ELSEVIER

Information and Software Technology 38 (1996) 313–322

INFORMATION AND SOFTWARE TECHNOLOGY

Challenges in the successful transfer of formal methods technology into industrial applications

Hossein Saiedian[a], Michael G. Hinchey[b]

[a]*Department of Computer Science, University of Nebraska, Omaha, NE 68182, USA*
[b]*Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ 07102, USA*

Received 10 February 1995; revised 5 May 1995

Abstract

The primary objective of this article is to discuss a number of challenges (presented in terms of 'guidelines') that must be addressed in a pragmatic manner in order to transfer formal methods technology into the actual workplace and to ensure that formal methods are actually used on an industrial scale.

Keywords: Formal methods; Marketing; Simulation; Training; Education; Reusable framework

1. Introduction

Computer-based systems are built in areas such as air traffic control, on-line hospital patient record management, stock transaction control systems, and so forth. These systems play important roles in our daily lives. As such systems turn from being merely 'information systems', and are used more and more in increasingly sensitive and sometimes life-critical environments, operating over distributed platforms and with strict requirements on response times, inadvertent errors in their specification, design, and implementation could have major impact on our safety and well-being. Some of these systems will consist of millions of lines of code and their complexity may eventually supersede the complexity of any artefact ever built by mankind.

It is important that the developers of such systems employ those methods that can offer a high assurance that a system will operate as desired. Thus, it is essential that such developers seek assurance that the system requirements accurately capture the users' critical requirements, that the system design correctly reflects the system requirements, and that an implementation in software (or hardware) is an accurate realization of the system design. By integrating *formal methods* into the development process of a system, such added assurance can be achieved to a high degree.

Formal methods provide a notation for a *formal specification* of a system. A formal specification is needed for effectively describing the desired behaviour of a proposed system at an abstract level that can be reasoned about, either formally in mathematics, or informally but rigorously. A specification is formal if it has a precise and unambiguous semantics. Precision and unambiguity are important because during the development of a large system, many individuals have to agree on the interpretation of the specification in order to produce a correct implementation. A precise and unambiguous semantics is normally given in the form of an abstract mathematical model with a set of definitions.

Despite the clear and uncontentious advantages of formal methods over non-formal traditional (structured) development methods, the developers of computer-based systems are often unmotivated to consider using them. Furthermore, since some very basic definitions and applications of formal methods are confused with certain 'myths', the uninitiated practitioner may even find it easier to disregard these methods altogether than investigating their potentials. In fact, practitioners often perceive formal methods as an academic exercise, difficult to use, and insufficiently supported by automated tools. However, these and many other similar 'myths' of formal methods are unfounded and many of them have been addressed in detail [1,2].

The primary objective of this article is to discuss a number of challenges that must be addressed in a pragmatic manner in order to secure the transfer of formal methods technology into the workplace and to ensure that formal methods are actually used on an industrial scale. These challenges, expressed in terms of guidelines, include initiation of joint industrial-strength case studies by both the academia and industry, clarifying when formal methods should be used in the software process, the importance of developing automated tools, introducing a framework for reusing formal specifications, developing executable specification languages for simulation and rapid prototyping, running industrial courses to train software practitioners, and educating the future generation of software engineers. Each of these guidelines is discussed in more detail below.

SSDI 0950-5849(95)01057-2

2. Guideline 1: Initiating further industrial-strength case studies

Although widely cited as techniques that can result in systems of the highest integrity, and to an extent mandated in certain classes of applications [3], formal methods remain one of the most controversial areas of modern software engineering practice.

They are the subject of extreme hyperbole by self-styled 'experts' who fail to understand what formal methods actually are, and of deep criticism by proponents of other techniques who see formal methods as merely an opportunity for academics to exercise their intellects over whichever notation is the current 'flavour-of-the-month'.

After a quarter of a century's application, one would have hoped that the half-truths and unwarranted prejudices that abound in the software engineering industry would have died out. Unfortunately this is not the case, and many practising software engineers seem happy to take to heart both criticisms and extreme exaggerations regarding what formal methods can and cannot do [4].

Most practitioners perceive formal methods as academic tools which are difficult to use. They are reluctant to use them despite their considerable advantage over traditional methods. It is difficult to say what really needs to be done to convince industrialists to take formal methods to their hearts; this is a research topic in itself. However, a certain degree of appropriate 'marketing', highlighting the applicability of formal methods in certain classes of system and in particular environments, *should* help to save formal methods from this unfair perception, and to widen their range of acceptability within the system development community at large.

