Information Systems and the Engineering Paradigm: Integrating the Formal Methods into the Development Process*

Hossein Saiedian

Department of Computer Science, University of Nebraska, Omaha, U.S.A.

Formal methods are mathematically-based techniques which can be used for rigorous modelling, analysis, specification, and design of information systems. We discuss the role of formal methods in the context of an engineering paradigm and how it applies to information systems development. An introduction to precise, concise, and unambiguous description of information systems using a formal method is provided. An example is given to illustrate the effectiveness of formal methods in describing an information system. The misconceptions about the use of formal methods are examined, and guidelines for transferring formal methods technology into the actual workplace are presented. This discussion emphasizes automated tools to assist in developing systems specification, marketing of formal methods through additional industrial-strength case studies, and the need for a framework for reusable specifications. The overall aim of the paper is to narrow the gap between the results of academia and pragmatic concerns of the information systems industry, emphasizing the importance of formal methods in the development of information systems and encouraging further progress in information systems methodologies.

Keywords: Formal Methods, Engineering Paradigm, Software Development, Language Z

1. Introduction

The increased cost of software has become a major concern of the computing industry. It is reported that, in the mid 1950s, software cost was less than 20% of total computing cost, but now it exceeds 80% (Biermann 1990). Precise and up-to-date figures representing the actual cost is neither available nor is easy to establish. Boehm (1987) suggested that in 1985 worldwide software cost exceeded $ 140 billion and was growing at a rate of 12% per year. According to this statistic, software cost will reach $ 435 billion by the year 1995. Small improvements in software development can, therefore, result in a substantial reduction in this high cost. A large portion of software cost is due to its maintenance. The maintenance of existing software systems can account for over 70% of all efforts expended by a software organization (Pressman 1992). This percentage continues to rise. Researchers have indicated that a lack of rigorous practices which eliminate residual errors in specification and produces precise and unambiguous specification in software development activities has translated into a large portion of these maintenance costs. Cost-effective development of reliable information systems has been extensively studied as a central issue in information systems development. A considerable number of principles have been explored and some have proven effective in practical projects. These include software development methodologies and environments such as the phased approach, top-down design, structured programming, CASE tools, 4GLs, application generators, and so forth. Nevertheless, problems in reliable software development are not completely resolved and many enterprises still suffer enormously from the high cost of software. The concern for reliable and cost-

* This research was supported in part by a UCR grant from the University Nebraska at Omaha.

effective construction of information systems has led to a major emphasis in discovering new techniques to improve information systems development methodologies.

2. Information Systems as an Engineering Discipline

The high cost of software has not been the only concern. Another major concern of the computing industry has been the reliability of its information systems. Industry has enhanced its information system infrastructure, and information system are built in areas such as air traffic control, airline reservations, military command and control, hospital on-line patient record management, stock transactions. These systems play an important role in our daily lives and the competitiveness of our country. Incorrect functionality or inefficient performance of these systems could have major impact on our safety and well being. Some of these systems will have hundreds of thousands of lines of code. The complexity of these systems may supersede the complexity of any artifact ever built by mankind. The above characteristics of information systems imply that the specification, design, implementation, and validation of these systems is an engineering process of considerable difficulty comparable to other complex engineering products. The question is, are our future information system specialists prepared for the design and maintenance of such systems? Will they be concerned with the efficiency, reliability, safety, ease of use, and robustness of their information system products just like engineers are concerned with their products? If we are to use traditional engineering disciplines as guides for our information system design, we have to understand the engineering methods and the process by which engineers have achieved their goals. (Note that the emphasis here is not necessarily on what engineering is but on what engineers do.) Although these methods are many and varied, we like to emphasize on the application of science and by extension mathematics in the activities of practicing engineers. By doing so, we like to assert that formal methods may mirror these activities in the area of information systems. Combining the investigation into engineering uses of mathematics with our taxonomy of formal methods, we hope to establish the important place of formal methods in information system development, point to the parallel between mathematics/engineering and formal methods/information systems, and emphasize the importance of introducing the concept of formal methods in information systems curriculum. There are a number of excellent references on what constitutes an engineering process. One such reference is Koen (1985). According to Koen, the engineering method is "the strategy for causing the best change in a poorly understood or uncertain situation within the available resources," and further explains that engineers base their work on choosing the best heuristics from a pool of given heuristics. Indeed, what distinguishes an engineering practice is its assessment of a body of heuristics that succeed more often than they fail. However, the most important aspect of Koen's thesis and the one which we emphasize is his emphasis on use of science, and by extension, mathematics part of the engineering process. (The information system also has its own heuristics, e.g., avoid goto, limit modules to one page, avoid excessive global data definition, etc. It is important to use information system heuristics that have proven effective. Our goal is to emphasize the more important thesis of Koen, that of the application of mathematics.) Although space does not permit detailed discussion of Koen's thesis and its implications, the overall theme, i.e., the emphasis on the role of science and mathematics, provides a framework for judging the place of formal methods, which in our opinion play an analogous role in information systems as mathematics do in engineering. Great achievements in past chemical, electrical, industrial, civil and mechanical engineering have been firmly based on an understanding and application of formal methods. The time has come to use formal methods for information system development and to construct a pragmatic framework for teaching the basic concepts to newcomers.

