N89-16307 S28-61 167052 10P

Recent Trends Related to the Use of Formal Methods in Software Engineering

Søren Prehn
Dansk Datamatik Center
Lundtoftevej 1C
DK-2800 Lyngby (Copenhagen)
Denmark

Abstract:

An account is given of some recent developments and trends related to the development and use of formal methods in software engineering. The paper focuses on ongoing activities in Europe, since there seems to be a notable difference in attitude towards industrial usage of formal methods in Europe and in the U.S.

A more detailed account is given of the currently most widespread formal method in Europe: the *Vienna Development Method*. A currently ongoing project, *RAISE*, aiming at developing a second generation formal method and related tools is described.

Finally, the use of Ada® is discussed in relation to the application of formal methods, and the potential for constructing Ada-specific tools based on such methods is considered.

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

1. Introduction and Background

It is well-known that the increasing use of software systems of an increasingly complex nature imposes greater requirements to the quality of software, its documentation and maintainability. It is also well-known that since the term "software crisis" emerged, little progress has actually been made in industrial software development environments towards meeting these requirements.

In this paper, we advocate the viewpoint that industrial software engineering today really is not *engineering*, and that real progress is to be sought in the maturation of present software production technology into a true engineering discipline.

It is believed that the characteristics of a true engineering discipline are twofold:

- the discipline must have a mathematical foundation

- the day-to-day practises of the discipline are not necessarily truly formal

This is to be understood in the following way. The requirement for a mathematical foundation is triggered by the desire to be able to *reason* about the objects created during software development (such as specifications, programs, and design decisions) in a way that allows one to determine whether any such reasoning is valid or not; in particular one would like to be able to reason about the *functional correctness* of a program with respect to a specification. On the other hand we believe, in particular when one considers industrial software development, that such formal reasoning will mainly take place in order to establish ("once and for all") general rules and techniques whose correctness and soundness are verifiable. On a day-to-day basis there is presently no hope that development of any but trivial (small) programs can be thoroughly reasoned about in a formal way: the combinatorial complexity is simply too high. Thus we advocate the daily use of rules and techniques whose formal correctness and soundness have previously been established.

This is well in accordance with the way established engineering disciplines work. For example, electronics engineering has a rather firm basis in mathematics (e.g.: the use of complex calculus to describe quasi-stationary circuitry) and makes heavy use of various formal notations (such as diagrams, being a language with a precise, mathematical meaning (and a graphical syntax)). In daily life, the electronics engineer goes about his job mainly on the basis of previously established design principles, without considering the formal proofs of their soundness. However, from time to time, it is necessary to bring in formality, to make mathematical analysis and conduct proofs. This typically happens when a completely new sort of circuitry is being considered, or when requirements to circuitry functionality and reliability are particularly strict.

Here it is worth noting that only the fact that electronics engineering has a mathematical basis makes this possible; it would not have worked to base daily practises on informal notions, and then bring in formality from time to time.

The analogy offers another interesting observation: there seems to be two different styles of work involved: one style is based on using sound development rules, another on formally analysing (e.g.: proving the correctness of) an otherwise constructed object (such as the design of an electronic circuitry). We shall return to this dichotomy.

It is not surprising that software development has not yet evolved into a true engineering discipline. The trade is relatively young, and the requirements to the (complexity of the) software systems to be produced are ever increasing. Mathematics and formality has, though, been successfully applied to various aspects of software development. The availability of BNF grammars and parser generators is the classical, convincing example.

The scene is, however, beginning to change. In Europe, information technology industry in general demonstrates a growing interest for formal specification and design languages, for formal development rules, and for formal verification techniques. This, we believe, is in contrast to the trends in U.S. information technology industries, where the emphasis appears to be on tools, workstations, and environments, rather than on the methods they should support.

The purpose of this paper is to outline current trends in Europe. Given the space available, it is impossible to give a complete and covering picture, let alone to go into much technical detail. It is hoped, however, that the material presented will stimulate discussions on introducing formal methods into industrial software engineering environments.

In section 2, an overall scenario is presented, and a number of relevant research and development projects are mentioned. In section 3, an account is given of the so-called Vienna Development Method (VDM), which was the first purportedly formal method to reach any industrial significance, despite its shortcomings. In section 4, an account is given of the RAISE project, whose explicit objective is to provide formal languages and techniques for software engineering (in the above sense) as well as support tools. Finally, in section 5, perspectives specifically concerned with Ada are discussed.

2. The European Scene

Although there has been some industrial interest in formal software development methods in the European information technology industry over the past decade, and even a few successful attempts to seriously apply such methods on "real" projects, formal software development methods have had no pervasive impact. There has been a distinct, and partially well-founded, belief that formal methods were not sufficiently industrialized. Also there has been an assumption that formal methods probably were not worthwhile to apply or even harmful.