To demonstrate that formal methods pay off, more realistic large-scale examples performed in conjunction with industry (e.g. IBM's CICS) are necessary. These industrial case studies are not only necessary to advance the technology and demonstrate the potential benefits, they also help identify the needs of companies that adopt formal methods and serve to enhance the integration of formal methods with current software engineering practices. In fact, the proponents of formal methods should continuously look for further cooperative opportunities with industry to plan for newer industrial-strength case studies to avoid repeatedly reciting old case studies (such as CICS, Darlington project, etc.). To convince the 'uninitiated' practitioners, large case studies are quite essential. Of course, if formal methods are to be 'marketed', then the results of these applications (whether successful or otherwise) need to be disseminated to potential users of formal methods.

A survey of industrial usage on both sides of the Atlantic has become widely quoted, and should play a major role in highlighting the benefits of formal methods [5]. Hinchey and Bowen [6] attempt to consolidate much of this information in an industrially useful manner. It includes a summary of the findings of the aforementioned survey, and contains contributions relating to a number of the systems addressed in that survey, the contributions being written by the actual developers.

The case studies relate to a wide range of formal methods (B, CCS, VDM, Z, etc.) and to a wide range of applications (such as high-integrity systems in the avionics, nuclear projects, atomic energy control, railway industries, security-critical systems, etc.). The use of Z and B in CICS, often cited as a prime example of the successful industrialization of formal methods, is also discussed as well as more unusual applications such as a voting system.

Large case studies and industrial-usage reports can also assist in identifying the limits of formal methods. Formal methods have proven very useful for the specification of functional properties of a system. Non-functional properties of a software system such as reliability, cost constraints, performance, portability, man-machine interfaces, and resource consumption of executing programs are difficult, or perhaps even impossible to specify by means of formal methods. Some research has been carried out into formalizing resource consumption and resource allocation as well as in fault-tolerance/reliability and user-interface design [7]. However, it is only with practical applications that the limits or constraints of formal methods will be revealed.

In general, formal methods tend to address semantic issues rather than pragmatic issues of software development. Management and practitioners are, however, more concerned with pragmatic issues.

Weber-Wulff [8] addresses a number of ways in which management is more likely to be convinced of the appropriateness of formal methods for their own environments. The more relevant of these propositions, from our point-of-view are:

(1) *An industrial formal method should be confinable.* That is, a formal method should be applicable to particular aspects of system development and should not have adverse effects on other areas. Most of the more successful applications of formal methods have focused on a relatively small proportion of the system, although that is not to say that such applications have been trivial. The key is to focus on critical portions, where the cost of failure would be excessive. To suggest that formal methods should be applied to all aspects of the development is ludicrous [4], and it is likely that the benefits of using formal methods would be lost if one were to attempt to introduce such a policy.

(2) *The use of formal methods should be reversible.* Management may be loath to commit to formal development if they believe that such a step is cast in stone. If it is found that the cost of formal methods is just not justified for the system in question, or that the expertise required is just not available within the organization (and that consultancy is not feasible) then

it is important that the organization can revert to its previous development processes.

A practical means of ensuring this at the outset might involve method integration. Success has been reported in the use of structured and formal methods in parallel [9]. There is a certain point, however, beyond which this becomes unnecessary and costly. Much interaction between the parallel development is required if the benefits of using formal methods are to be highlighted, and if the structured method's support for the software process is to be exploited.

(3) *A formal method should be open to allow interchangeability of software components or to allow use of different tools.*
In many respects this is related to the previous propositions. If a formal method cannot interact with other development methods and confines the developers to using only software components developed with that method, then it will be very difficult to convince management to adopt that method. The re-use of legacy code has been highlighted as a means of increasing productivity, increasing confidence in system integrity by reusing components that have been formally proven to be correct, and reducing lead-time to market. Prohibiting such reuse, or parallel development (using other development processes) would be a major inhibition to the integration of formal methods into industrial development.

(4) *There must be adequate documentation for an industrial formal method.*
The industrialization of a formal method requires adequate support in terms of documentation, teaching materials and tool support. The industrial take-up of a formal method takes at least a decade [10], and is reliant on the development of appropriate tools, the publication of textbooks and other materials, and the teaching of the method in academic environments so that graduates may help in the technology transfer to industry. Just as a number of excellent programming languages have died sudden-deaths due to lack of documentary support, the same can be said of a number of formal methods. Hewlett-Packard's proprietary specification language, HP-SL, despite having many useful features, completely failed the technology transfer test; the executable specification language Paisley seems to have suffered a similar fate. One cannot but wonder whether the COLD family of languages is endangered also. Despite best efforts at publication in book format, the languages seem to be used almost exclusively at Philips.

Lastly, it may now be imperative to name a few important organizations that suggest and/or mandate the use of formal methods for (safety-critical) software development: the International Electrotechnical Commission, the European Space Agency, the UK Ministry of Defence, Canada's Atomic Energy Control Board and Ontario Hydro [4].