3. Objectives and Organization of the Paper

We believe that formal methods should be recognized as an important element of information systems development process and should be the foundation for various techniques of validation of software including verification, testing, and

walk-through reviews. The aim of this paper is to contribute to the wider use of formal methods in information systems development. Our objective is not to instruct the readers in the theoretical world of formal methods, but rather in the importance of their application in developing information systems. The organization of this paper is as follows:

- We first discuss formal methods and then discuss the effectiveness of specifications that are *formal*, i.e., based on a mathematical approach.
- We will then give an example to show how a formal system, called Z, can be used to formally describe the functionality of an information system. This example is a simplified version of an airline reservation system; nevertheless, the effectiveness of formalism is illustrated.
- We then examine the roots of some common misconceptions about formal methods.
- Finally, the paper ends with a discussion of how to transfer the formal methods technology into the workplace and focus on further research areas for making formal methods more widely used.

Our overall aim is to narrow the gap between the results of academia and pragmatic concerns of industry as well as emphasizing the importance of formal methods in the development of information systems and encouraging further progress in information system methodologies.

4. Formal Methods

A method is defined as a set of *procedures* (or guidelines) for *selecting* and applying *tools* and techniques to *construct* a (reliable) artifact (in this case, an information system). A tool is formal if it can be mechanically manipulated according to some well-defined, mathematically-based rules. That is, the the tool should have a precise mathematical semantics. (Note that a notation is a form of tool.) If a method uses formal tools (or notations), then it is called formal. The objective of formal methods is to achieve rigor in reasoning as well as achieving mechanical support. Formal methods of software development use a mathematical notation. By mathematics, we do not mean derivations, integration, differential equations, and such. Most popular formal methods employ only a limited branch of mathematics consisting of set theory and logic. The elements of both set theory and logic (which are taught in discrete mathematics classes) are easily understood. Such notation is used *precisely* describe the properties of a software without unduly constraining the way in which these properties are to be realized. In other words, they describe what the system must do without saying how it is to be done. This abstraction is very useful in developing an software since it allows questions about the properties of the system to be answered *confidently* without speculating about the meaning of certain phrases in an imprecisely worded prose description. Furthermore, formal methods provide a basis for logical reasoning about the artifacts produced. (These artifacts may be abstract in terms of specifications, or they may be concrete, in terms of programming code.) We believe that it is important to build an (abstract) mathematical description of an information system using formal methods before building the system itself. The reason for doing so is to achieve more precision in the descriptions and to explore the validity of the design by reasoning about the description. Since formal methods deal with the semantics aspect of an information system, they can help us determine which functional properties to capture in abstract specification and which to focus on in concrete design.

5. Formal Methods: Modelling and Specification

Formal methods can be applied during the specification, design, and implementation phases of information systems development. Some formal systems like Z (Diller 1994) are considered as a modelling and specification tool while a system like the VDM (Jones 1990) emphasizes design. Dijkstra's "weakest precondition" approach can be used during the implementation to prove the correctness of a program. Most formal methods support both system modelling and reasoning. (Of course the degree in which emphasis is shifted from modelling to reasoning is varied in different approaches.) A formal

