].

When an SDRR program generator is applied to a DSDL specification, it automatically applies the necessary transformations. The pipeline of transformation tools constitutes a very advanced optimizing compiler that takes ADL as input and generates conventional, imperative target language code as output. In the prototype, the target language is Ada.

2.3 Formal Methods made Effective

In the experiment with the Message Translation and Validation (MTV) system, our status with respect to the success indicators is as follows:

Focused, well-defined and well-understood problem domain The requirements for Message Translation and Validation (MTV) modules common in

command and control systems (C^3I) are well understood and reasonably well documented.

Coherent User Community Input was solicited early on from current system users as to what would be useful in a tool for writing formal specifications, including both a description of their "blackboard notations" and elements of commonly occurring specifications that could be captured efficiently.

Intuitive Interface We developed a DSDL encapsulating some of the notations and structure of the informal specification documents used by engineers to describe message formats and data constraints.

Large Grain Approach The problem requires software components to be generated from formal specifications and integrated into an existing system. Formal methods are used by the design experts to define and generate components, not to verify code.

Reasonable integration into community The intent is for formal specifications written in the DSDL to replace users' informal textual documents. Statements in the formal language correspond closely to statements in the informal document, minimizing the amount of new notation to be learned and facilitating the transition from one design style to the other.

2.4 Formal Methods Made Relevant

A recent survey of industrial applications of formal methods [9] indicated that the following issues represent some of the biggest barriers to getting formal methods into the large-scale software development process.

- Most programming and specification languages lack the semantic base required to support the full application of formal methods.
- Aspects of run-time environments and performance aspects receive inadequate treatment in formal method applications
- Inadequate cost models exist for measuring the current or projected gains experienced in the development process when formal methods are used over other standard development methodologies

SDRR is a method that provides solutions to the first two issues, and our proof-of-concept experiment has provided us with metrics and cost data that will lead to the development of preliminary cost models.[12]

- Our mathematically-based transformation tools and the algebraic design language (ADL) give us the full support required to incorporate formal methods into software systems.
- One level of our domain analysis provides us with the requirements information that makes it possible to capture important run-time environment information about the system which is incorporated into the parameterized templates mechanism.

- Although we do not as yet have a cost model, we do have preliminary data characterizing the scope of our effort[15]. We have measured cost, schedule, size and effort of both the domain specific and the reusable parts of our technology and have conducted an experiment to demonstrate the impact of the technology on productivity, reliability, flexibility, predictability and usability[12].

We claim that SDRR provides a way to introduce formal methods into an existing system without incurring a high cost from retrofitting the new technology. This is due to the flexibility of our system, which can be linked into systems that are not formally defined or modeled. Specifically, formalizing the specification of the design requirements in our method does not require that a formal definition exist for the "blackboard notations", the target language, the target environment, or the existing system into which the components are to be incorporated.

Developers of software components generators can be provided with a reusable set of tools and a method for their use. These developers need not be experts in either mathematics or the formal theory behind the transformation methods, because the tools and library code are highly modular and can operate as a "black box". After the initial development effort, users will be left with a system that allows them to go from a specification to a software component without having to understand the details of how the generator is constructed.

To test out claims of the benefits of the SDRR method we conducted an experiment in which a set of four subjects working for an independent organization did a series of tasks specified by the Air Force in our specification-based generator technology and an existing "good" reuse technology based on program templates [18]. The experiment demonstrated a statistically significant productivity improvement over templates. The observed ratio of average effort hours to complete a task was 2.92. The confidence that the difference in the means was significant, based on a single-factor analysis of variance, was 99.5%. The experiment also demonstrated that the subjects were more likely to correctly specify the problem with the formal method than with the templates-based method [12].

3 Conclusion

SDRR makes a major contribution in the application of formal methods to real problems. We believe that we can have a direct impact on the large scale software development process by making transformation-based generation technology automatic and effective, and by facilitating technology transfer and design reuse. In terms of the aims espoused at the 1994 Monterey Workshop, our status is as follows

- We have incorporated formal models and algorithms in our transformation tools, and our method describes a plan to solve a class of significant problems.
- Our three-level domain analysis captures design requirements at the appropriate levels, so that software development can occur rapidly in response to changes in user requirements.

- By involving the user community in the design process, and by preserving their notations in a specification language that is intuitive to them, we preserve their role in the development process, and do not stifle their creativity by making them substantially change the way they do business.