3. Guideline 2: Clarifying *when* to use formal methods

An important responsibility of the proponents of formal methods is to clarify when in the development process formal methods should be applied. As full formal development is rarely employed, thus far the greatest benefits of formal methods have been demonstrated at the early stages of development for the purposes of modelling and specification.

What normally discourages practitioners is the mathematics involved in proving programs correct (program verification). Program verification, however, is carried out at the later phases of development when actual programs have been coded. Program coding is not necessarily the most error-prone part of the development, especially if the overall structure of the system under development has been properly designed and well-conceived. The need for complicated programs and, by extension, program verification, is in fact a sign of poor design.

The greatest benefit of formal methods emerges when they are employed during the specification and modelling stages, early in the development process. During these stages, formal methods can be applied profitably to develop clear and concise specifications. The simple act of precise specification and modelling often provides the greatest benefit, although reasoning about specifications can also provide considerable additional advantages. Consider, for example, one of the better known real-life applications of formal methods—the application of SCR (Software Cost Reduction) techniques to the Darlington Nuclear Facility in Canada.

That application is often cited as a major argument in favour of formal methods. Indeed, the work involved highlighted several errors in the existing code that would have gone undetected by testing. Specifications and models derived from the implementation were amenable to formal examination and analysis. The application is also often cited, however, as a means of highlighting the extreme cost of formal methods. It is true that the project cost several million dollars, although the software consisted of just 2000 lines of code.

Nevertheless, the project stands as one of the great success stories of formal methods, and it is our contention (and that of David Parnas, who acted as a consultant in establishing the Ontario Hydro standards [11]) that the costs would have been significantly less had formal methods been used in the initial stages of development (rather than used to 'backfit' the existing code).

Although Darlington was expensive, and many might dispute such a high investment, we believe that (in this case) the investment was warranted, due to the catastrophic destruction and loss of human life that would have ensued as a result of a system failure. However, that is not to say that formal methods are justified in all system development, and one must clarify *when* formal methods should be applied.

Applying formal methods blindly to *all* aspects of a system will certainly not be cost-effective. Most successful applications of formal methods have concentrated on *critical components*, and indeed the use of formal methods is justified in all high-integrity systems, or components of systems which are required to meet the highest integrity standards [3], that is, where 'correctness' is of the essence.

4. Guideline 3: Investing in automated tools

One factor limiting the use of formal methods is the lack of investment in automated tools and support structures to reduce the efforts of applying these methods. In fact, lack of support tools is often seen as a major barrier to using formal methods.

A key factor in the acceptance of high-level languages has been the presence of a comprehensive set of tools to support the user. If formal languages are to achieve the same level of acceptance, they too require extensive automated support. Support tools may reduce the learning time, thereby aiding their widespread use. Automated tools may include:

- special editing environment;
- syntax checkers;
- animation tool;
- refinement and proof tools.

A special editing environment for Z would, for example, provide a specifier with a number of pop-up menus from which the specifier could view global schemas, local schemas, state schemas, operation schemas, defined sets, etc. The editor would also make schema creation, modification, deletion, etc. more flexible.

In addition to the above, good interfaces to specification languages, transformation tools for taking popular methods and converting them into formal methods, and tools for inferencing from specifications to assist software validation are needed. Craigen [12] classifies the potential advantages arising from the use of formal methods tools as follows:

(1) *Soundness.* A formal methods tool can guarantee that sound reasoning is performed because it would have the capability in taking care of the minutiae involved in formal reasoning in comparison to a system developer.
(2) *Tracking.* A formal methods tool can accurately maintain a database of dependencies, incomplete definitions, proven and unproven properties, etc.
(3) *Magnification.* Formal methods tools have the potential for increasing our capabilities in developing formally specified and verified systems.

Furthermore, a specification is and should be considered a major reference document for the customer as well as the developer. It is impractical, however, to expect a customer to read mathematical expressions. A large amount of work needs to be done in this area for developing tools, for example, to animate the mathematical expressions in a specification document so that a customer may understand them more easily. It should be noted that a number of very useful tools have already been developed for Z, including ZTC, *f*UZZ, and CADiZ. Similarly, automated toolboxes for VDM-SL support formal development as well as type and semantic checkers as well as a pretty-printer that produces LATEX code. Certain formal models incorporate tool support directly. For example, OBJ includes an executable subset while Larch supports a theorem prover. More tools have recently been reported [4].

5. Guideline 4: Constructing a reusable framework for formal specifications

Traditionally, formal methods have been used for functional specification of software (and hardware) systems, focusing largely on abstraction techniques (and refining abstractions into some implementation).