system like Z can be used to produce a clear and concise model of an information system. This by itself is of sufficient value to encourage and warrant the use of formal methods. This in fact is our emphasis. That is, we emphasize the use of formal methods for specification and modelling. (Of course a formal model of a system produced by a methods like Z can be used in conjunction with verified design by a series of refinement during which abstract descriptions are transformed into more concrete design structures. While this transformation is being done, *inference rules* can be used to uncover inconsistencies.) The development of any large system has to be preceded by a specification of what is required. Without such a specification, the system's developers will have no firm statement of the needs of the would-be users of the system. The need for precise specification is accepted in most engineering disciplines. Information systems are in no less need of precision than other engineering tasks. We place special importance on the requirement analysis and specification phase since we believe that precise and concise specifications are an essential prerequisite to the successful development and evolution of a system. If the system is not precisely specified, the design process could become chaotic, resulting in a system that is complicated, error-prone and difficult to maintain. Many aspects of an information system must be specified including its functionality, performance and cost. In this paper attention is focused on a system's functionality. The requirements specification phase, i.e., when the functions of an software are specified, plays a critical role. Its role is so critical that failure to properly carry out this phase is historically known to cause great financial losses. Hence requirements specification has been the subject of great deal of attention in recent years. The existence of workshops and conferences devoted to this subject, e.g., the *International Workshop on Software Specification and Design, Formal Methods Europe, ACM SIGSOFT International Workshop on Formal Methods in Software Development and Workshop on Industrial-Strength Formal Techniques* bear witness to the importance of this field. Requirements can be specified informally via a natural language. Informal specifications alone are certainly not appropriate because they are normally inconsistent, inaccurate, ambiguous and they rapidly become bulky and it would be very difficult to check for their completeness. Semi-formal approaches to specification have been developed since the 1970's to improve the practices used in software development. A particular emphasis in semi-formal approaches (e.g., data flow diagrams) is on graphical representation of the software being built. Major problems with semi-formal methods include:

- their lack of a precise semantics that can be used to reason about or verify the properties of the software, and
- generally "free" interpretation.

To overcome the limitations of informal and semi-formal approaches, we emphasize using *formal methods* to produce precise and unambiguous specifications (known as *formal specifications*). A formal approach is also needed to achieve precise communication among various users and developers of a system. Precise communication is critical because in large projects a group of people must arrive at a consistent understanding of the proposed system in order to successfully construct it. We do not assert that informal (e.g., natural language descriptions) and semi-formal specifications are useless. Clear use of a natural language obviously has a place in describing systems. In fact, they may be very useful as a first introduction to a software and as comments to enhance the readability of formal specifications. Thus formal and informal specifications must not be regarded as the only alternatives but rather as complementary. In order to achieve *precision*, however, a specification formal. Figure 1 depicts the role of the requirements specification document in the software process. In addition to precision in specifications, formal methods enable the designer to use *rigorous mathematical reasoning* to prove the properties of the software. Thus design errors (e.g., inconsistencies, incompleteness, contradictions) can be detected in an early stage of development.

6. An Example: A Flight Reservation System

We mentioned that formal methods can be used during both early stages of software development for modelling and specification purposes and in later stages for verification, formal walkthrough, or to show the equivalence of different implementations. For this example, the use

of formal methods for specification is emphasized. We provide an example in Z to show the effectiveness of formal methods in the specification of an information system. An important feature of Z is that it provides a framework within which a specification of a system can be constructed incrementally. The main building block of this framework is a two-dimensional graphical structure called schema. Schemas group all the relevant information that belongs to a system, and describe its static aspects (e.g., the states it occupies) and the dynamic properties (e.g., the operations that are possible on the state). A complete Z specification is constructed from schemas which themselves can be expressed in terms of other schemas. (This is similar to construction of a program by a series of program modules which can be read and understood in isolation.) A schema has the following structure:

SchemaName

Signature
Predicate

The signature introduces variables that define the state and is analogous to the declaration part of a Pascal program. The predicate of a schema refers to variables (both local and "global"—those that are defined in other schemas) and relates the values of these variables to each other. (To reduce unnecessary clutter, entries of declaration and conjuncts of predicate are placed on separate lines. However, it is possible to place more than one declaration entry, separated by a semicolon, on each line. Similarly, conjuncts of predicate can be placed on a single line and explicitly connected by an "and" connective.) The signature and predicate part of a schema are separated by a horizontal bar which is pronounced as "such that." There is an equivalent linear notation for schemas shown as [*Signature* | *Predicate*]. To illustrate the purpose of each part of a schema more specifically, we will give the specification of a simple Flight Reservation System (FRS). For the sake of simplicity, we assume there is only one type of flight. (It will not be difficult to extend this specification to include more than one flight.) Since our main objective is to illustrate the concepts of formal specification in Z, we'll keep the example very small and simple by ignoring other aspects of reservation such as destination, time, date, etc. Only reservation and cancelation operations and simple queries are considered.