What we have accomplished in the SDRR proof-of-concept demonstration project:

- Incorporated formal methods into the development of appropriate domain specific design languages and implementing flexible and maintainable generators supporting them.
- Built and rigorously tested a reusable tool suite incorporating formal mathematical algorithms to support this method.
- Built a generator for a real-world problem constructed by applying the SDRR method.
- Performed an experiment comparing the resulting generator to the current state-of-the-art illustrating that we can make formal methods effective [12].
- Compiled a record of the process and metrics data characterizing our experience with the method and its development, thus planning to show that using formal methods can be relevant [15].

Together, this combination of research, demonstration, and experimentation exemplify a new paradigm for the rapid transfer of technology from an academic research institution into industrial and government software development practice.

References

[1] Jeffrey Bell et al. Software design for reliability and reuse: A proof-of-concept demonstration. In *TRI-Ada '94 Proceedings*, pages 396–404. ACM, November 1994.

[2] Jeffrey M. Bell. An implementation of Reynold's defunctionalization method for a modern functional language. Master's thesis, Oregon Graduate Institute CSE, January 1994.

[3] Jeffrey M. Bell and James Hook. Defunctionalization of typed programs. Technical Report 94-025, Oregon Graduate Institute CSE, February 1994.

[4] Françoise Bellegarde and James Hook. Monads, indexes, and transformations. In *TAPSOFT '93: Theory and Practice of Software Development*, volume 668 of *LNCS*, pages 314–327. Springer-Verlag, 1993.

[5] Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2–3):287–311, 1994.

[6] Francoise Bellegarde. Program transformation and rewriting. Technical Report CSE-90-021, Oregon Graduate Institute CSE, September 1990.

[7] Pacific Software Research Center. SDRR project Phase I final scientific and technical report, February 1995.

[8] Wei-Ngan Chin and John Darlington. Higher-order removal: A modular approach. Unpublished work, 1993.

[9] Dan Craigen, Susan Gerhart, and Ted Ralston. An International Survey of Industrial Applications of Formal Methods. Volume 1: Purpose, Approach, Analysis, and Conclusions. Technical report, US Department of Commerce, NIST, 1993. Technical Report NISTGCR 93/626.

[10] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 190–203. Springer-Verlag, 1985.

[11] Richard Kieburtz and Jeffrey R. Lewis. Algebraic design language. Technical Report 94-002, Oregon Graduate Institute CSE, 1994.

[12] Richard B. Kieburtz. Results of the sdrr validation experiment, February 1995. In [7].

[13] Richard B. Kieburtz. Software design for reliability and reuse—Method definition, February 1995. In [7].

[14] Richard B. Kieburtz, Françoise Bellegarde, Jef Bell James Hook, Jeffrey Lewis, Dina Oliva, Tim Sheard Lisa Walton, and Tong Zhou. Calculating software generators from solution specifications. Technical Report OGI-CSE-94-032B, Oregon Graduate Institute CSE, October 1994.

[15] Alexei Kotov. Measurement final report, February 1995. In [7].

[16] Jeffrey R. Lewis. A specification for an MTV generator. Technical Report 94-003, Oregon Graduate Institute CSE, 1994.

[17] Luqi, Joseph Goguen, and Valdis Berzins. Formal support for software evolution. In *Proceedings of the 1994 Monterey Workshop*. U.S. Naval Postgraduate School, September 1994.

[18] Charles Plinta, Kenneth Lee, and Michael Rissman. A model solution for C^3I message translation and validation. Technical report, Software Engineering Institute, Carnegie Mellon University, December 1989. CMU/SEI-89-TR-12 ESD-89-TR-20.

[19] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.

[20] Jacob Schwartz and Kirk Snyder. New York University Design of Applications for Multimedia Applications Development. In *Proceedings of the 1994 Monterey Workshop*. U.S. Naval Postgraduate School, September 1994.

[21] Tim Sheard and Leonidas Fegaras. Optimizing algebraic programs. Technical Report 94-004, Oregon Graduate Institute CSE, 1994.

[22] Steve Vestal. Honeywell technology center formal methods for complex evolving systems. In *Proceedings of the 1994 Monterey Workshop*. U.S. Naval Postgraduate School, September 1994.

[23] Dennis Volpano and Richard B. Kieburtz. Software templates. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 55–60. IEEE Computer Society, Aug 1985.

[24] Dennis Volpano and Richard B. Kieburtz. The templates approach to software reuse. In Ted J. Biggersstaff and Alan J. Perlis, editors, *Software Reusability*, pages 247–255. ACM Press, 1989.

[25] Richard Waldinger and Michael Lowry. NASA ames research center AMPHION: Towards kinder, gentler formal methods. In *Proceedings of the 1994 Monterey Workshop*. U.S. Naval Postgraduate School, September 1994.