However, formal methods are now beginning to come about in industrialized form, and it is becoming increasingly clear to industry that software development practises must be seriously improved if the potential and challenges offered by the continuous hardware technology evolution are to be met.

Also, European academe has a strong tradition for research in the formal methods area, and there is today a strong desire to transfer the acquired knowledge and expertise to industry.

Probably, the most visible evidence of this trend is the joint industrial and academe support of and participation in projects, concerned with formal methods, sponsored by the Commission of the European Communities (CEC). It is interesting to note that these projects typically involve cooperation between some four to six partners, industries as well as universities.

In order to give an idea of the range of activities and institutions involved we list a number of projects, totalling several hundred person years of effort, sponsored under the ESPRIT program [ESPRIT 86] (European Strategic Programme for Research and development in Information Technology). For each project, name, title, and participants are indicated:

FORMAST
  Formal Methods for Asynchronous Systems Technology
    Advanced System Architectures (United Kingdom)
    Erno (West Germany)
    Imperial College (United Kingdom)
    University of Kaiserlautern (West Germany)

GRASPIN
  Personal Workstation for Incremental Graphical Specification
  and Formal Implementation of Non-Sequential Systems
    GMD (West Germany)
    Olivetti (Italy)
    Siemens (West Germany)

PROSPECTRA
  Program Development by Specification and Transformation
    University of Bremen (West Germany)
    University of Saarland (West Germany)

Systeam KG (West Germany)
University of Dortmund (West Germany)
Syseca Logiciel (France)
University of Passau (West Germany)
University of Stratchclyde (United Kingdom)

RAISE
Rigorous Approach to Industrial Software Engineering
Dansk Datamatik Center (Denmark)
Standard Telephone and Cables (United Kingdom)
Nordic Brown Boveri (Denmark)
International Computers Limited (United Kingdom)

METEOR
An Integrated Formal Approach to Industrial Software Development
Philips (Netherlands)
CGE (France)
AT-T & Philips (Belgium)
Stichting Matematish Centrum (Netherlands)
COPS Europe (Ireland)
Tech. Software Telematica (Italy)
University of Passau (West Germany)

GENESIS
A General Environment for Formal Systems Development
Imperial Software Technology (United Kingdom)
Imperial College (United Kingdom)
Philips (Netherlands)

It is not within the scope of this paper to elaborate on the actual contents of the individual projects. However, section 4 describes one of the projects (RAISE) in more detail. Another major project that should be mentioned is the Munich CIP project carried out at the Technical University of Munich [Bauer 76, CIP 85].

In Europe, the interest in formal methods appears to concentrate more on formal specification and formal development than on verification. That is, there is a belief in the transformational programming paradigm: if an implementation is produced solely by applying a series of transformations, each of which are correctness-preserving, to an initial specification, the implementation will necessarily be correct with respect to the initial specification, thus eliminating the need for verification. The interest in this style of development is connected with two concerns: firstly, it tends to eliminate an early introduction of (design) errors, and secondly, recording the series of transformations applied produces invaluable documentation of the system design process.

3. The Vienna Development Method (VDM)

VDM originated in the IBM Vienna Laboratories in the early seventies and was developed in connection with a project aimed at developing a production quality PL/I compiler. The project group initially worked on giving a formal semantics for PL/I; this effort probably constitutes the first example of successfully applying formal techniques to a fairly large-scale problem in an industrial environment [Bekic 74].

During the late seventies, VDM was further developed, and an increasing number of development projects using VDM emerged. Areas in which VDM was applied comprised not only programming languages and compilers, but also databases, operating systems, hardware specification, business applications, etc.

[Bjørner 83] contains an overview of VDM basics and an extensive bibliography.

[Bjørner 82] contains numerous major examples of VDM specifications.

Today, there is a rather pervasive interest in VDM in Europe, as witnessed by the formation of "VDM Europe", an interest group sponsored by the CEC and drawing participants from a fairly substantial number of European industries and universities, and by the formation of an industrial panel in the United Kingdom working towards making the VDM specification language into a British Standard.

Technically, VDM is based on the techniques developed for giving denotational semantics of programming languages. A denotational semantics is given as a homomorphism from an algebra of syntactic objects to an algebra of semantic objects, or, somewhat simplified, maps pieces of syntax onto semantic objects such as state transformations (functions from states to states). The principle readily adapts to numerous applications: many systems may conveniently be characterised by a state, which is manipulated by operations. Names of operations and their arguments are then considered to be syntactic objects.