To ensure that formal methods become an integral part of industrial software engineering, the use of these methods has to be made as cost-effective as possible. One way of achieving such cost-effectiveness is through the development of a framework for the reuse of existing formal specifications. In Europe, the ESPRIT project 'REDO' focused on the reverse engineering and redocumentation of existing software (mainly COBOL). The project resulted in the development of a useful compendium of techniques for the retrieval of specifications from existing software. Clearly specifications could be manipulated and reasoned about far more easily than badly structured COBOL (or even well-structured COBOL), and later the modified specifications could be used to regenerate well-structured systems that were well documented.

The idea of maintaining libraries of formally specified software components that can form the basic design repertoire of software developers is not an entirely new one [10,13], and such libraries will slowly be propagated. An interesting result is that libraries of specifications are easier to maintain than libraries of program fragments. In addition, specifications are abstract descriptions with little implementation bias, and can more easily be reused in different systems and different environments [10], with the cost of development amortized over multiple products, and with an increasing uniformity across a range of products [13].

The fact that formal specifications may be reused is also likely to encourage greater rigour in their derivation and documentation, as developers consciously consider that greater effort at this stage may in future result in faster and cheaper development with less effort.

Such factors are likely to increase the acceptability of formal methods in industrial development. In fact, while potential reuse might be a justification for the adoption of formal methods, it has been argued that reuse cannot exist without formal methods, as without a formal specification a program is neither adaptable nor portable, and thus not even *potentially* reusable [14].

6. Guideline 5: Exploiting simulation and rapid prototyping

Although they have proven to be very successful when applied at early stages in the development process, the advantages of formal methods are not always so apparent at the outset. This is because the cost structure of the development changes dramatically; initial phases of development are now more costly, which conspires to convince many developers that formal methods are expensive.

Formal methods *are* expensive, but in many cases this expense can be justified [10]. However, increased costs at the outset are generally more than outweighed by the reduced costs at later stages (i.e. at the implementation stage and during post-implementation testing).

The use of formal methods in rapid prototyping and simulation is one area where the usefulness and applicability of formal methods can be demonstrated to 'non-believers'. Rapid system prototyping and simulation have much in common in the sense that both involve the derivation and execution of an incomplete and inefficient version of the system under consideration. They do, however, have different aims (although these are certainly not incompatible), and are applied at different stages in the system life-cycle.

Prototyping is applied at the earlier stages of system development as a means of *validating* system requirements. It gives the user an opportunity to become *au fait* with the 'look-and-feel' of the final system, although much of the logic will still not have been implemented. The aim is to help in determining that the developer's view of the proposed system is coincident with that of the users. It can also help to identify some inconsistencies and incompatibilities in the stated requirements. It cannot, for example, be used to determine whether the requirements of efficiency of operation and requirements of ease of maintenance are mutually satisfiable. The prototype will in general be very inefficient, and will not necessarily conform to the stated design objectives.

Best practice holds that the code for a prototype should be discarded before implementation of the system. The prototype was merely to aid in eliciting and determining requirements and for *validation* of those requirements; that is, determining that we are building the 'right' system. It may have a strong *bias* towards particular implementations, and using it in future development is likely to breach design goals, resulting in an inefficient implementation that is difficult to maintain. Retaining a prototype in future development is effectively equivalent to the transformational or evolutionary approach to system development, with a certain degree of circumvention of the specification and design phases.

Simulation fits in at a different stage of the life-cycle. It is employed after the system has been specified, to *verify* that an implementation may be derived that is consistent both with the explicitly stated requirements, and with the system specification; in other words, that we are building the system 'right'. While prototyping had the aim of highlighting inconsistencies in the requirements, simulation has the aim of highlighting requirements that are left unsatisfied, or only partly satisfied.

Both rapid prototyping and simulation suffer from one major drawback. Like testing, which can only highlight the presence of software bugs, but not their absence, prototyping and simulation can only demonstrate the existence of contradictory requirements or the failure to fully satisfy particular requirements. They cannot demonstrate that no contradictory requirements exist, nor that all specified requirements are satisfied, respectively [15]. That is why attention has begun to be focused on the use of formal methods in both rapid system prototyping and simulation, as formal methods can actually augment both of these areas with *proof* [16].

The use of executable specification languages and the animation of formal specifications are clearly two means of facilitating prototyping and simulation, while retaining the ability to prove properties.

6.1. Executable specifications

We differentiate here between executable specifications and specification animation, although many authors consider them to be identical.

We consider specifications to be 'executable' when the specification language inherently supports explicit execution of specifications. While the means by which executions of such specifications are performed are varied and interesting in themselves, they are not of concern to us here.