6.1. State of the Flight Reservation System

The first step is to define the state of the FRS:

FRS

$Passengers : \mathbf{P}\ Person$
$\#Passengers \leq Capacity$

The state of the FRS has only one item called *Passengers* which is a set of *Person*. (In Z, **P** stands for 'set of'. *Person* is assumed to be a pre-defined type.) The predicate of the FRS is an invariant expression that says the size of the *Passengers* set must always be less than or equal to *Capacity*. (*Capacity* is a value representing the maximum number of passengers for the flight. A # symbol in Z represents the size operator.) Next we specify the initial state of FRS:

InitFRS

FRS'
$Passengers' = \{\}$

which essentially specifies that the *Passengers* set is initially empty. (The purpose of prime symbol is explained below.)

6.2. Operations on the Flight Reservation System

We first specify the *Reserve* operation:

Reserve

ΔFRS
$p? : Person$
$\#Passengers < Capacity$
$p? \notin Passengers$
$Passengers' = Passengers \cup \{p?\}$

- The declaration ΔFRS alerts us to the fact that the schema is describing a state change:

the state of the FRS will be modified because of the operation.

- Variable p is input to the *Reserve* operation. Note that the question mark, by the common convention, specifies that p is an input.
- The predicate part of the schema consists of three predicates. The first two are essentially the pre-conditions for the success of the operation. The first one states that the number of passengers booked for the flight should be less than the maximum capacity, while the second one specifies that the input p should not already be in the set of *Passengers*. (The latter predicate disallows multiple reservations by the same passenger).
- The last predicate is the post-condition of the operation and states that after the operation, the set of passengers includes input p. Conventional notation of adorning a variable with the prime symbol to show the new value of unprimed variable is adopted by Z. Thus, the notation *Passengers'* reflects the new value after the operation.

The operation for cancelation is specified below:

Cancel

ΔFRS
$p? : Person$

$p? \in Passengers$
$Passengers' = Passengers \setminus \{p?\}$

Note that in the above, the pre-condition states that a passenger must have a reservation for the cancelation to succeed. (Symbol is for set subtraction.)

Strengthening the Specification To strengthen the specifications we have to specify what is to happen when the pre-condition(s) of an operation are not fulfilled. Because Z allows incremental specifications, we can separately specify all error cases and use Z schema calculus to combine them. For the *Reserve* operations, the pre-conditions are:

$$\#Passengers < Capacity$$
$$p? \notin Passengers$$

When either of the above pre-conditions is false, we would like to output a suitable error message. A schema is constructed for each of the above:

FlightFull

ΞFRS
$response! : Report$

$\#Passengers' = Capacity$
$response! = \text{'Flight is full'}$

AlreadyReserved

ΞFRS
$p? : Person$
$response! : Report$

$p? \in Passengers$
$response! = \text{'Already reserved'}$

Note that:

- when an error occurs, the state of FRS should not be altered. The declaration ΞFRS indicates this fact.
- Variable response is used for outputting messages and is of pre-defined type *Report*. Output items in Z are decorated with an exclamation point.

For the sake of completeness we introduce the following schema:

Success

$response! : Report$

$response! = \text{'Successful'}$

to output 'Successful' when an operation has been carried out successfully. It is now possible to construct a robust version of *Reserve* by using operators of Z schema calculus:

$$RReserve \triangleq Reserve \wedge Succes$$
$$\vee$$
$$FlightFull$$
$$\vee$$
$$AlreadyReserved$$

The above notation introduces a new schema called *RReserve* obtained by combining four existing schemas. *RReserve* is in fact made of four schemas. Its signature can be pictured

as containing all variables of *Reserve, Success, FlightFull,* and *AlreadyReserved*. Likewise, its predicate is a correct combination of the predicates of these four schemas. If expanded, *RReserve* would look like the following:

RReserve

$$
\begin{array}{|l}
\hline
Passengers, Passengers' : \mathbf{P}\ Person \\
p? : Person \\
response! : Report \\
\hline
(\#Passengers' < Capacity \\
\quad p? \notin Passengers \\
\quad Passengers' = Passengers \cup p \\
\quad response! = \text{'Successful'}) \\
\vee \\
(\#Passengers = Capacity \\
\quad response! = \text{'Flight is full'} \\
\quad Passengers' = Passengers) \\
\quad \vee \\
\quad (p? \in Passengers \\
\quad response! = \text{'Already reserved'} \\
\quad Passengers' = Passengers) \\
\hline
\end{array}
$$

The definition of *RReserve* could have been developed like the above without the use of schema calculus. But such an approach would have been tedious and prone to errors. The schema calculus makes specification of complex operations a lot simpler and manageable by providing operators to combine existing schemas. The robust version of *Cancel* operation can be developed similarly. We first define a new schema called *NoSuchReservation*:

NoSuchReservation

$$
\begin{array}{|l}
\hline
\Xi FRS \\
p?: Person \\
response! : Report \\
\hline
p? \notin Passengers \\
response! = \text{'No such reservation'} \\
\hline
\end{array}
$$

The above schema is combined with *Cancel* to form the *RCancel* operation:

$$RCancel \triangleq (Cancel \vee Succes) \wedge (NoSuchReservation)$$

6.3. Querying the Flight Reservation System

Two examples of query operations are given below. One is used to determine whether a certain passenger has a reservation:

QueryReservation

$$
\begin{array}{|l}
\hline
\Xi FRS \\
p?: Person \\
response! : Report \\
\hline
(p? \in Passengers \wedge response! = \text{'OK'} \\
\vee \\
(p? \notin Passengers \wedge \\
response! = \text{'No reservation'}) \\
\hline
\end{array}
$$

the next is for determining the total number of passengers:

HowManyPassengers

$$
\begin{array}{|l}
\hline
\Xi FRS \\
count! : \mathbf{Z} \\
\hline
count! = \#Passengers \\
\hline
\end{array}
$$

Note that neither one of the above query operations alters the state of the FRS.

6.4. Observations

As it can be observed from the above, formal specifications can aid in arriving at the most useful abstract description of a system because they highlight the relevant details of a system. Formal specifications given in a language like Z allow one interpretation and, unlike programming languages, they are not required to contain efficiently implementable constructs and thus eliminate unnecessary details. This attribute contributes to the readability, minimality, and precision of specifications. The advantages of formal modelling and specifications, as evident from the above example, can be summarized as follows:

- Preciseness. Formal specifications allow requirements of an information system to be recorded accurately.

- Unambiguous. Different interpretations are avoided because the constructs used have well-defined meaning.
- Conciseness. A mathematical notation is capable of expressing complex facts about information systems in a short space.

6.5. A Few Comments about Z

The design of Z illustrates that informal (English) descriptions are complementary to formal specifications. As it can be seen from the above example, small chunks of formal descriptions, framed in schema boxes, are surrounded by informal prose, explaining the relationship between the formal expressions and reality, thus motivating discussion and subsequently decisions that are captured by formalism. Note that in the above example, the English statements were used primarily to explain the Z notation. Z also reminds (and allows) us to begin the specification with a simple description of the overall structures of an information system (with as few variables/operations as possible) and then expand the descriptions (while leaving the initial descriptions unchanged). This is done by: (1) embedding the initial general schema inside other schemas, and (2) forming new schemas by connecting existing schemas through schema connective operators. Such an approach encourages and facilitates modularity: individual requirements are defined separately and then they are joined together by connective operators. The schema calculus of Z is based on simple connective operators ($\wedge$, $\vee$, $\Longrightarrow$ and negation).

7. Misconceptions about Formal Methods

There are a number of common misconceptions about the uses of formal methods. For example, it is stated that formal methods are difficult to use, too mathematical, only applicable to trivial academic problems, and not widely used. In this section, we examine a number of such misconceptions. Many of these and other misconceptions are discussed in greater detail in the context of "seven myths of formal methods" by Hall (1990). (A recent article by Bowen & Hinchey (1994) discovers and addresses seven more myths of formal methods.)

7.1. Formal Methods are for Program Verification Only

One misconception is that formal methods are for program verification only. A lot of work, especially in academic institutions, has concentrated on using formal methods for program verification. Program verification makes formal methods seem very difficult and technical. In fact, one reason that formal methods is unpopularly perceived is that they are often confused with proving programs corrects. Program verification, however, is only one aspect of formal methods and perhaps the least significant. Program verification is normally applied to the later stages of development well after the system has been modelled and specified. The primary use of formal methods is for writing specifications that precisely define what a system is intended to do.

7.2. Formal Methods are Inapplicable to Real Projects

Another misconception is that formal methods are not used in industry for real projects. The fact is that they are. There are numerous reports on practical and effective use of formal methods in industry (Hinchey & Bowen 1995, Woodcock & Larsen 1993, Hall 1990). The range of applications, from small to large systems, possessing attributes such as real-time, interactive, robust, and secure, has been sufficiently wide to test the viability of formal methods, and to establish a considerable body of knowledge and experience. Several authors have discussed experience gained in applying formal specification techniques to IBM's CICS transaction processing system. CICS is a large, 20-year-old system and contains over half a million lines of code. The language Z was used by IBM to re-specify CICS to improve its maintainability. Industry's concern for mathematical techniques is also evident from a recently completed report by the Computer Science and Technology Board (Press 1990). This report points to the need for strengthened mathematical foundations in information systems and related areas. For example, the report warns that as the software developers begin to envision information systems that require many thousands

of person-years, current pragmatic or heuristics approaches begin to appear less adequate to meet application needs. Therefore, the developers may have to call for more systematic approaches. More mathematics, science, and engineering are needed to meet their call. For an international survey on industrial applications of formal methods see (Craigen, Gerhart & Ralston 1993). This report evaluates international industrial experience in using formal methods, provide an authoritative record on the practical experience of formal methods to date, and suggests areas where future research and technology development are needed.

7.3. Formal Methods are too Mathematical

Another misconception is that formal methods are too mathematical and too complicated requiring extensive mathematical background to understand and use them. Formal methods are based on mathematics. However, the mathematics of formal methods are not difficult to learn. Using them requires some training, but experience has shown that such training is not difficult and that people with only high school math can develop the skills to write good formal specifications. Most popular formal methods 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 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 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 negative perception of the role of mathematical techniques in requirements specification is very unfortunate. When problems become very large and complex in other engineering disciplines, they normally turn to mathematics for help. Unfortunately, some software developers feel that formal methods and mathematical tools are of academic interest only and that real problems are too large and complex to be handled by formal methods and mathematical tools. The need for a broader use of mathematical techniques and concerns for lack of rigor and accountability in software development is not felt just by the academicians. Consider, for example, a recent report released by the Subcommittee and Oversight of the U.S. House of Representatives Committee on Science, Space and Technology (House of Representatives 1989). 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, Cherniavksy (Cherniavsky 1990) 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. Formal Methods and the Workplace

Numerous pragmatic challenges are still ahead in order to transfer formal methods technology into the actual workplace and make them more popular. We will give our guidelines and suggestions in this section. Most importantly, professionals and students must be trained and educated in the use of formal methods. Training and education guidelines are given at the end of this section. (A much more detailed discussion of these and other guidelines is given by Saiedian & Hinchey (1994).)

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 methods 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. In our opinion, automated tools may include:

- Special editing environment
- Syntax checkers
- Animation tool
- Refinement and Proof Tools

A special editing environment for language 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. (There is a specification environment for the formal method Z. It is called ZED. According to Dharap & Narayana (1992), ZED is an interactive and incremental environment for the specification language Z. The basic unit of editing in ZED is a paragraph which can be a schema, a definition, or an axiom, etc. An incremental parser permits the specification to be altered and the effects of changes in the specification to optimally propagate only to those attributes that are affected. An incremental type inferencing algorithm provides for type-checking the specification. A WYSIWYG editor with graphical fonts allows the pleasant display of the paragraphs as they are typed. To permit documenting the specifications, ZED integrates LaTeX and its previewers. Menu-based cut and paste facilities for fonts and for components of the specification enhance the power of ZED.) In addition to the above, good interface 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. Furthermore, a specification is and should be considered a major reference document for the customer as well 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.