An executable specification language offers a distinct advantage—it augments the conceptual model of the proposed system, derived as part of the system specification phase, with a behavioural model of that same system. This permits validation and verification (as appropriate) at earlier stages in the system development than when using traditional development methods [1].

There is a fine line between executable specifications and actual implementations—that of resource management. While a good specification only deals with the functionality and performance properties of the system under consideration, implementations must meet performance goals in the execution environment through the optimal use of resources.

The use of executable specifications has been criticized for unnecessarily constraining the range of possible implementations [17]. While specifications are expressed in terms of the problem domain in a highly abstract manner, the associated implementation is usually much less 'elegant'. It has been claimed that implementors may be tempted to follow the algorithmic structure of the executable specification, although this may still be far from the ideal, producing particular results in cases where a more implicit specification would have allowed a greater range of results.

Hayes and Jones [17] also claim that while executable specifications can indeed help in early validation and verification, it is easier to prove the correctness of an implementation with respect to a highly abstract equivalent specification rather than against an implementation with different data and program structures. This is crucial; it indicates that executable specifications, while permitting prototyping and simulation, in the long run may hinder *proof of correctness*.

6.2. Animating formal specifications

While executable specifications incorporate inherent support in the specification language, animation applies to specification languages which are not normally executable.

In this category we include the animation of Z in Prolog [18] and the direct translation of VDM to SML [19], as well as the interpretation and compilation of Z as a set-oriented programming language [20].

Such specification languages were not intended to be executable, but by appropriately restating them directly in the notation of a declarative programming language, become so. In fact, with appropriate manipulations, such animations can be made reasonably efficient [1].

Such an approach seems preferable to executable specification languages. It too provides a behavioural model of the system, but without sacrificing abstraction levels. It supports rapid prototyping and even more powerful simulation, but prototypes and simulations are not used in future development. The refinement of the specification to a lower-level implementation, augmented with the discharge of various *proof obligations* ensures that the eventual implementation in a conventional (procedural) programming language satisfies the specification.

Work on executable specifications and specification animation needs to be expanded further. It is one way (and perhaps one of the more effective ways) to increase the acceptance of formal methods by demonstrating that such methods can aid in prototyping and simulation, giving tangible evidence of the satisfaction (or otherwise) of system requirements, and increasing productivity and reducing development costs.

7. Guideline 6: Running industrial courses to train professionals

Since the job of software developers is product oriented, they require a different kind of education than that typically taught by research institutions and computer science departments. The ideal approach for educating the practitioners is to develop a curriculum for a graduate professional degree (analogous to an MBA degree but perhaps with less course work). Such a curriculum would cover the necessary background for using formal methods (e.g. discrete mathematics courses covering sets and logic) and would present a variety of principles, tools, and skills in applying formal methods during software development. Such a professional curriculum is, unfortunately, not very practical now but it should be considered for near future.

The professional degree is not the only approach. A good deal of knowledge of formal methods for software engineering can be found in professional workshops in industry and can be attained through apprenticeship. Typical workshops on formal methods present concepts and comparisons of various types of specifications for different software components (e.g. data structures, files, single procedures, composite objects, programs, etc.). Examples are developed and the relationships between formal specifications and other topics such as logic programming, program verification and 'clean-room development' are illustrated. We suggest the following hints for the information systems professionals:

- Training in discrete mathematics covering elementary set theory and logic should be the first step. For those who have a mathematical background but are unfamiliar with the basic concepts of set theory and propositional logic one or at most two days suffices to introduce the ideas. For others one week of training is required.
- Training in a particular formal method such as Z or VDM should be the next step. Such training typically takes three to five weeks, once the participant has the necessary mathematical background. After such a short training, an individual will be able to read and write formal specifications but not necessarily for complex systems. In order to handle complex systems confidently, gradual training and continuous practice is required.
- Tutoring and consultation in a real project is helpful, so is participation in workshops where one can study a problem and describe it formally with the help of a tutor.

8. Guideline 7: Educating the future generation of software engineers

Educating students, who are our future software engineers, in formal methods is important because we will be preparing them for career growth and through them, we will infuse the formal methods technology into industry. In this section, we *emphasize* curriculum materials which are in the direction of increased formalism in software development. This includes discussion of necessary course work and tools that would help students appreciate the need for formal methods. For additional details, please see Saiedian's pedagogical work [21,22]. Note that our goal here is not to propose a new curriculum; nor is it to single out any particular department's curriculum (although we do believe that many current undergraduate curriculums are focused on the very narrow areas of computer science and programming and that treatment of mathematics and logic in these curriculums is often quite shallow). At the end of this section, we discuss the importance of tools to accommodate students' understanding of formal concepts.