Marketing of Formal Methods As mentioned earlier, 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. A study needs to be done to discover what it will take to move formal methods from this unfair perception into a wider acceptability within the information systems community. Case studies must be developed to demonstrate the applicability of formal methods with the intention of convincing the practitioners that the benefits outweigh the difficulties of transition and, results disseminated. 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 not only are necessary for advancing the technology and demonstrating the potential benefits, they also help identify the needs of companies that adopt formal methods and enhance the integration of formal methods with current software engineering practices. In general, formal methods tend to address semantic issues rather than pragmatic issues of a software. Managers and practitioners are, however, most concerned with pragmatics issues. This understanding would help researchers delineate more precisely where formal methods are most useful. Case studies 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, performance, portability, man-machine interfaces, or resource consumption of running programs are difficult, or perhaps even impossible to specify by means of formal methods. Research needs to be done to find out if formal methods can in fact be used for such purposes. It is only in practical applications that the limits or constraints of formal methods are revealed. Debora Weber-Wulff (Weber-Wulff 1993) addresses some of the typical questions in attempting to introduce formal methods into industry, elaborates on other marketing aspects of formal methods, and offers 10 propositions for industrial strength formal methods. Some of these propositions include:

1. An industrial formal method should be confinable, i.e., restricted to just one aspect of a large software project, so that it wont disrupt the rest.
2. The uses of formal methods should be reversible so that if it decided to drop the uses of these methods, it would be possible to

revert to the previous status in the development.

3. A formal methods should be open to allow interchangeability of software components or to allow uses of different tools.
4. A formal methods should not be coercive or overly concerned with imposing and enforcing restrictions on trivial aspects of software development.
5. A formal methods should be teachable.
6. A formal methods should be inexpensive.
7. There must adequate documentation for an industrial formal methods technique.

Clarify "WHEN" to Use Formal Methods An important responsibility of proponents of formal methods is to clarify when in the development process formal methods should be applied. The greatest benefit of formal methods is at the early stages of development for modelling and specification. What normally disappoints practitioners is the mathematics involved in proving programs correct (also known as program verification). Program verification, however, is carried out at the last phase 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. Thus, to ensure wider acceptance of formal methods, the information systems practitioners should be reminded that formal methods have the highest leverage during the specification of a system.

Formal Specifications as Reusable Frameworks Traditional efforts on formal methods have been for functional specification of a software system and have largely focused on abstractions techniques (and refining abstractions into some implementation). To make formal methods an integral part of industrial software development, the use of these methods has to be as cost-effective as possible. One way of achieving such cost-effectiveness is through development of a framework for reuseability of already developed formal specifications. Such a framework has numerous benefits (Garlan & Delisle 1990):

1. If a single specification can serve a number of products, its development cost can be amortized over those products.
2. The development of several products from the same specification can lead to uniformity across those products as well as their development.
3. Reusability in specification may lead to correspondingly reusable products.
4. The fact that a specification can serve as framework for several products, may cause its developers to strive for particularly elegant abstractions. This in turn may lead to cleaner definitions of the fundamental concepts behind related applications.
5. The job of defining specifications for the framework may be delegated to a small team of highly skilled engineers.
6. Libraries of formally specified software components that form the basic design repertoire of software developers may gradually be produced.

Thus research work on a reusable framework for formal specifications should be expanded. The work reported by Garlan & Delisle (1990) is but one such approach used in Tektronix Laboratories.

Executable Specifications and Rapid Prototyping A promising approach to rapid prototyping is the executable specification approach. The basic idea is that if a specification language is formal and has operational semantics, then it is possible to construct a system that can execute it directly. This will reduce the cost of software development and allows rapid prototyping. Work on executable specifications needs to be expanded. It is one way (and perhaps the most effective way) to increase the acceptance of formal methods by demonstrating that such methods increase productivity and can reduce development cost.

Training for 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 development 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 professional:

- 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 one to three weeks, once the participant has the necessary mathematical background.
- 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.

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 development. 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 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 (Saiedian 1992). Students should be taught the skills for formalizing problems and behaviors and adjusting the level of rigor to fit software development processes. (In the U.S., the Software Engineering Institute has initiated such teaching programs.) Students learn more by active participation than just by observing. Theoretical concepts (such as discrete mathematics) should be reinforced with hands on experience in labs. Since such courses should be taught early in college (to provide necessary background for high-level courses), educators must ensure that the students learn the concepts well. As is often the case, the 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 a creative environment. Such a tool is helpful in solving various discrete math problems which would be tedious to work by hand. Freed from the mechanical aspects of these calculations, the students can focus their attention on the concepts which form the basis of the material being studied.

9. Conclusions

Formal methods in the context of overall engineering was discussed. We argued that formal methods enable a software developer to specify a system via a rigorous mathematical notation. Such an approach to specification eliminates many of the problems associated with software development such as ambiguity, impreciseness, incompleteness and inconsistency.

The errors are discovered and corrected more easily; not through an ad hoc review, but through application of mathematical reasoning. When used during the early stages of software development, formal methods enable the software developer to discover and correct errors that otherwise might go undetected, therefore increasing the quality of the software and it maintenance and decreasing its failure rate as well as its maintenance cost. Although such ideas are the objectives of all specification methods, the use of formal methods results in a much higher likelihood of achieving them. While it may be easier to educate students in formal methods within an academic setting, it is less easy to convince the 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 development 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.) In the latter section of this paper, we gave a number of guidelines for making formal methods more acceptable and for transferring formal methods technology into the actual workplace. We also encourage integrating formal methods as part of a pragmatic approach in information systems education (through courses in discrete mathematics, mathematical logic, and formal methods) to prepare our new graduates for the future while slowly transferring the technology itself into industry.

References

BIERMANN A. W., (1990) *Great Ideas in Computer Science*, MIT Press.

BOEHM B., (1987) 'Improving software engineering production', *IEEE Software* 20(9) 43–58.

BOWEN J. & HINCHEY M., (1994) Seven more myths of formal methods, Technical Report PRG–TR–7–94, Oxford University Computing Laboratory, Oxford, England.

CHERNIAVSKY J. C., (1990) 'Software failures attract congressional attention', *Computer Research Review* 2(1) 4–5.

CRAIGEN D., GERHART S. & RALSTON T., (1993) 'An international survey of industrial applications of formal methods'. NISTGCR 93/626, U.S. Department of Commerce.

DHARAP S. & NARAYANA K., (1992) 'The ZED environment', (An article posted to the USENET newsgroup comp.specification.z).

DILLER A., (1994) *Z: An Introduction to Formal Methods*, 2nd Edition, John Wiley.

GARLAN D. & DELISLE N., (1990) Formal specifications as reusable frameworks, in 'VDM'90', *LNCS 428*, Springer–Verlag, pp. 150–163.

HALL A., (1990) 'Seven myths of formal methods', *IEEE Software* 7(5), 11–19.

HINCHEY M. & BOWEN J., eds (1995) *Application of Formal Methods*, Prentice Hall International.

HOUSE OF REPRESENTATIVES, (1989) Bugs in the program problems in federal government computer software development and regulation, Technical Report 052–070–06604–1, Subcommittee and Oversight of the House of Representatives Committee on Science, Space, and Technology, Superintendent of Documents; Government Printing Office, Washington, D.C., 20402.

JONES C. B., (1990) *Systematic Software Development using VDM*, Prentice–Hall International.

KOEN B., (1985) 'Definition of the engineering methods', American Society for Engineering Education.

PRESS N. A., (1990) 'Computer Science and Technology Board Report: Scaling Up: A Research Agenda for Software Engineering', *Communications of the ACM* 33(3), 281–293. Excerpted.

PRESSMAN R., (1992) *Software Engineering: A Practitioners' Approach*, 3rd Edition, McGraw–Hill.

SAIEDIAN H., (1992) 'Mathematics of computing', *Journal of Computer Science Education* 3(3), 203–221.

SAIEDIAN H. & HINCHEY M., (1994) 'Transferring the formal methods technology into the workplace', Submitted for publication.

WEBER–WULFF D., (1993) Selling formal methods to industry, in 'Formal Methods 93', *LNCS 670*, Springer–Verlag, pp. 671–679.

WOODCOCK J. & *Larsen P.*, eds (1993) FME'93: Industrial Strength Formal Methods, *LNCS 670*, Springer–Verlag.

Received: June, 1994
Accepted: November, 1994

Contact address:
Department of Computer Science
University of Nebraska
Omaha, NE 68182 U.S.A.

HOSSEIN SAIEDIAN is an assistant professor in the Department of Computer Science at the University of Nebraska at Omaha. Dr. Saiedian received his PhD degree from Kansas State University in 1989. He is a member of the IEEE Computer Society, Sigma Xi, the ACM, and curently serves as the Chair of the ACM SIGICE (Special Interest Group in Individual Computing Enviroments). Dr. Saiedian research articles have been accepted for publications in *IEEE Computer, Computer Networks* and *ISDN Systems, Journal of Systems ans Software, Journal of Information and Software technology, Office Systems Research Journal, Journal of Microcomputer Applications* and many others. His research interests include sowtware engineering models, formal methods and object-oriented computing.