8.1. Formal courses

In this subsection we would like to emphasize the courses that we believe should be emphasized more strongly in the computer science curriculums for exposing the students to the concepts related to formal methods.

Discrete mathematics. Discrete mathematics is a study of calculations involving a finite number of steps and is the foundation for much of computer science. It focuses on the understanding of concepts and provides invaluable tools for thinking and problem solving. Discrete mathematics is especially important when a computer science student is not required to study much of ancillary mathematics.

Computer science students should take at least a one-semester course in discrete mathematics covering fundamental topics such as set theory, functions, relations, graphs, and combinatories. There have been few attempts to teach these topics to freshmen/sophomores in a thorough fashion that would relate discrete mathematical concepts to computer science and software development.

Mathematical logic. Lôgic is fundamental to many of the notations and concepts in computing science. Mathematical logic allows students to formulate and solve a wide class of problems mathematically, is fundamental in understanding the meaning of algorithms, and represents the foundation of logic programming. Students thus must have a deep understanding of concepts such as decision procedures and higher order logic, and the relationship between set theory and lambda calculus. A one-semester course which focuses on these concepts is essential.

The mathematical logic course, together with the discrete mathematics course, should enhance a student's ability of abstract specification and the mathematical skills for specifying, manipulating, and analysing programs.

Formal specification. In addition to the above courses, students should take a formal specification project course. Such a course should tie together the abstract concepts learned in the discrete mathematics and logic courses and provides an opportunity to make practical use of these concepts.

Students must have sufficient experience to be able to appreciate the need for proper specification. This experience may be developed in such a class and could come from the ad hoc development of a software of some complexity or, better still, from attempts to modify a poorly documented and poorly modularized system. Such a course should not simply survey several different approaches but rather give an in-depth practice with one or two approaches that have proven useful (e.g. Larch, VDM, and Z).

One misconception about formal methods is that they are too mathematical and too complicated, requiring a PhD to understand them. Formal methods *are* based on mathematics. However, the mathematics of formal methods are not difficult to learn. Using them requires some practice, but our observation is that such practice is not difficult and that people with only high school math can develop the skills to write good formal specifications. Most popular formal specification languages (e.g. Z and VDM) employ only a limited branch of mathematics consisting of set theory and logic. The elements of both set theory and logic are easily understood and are taught early in high school these days. Certainly, anyone who can learn a programming language can learn a specification language like Z. In fact learning a specification language such as like Z should be easier than a programming language like COBOL. Z is smaller; it is abstract and is implementation-independent. For example, Z uses data types like sets instead of a programming language's types like arrays. This kind of representation captures the essence of what is required better than the corresponding implementation structures. The specification of a problem in Z is shorter and much easier to understand than its expression in a programming language like COBOL.

The need for a broader use of mathematical techniques and concerns for lack of rigour and accountability in software engineering is not felt just by the computer scientists. Consider, for example, the 1990 report released by the Subcommittee and Oversight of the US House of Representatives Committee on Science, Space and Technology. This report addresses the problem of software reliability and quality and criticizes universities for not providing adequate education for software engineers. In an article summarizing this congressional report, Cherniavsky [23] writes:

> [... there is] a fundamental difference between software engineers and other engineers. Engineers are well trained in the mathematics necessary for good engineering. Software engineers are not trained in the disciplines necessary to assure high-quality software ...

8.2. Tools for students

A glance at the structure of most popular formal methods (e.g. Z and VDM) will show that elementary set theory and mathematical logic are of prime importance in these systems and are heavily used in the context of software engineering. Students need to be familiar with these concepts and how they provide a basis for precise definition of the entities we perceive in an information system. Both of these concepts are covered in a *discrete mathematics* course. (It is called discrete mathematics to distinguish it from the continuous mathematics of real numbers that include differential and integral calculus.) Discrete mathematics is a study of calculations involving a finite number of steps and is the foundation for much of computing science. It focuses on the understanding of concepts and provides invaluable tools for thinking and problem solving. Discrete mathematics is especially important when a student is not required to study much of ancillary mathematics. Students should be taught the skills for formalizing problems and behaviours and adjusting the level of rigour to fit software development processes.

It has been our experience that students learn more by active participation than by just observing. Theoretical concepts (such as discrete mathematics and graph theory concepts) should be reinforced with hands-on experience in labs. Since such courses should be taught early in college (to provide the necessary background for high-level courses), educators must ensure that students learn the concepts well. As is often the case, students have difficulty with theoretical concepts that are described in books using definitions, theorems, and proofs. A tool which visualizes theoretical concepts and allows a student to experiment with these concepts creates an attractive environment. Such a tool is helpful, for example, in solving various graph theory problems which would be tedious to work by hand, and would allow easy construction, easily manipulation and flexible composition of graphs. Freed from the mechanical aspects of these tasks, students can focus their attention on the concepts which form the basis of the material being studied.

Several discrete mathematics tools, such as *SetPlayer*, have been developed at Rensselaer that could be used to enrich the undergraduate computer science classes. All of these tools include a help facility or user manual. *SetPlayer* is an interactive command-driven software system for set manipulation. It can be used as a research and educational tool in discrete mathematics. A novel feature of the system is its ability to manipulate sets represented *symbolically*.

For formal specification purposes, a tool, with similar functions as the above tools can help students in many ways. For example, an integrated environment may provide specialized editors, a static analyser (parser and type checker), and refinement tools. A specialized user-friendly editor is of significant importance since most formal methods use mathematical notations not available in traditional editors. Visualization is important and can help students learn the concepts more effectively. A specialized environment for popular formal notations such as Z or VDM can assist students in the creation of specification schemas through the use of a visual notation and present the essential structures in diagrammatical form that could enhance learning. Furthermore, a tool that would extract from specifications in Z or VDM a definition for another diagrammatical tool, e.g. a structure chart, and generate the corresponding charts, would be even more interesting as it would teach students how a formal methodology relates to traditional approaches. Since students may already be familiar with traditional approaches, they can relate to and learn the formal methods approach when they can relate it to concepts with which they are already familiar.

A number of formal methods incorporate tool support as part of the method itself, although we have not directly used them in the classroom to see their effectiveness. Examples include OBJ which offers executable subsets, Larch which offers theorem prover, and ZTC and *f*uzz which offer type-checking for Z. ZTC is a PC-based public domain software while *f*uzz is a relatively inexpensive commercialized system that runs under UNIX. CADiZ also offers a suite of tools for Z and supports refinement to Ada code.

Two tools that we quite often use for pretty-printing of Z specifications include the LATEX 'style' macros zed.sty and oz.sty. Both of these macros are freely available electronically via anonymous FTP. When using these macros, students no longer need to hassle with their editors to typeset special symbols and/or schema boxes. Every Z construct or symbol can be typed in through an ASCII terminal and either one of the above LATEX macros can be used to generate beautiful Z output. For example, one can type in the ASCII text given below:

```
\begin{schema}{BlockRequest}
    \Delta Storage \\
    user?: USERS \\
    block!: BLOCKS
\where
    free \neq \emptyset \\
    block! \in free \\
    free' = free \zhide \{block?\} \\
    dir' = dir \union
           \{block? \mapsto user?\}
\end{schema}
```

to obtain the following typeset Z output:

BlockRequest

$\Delta Storage$
$user? : USERS$
$block! : BLOCKS$

$free \neq \emptyset$
$block! \in free$
$free' = free \setminus \{block?\}$
$dir' = dir \cup \{block? \mapsto user?\}$

As it can be observed, these LATEX macros can save students' time that would otherwise be spent on the mechanical and time-intensive aspects of preparing specifications in Z notation. Both of these two macros can quickly be learned by LATEX users. Similar macros have been developed for VDM.

Yet another aspect of a tool that frees students from the mechanical aspects of developing formal schemas and allows them to concentrate on the conceptual modelling aspects of software development, is to provide pop-up and pull-down menus for selecting specification schemas, verifying the well-foundedness of specification schemas, and mapping them into more concrete definitions and programs to make a prototype implementation. When these mechanical aspects of conceptual modelling and mapping into an operational model are handled by a tool, students can concentrate on how to improve the quality of representations and reason about the correctness of models. Student productivity and knowledge in formal methods will be substantially increased when using an easy-to-use environment, with context-sensitive help and debugging, analogous to the program development productivity improvement 'Turbo Pascal' brought to beginning programming students during the 1980s.

9. Conclusions

We, like so many other proponents of formal methods, believe that system specification via rigorous mathematical notations can help to eliminate (or at least ameliorate) many of the problems associated with software engineering, such as ambiguity, imprecision, incompleteness and inconsistency. Errors may be discovered and corrected more easily, not through an ad hoc review, but by the application of mathematical reasoning. We believe that formal methods can be particularly useful during the early stages of software engineering, and enable the software developer to discover and correct errors that otherwise might go undetected, increasing the quality of the software and its maintainability, while decreasing its failure rate as well as its maintenance costs. Although such ideas are the objectives of all software development methods, the use of formal methods results in much higher likelihood of achieving them.

Unfortunately, industrialists have been slow to accept these ideas, and the uptake of formal methods has been much slower than one would expect, and desire. We have highlighted some areas whereby we feel the proponents of formal methods fail to 'sell' them adequately.

While it may be relatively easy to educate students in formal methods within an academic setting, it is less easy to convince industry to accept such methods. Regardless of how many case studies are presented, information systems managers, who rarely have a technical degree, are still fearful of what the consequence may be in terms of re-education and/or training of present practitioners, the long-term influence of formal methods on the software engineering process, and the change-over from ad hoc approaches to formal methods. (Managers often equate formal methods with the theoretical underpinning of programming or engineering practices.)

We believe, however, that with greater marketing, greater emphasis on *when* formal methods are required, with appropriate reuse of formal methods and the use of formal methods in prototyping and simulation, greater emphasis on tool support and educations, we may succeed in transferring formal methods technology into the actual workplace. We also believe in a pragmatic approach to the education of future system developers (thorough grounding in discrete mathematics, mathematical logic, and formal methods) so as to prepare our new graduates for the future application of such techniques, while slowly achieving the transfer of the technology itself into industry.

Acknowledgements

We would like to thank anonymous reviewers for their fine suggestions that improved the presentation of our article. Hossein Saiedian's research was partially funded by a UCR grant from the University Committee on Research, University of Nebraska at Omaha.

References

[1] Fuchs, N E 'Specifications are (preferably) executable', *IEE/BCS Software Engineering J.* Vol 7 No 5 (September 1992) pp 323–334

[2] Hall, A 'Seven myths of formal methods', *IEEE Software* Vol 7 No 5 (September 1990) pp 11–19

[3] Bowen J P and Hinchey, M G 'Formal methods and safety-critical standards', *IEEE Computer* Vol 27 No 8 (August 1994) pp 68–71

[4] Bowen, J P and Hinchey, M G 'Seven more myths of formal methods', *IEEE Software* Vol 12 No 4 (July 1995) pp 34–41

[5] Craigen, D, Gerhart, S and Ralston, T 'An international survey of industrial applications of formal methods', (March 1993) NISTGCR 93/626, US Department of Commerce

[6] Hinchey, M G and Bowen, J P, (eds) *Applications of formal methods* Prentice-Hall (1995)

[7] Dix, A J *Formal methods for interactive systems* Academic Press (1991)

[8] Weber-Wulff, D 'Selling formal methods to industry', in *Formal Methods 93*, LNCS 670, Springer-Verlag (1993) pp 671–679

[9] Leveson, N 'Software safety in embedded computer systems' *Commun. ACM* Vol 34 No 2 (February 1991) pp 34–46

[10] Bowen, J P and Hinchey, M G 'Ten commandments of formal methods' *IEEE Computer* Vol 28 No 4 (April 1995) pp 56–63

[11] Parnas, D L 'Using mathematical descriptions in the inspection of safety-critical software' in Hinchey and Bowen (eds) *Applications of Formal Methods* Prentice-Hall (1995)

[12] Craigen, D 'Tool support for formal methods in *13th Int. Conf. on Soft. Eng.* IEEE-CS (1991) pp 184–185

[13] Garlan, D and Delisle, N 'Formal specifications as reusable frameworks' in *VDM '90* LNCS 428, Springer-Verlag (1990) pp 150–163

[14] Boyle, J M 'Abstract programming and program transformation: an approach to reusing programs' in Biggerstaff, T J and Perlis, A J (eds) *Software reusability: concepts and models* ACM Press (1989)

[15] Hinchey, M G and Jarvis, S A *Concurrent systems: formal development in CSP* McGraw-Hill (1995)

[16] Hekmatpour, S and Ince, D C *System prototyping, formal methods and VDM* Addison-Wesley (1989)

[17] Hayes, I J and Jones, C B 'Specifications are not (necessarily) executable' *IEE/BCS Software Engineering J.* Vol 4 No 6 (November 1989) pp 330–338

[18] West, M M and Eaglestone, B M 'Software development: two approaches to animation of Z specifications using Prolog' *IEE/BCS Soft. Eng. J.* Vol 7 No 4 (July 1992) pp 264–276

[19] O'Neill, G 'Automatic translation of VDM specifications into standard ML programs' *The Computer J.* Vol 35 No 6 (December 1992) pp 623–624
[20] Valentine, S H 'The programming language Z^{--}'. *J. of Inf. and Soft. Technol.* Vol 37 No 5/6 (May/June 1995) pp 293–302
[21] Saiedian, H 'Mathematics of computing' *Computer Science Education* Vol 3 No 3 (1992) pp 203–221
[22] Saiedian, H 'Towards increased formalism in software engineering education' *ACM SIGCSE Quarterly Bulletin* Vol 25 No 1 (March 1993) pp 193–197
[23] Cherniavsky, J C 'Software failures attract congressional attention' *Computer Research Review* Vol 2 No 1 (January 1990) pp 4–5