VDM is *model-oriented.* By this is meant that the objects (syntactic and semantic) are explicitly constructed in terms of given constructors such as sets, lists, maps, and functions. This is in contrast to *property-oriented* specification approaches, such as algrebraic specification approaches, where objects are defined implicitly by the equational rules for the operations that manipulates them.

It is strongly believed that this aspect of VDM has been crucial for larger applications, and for the acceptability of VDM in industrial environments: model-oriented specifications tend to appeal much more to software engineering intuition than does property-oriented specifications. On the other hand it also clear that a model-oriented specification methodology may easily be abused to produce very operational "specifications" and presents a prevalent danger of over-specification.

4. The RAISE Project

The RAISE project (Rigorous Approach to Industrial Software Engineering) is a 115 person-year effort undertaken by a consortium consisting of Dansk Datamatik Center and Nordic Brown Boveri (Denmark), and Standard Telephone and Cables p.l.c. and International Computers Limited (United Kingdom). The project is partially funded by the Commission of the European Communities under the ESPRIT programme, and is carried out in the period 1985 to 1989. An overview of the RAISE project is given in [Meiling 85].

The RAISE project will provide an environment consisting of

- a *wide spectrum language* in which one can express abstract, formal specifications, designs, and algorithms
- means for expressing and affecting *transformations* of such entities
- proof systems and techniques serving to verify the correctness of such transformations
- a comprehensive tool set

Also, the project has been designed to include production of educational, training and technology transfer material alongside with the development of the above.

In RAISE, *Rigorous* hints at the underlying dogma that, although the RAISE language is formally defined and in principle enables the user to proceed strictly formally in developing a software system, practical conditions and requirements force one to choose, pragmatically, to carry out various parts of a development with varying degrees of formality. The philosophy behind the design of the RAISE tool set is to facilitate such a working style rather than to force a user into unmanageable formality.

RAISE encourages development by application of correctness preserving transformations, and allows for the development and verification of such transformations. The choice of using a specifically designed wide spectrum language implies that most of a development can be carried out independently of any perspective implementation language: only a final step in a development will carry a detailed, operational design into code. Typically, the code of a software system will therefore not exploit all the bells and whistles of the implementation language; indeed, it is hoped that only rather well-behaved systems will then result.

In RAISE, *Industrial* hints not only at the above-mentioned pragmatic choices that should be catered for, but also at truly quality tools and man-machine interfaces, usability of methodologies for "real" software systems, including the ability to obtain efficient end-products. In order to ensure conformance with these requirements, the project has been designed to include a number of *industrial trials*, i.e. applications of (intermediate versions of) languages, methods and tools during the course of the project; such industrial trials are to take place in actual industrial projects not otherwise connected with RAISE.

5. Some Future Perspectives

At present, it is fair to say that the industrial use of formal methods in Europe is beginning to happen. There is, though, still a long way to go. The major obstacles we are facing are:

- insufficient matureness of formal methods
- lack of management awareness
- lack of educational material and capacity
- lack of tools

A number of projects have been mentioned which attempt to seriously work towards more mature formal methods, keeping the more pragmatic requirements to the potential for industrial usage in focus. These projects were designed to bring out the best of earlier formal methods, combined with the most recent advances in research. It is believed that the next 2 to 5 years will bring about radical progress.

By the term "management awareness" we primarily think about first and second level managers' willingness to allow or force formal methods to be introduced into projects and divisions. The present, rather widespread conservatism is well understandable: although a number of successful projects having employed formal methods can be identified, it is, in all fairness, characteristic for such projects that they have been carried out in particularly friendly environments. Will formal methods actually port to "real" industrial environments? The most important part of the answer, we believe, is reflected in our next concern.

Availability of educational material and sufficient well-qualified personnel to aid in the introduction of new technology are invariably a major concern in any situation of evolution, and indeed also for the introduction of formal methods. However, we beleive that availability of text books, workshops, and courses is not sufficient. It appears to be a general experience that the introduction of formal methods should happen (1) in connection with a real project, (2) be preceeded by intense education (not just training), and (3) -- crucially -- be supported by on-project consultancy provided by experienced practitioners.

For the moment, few tools supporting formal methods are available. So, basically experiences today have been painstakingly acquired using paper and pencil. And scepticists may reasonably ask whether one can have more confidence in formal specifications and designs not checked by tools than in programs not checked by a compiler. Nevertheless, projects based on 3 levels of paper-and-pencil description (specification, high-level and low-level designs) preceeding the implementation have proved to come up with rather startling net productivity figures and low error rates. With really good tools, we should be able to do even better. It is important to us, however, that method design, understanding and experience preceed the construction of tools.

The Perspective for Ada and Formal Methods

Ada is probably one of the most complicated programming languages ever designed. The complexity is clearly witnessed by the immense amount of resources that has been required to bring about a reasonably debugged reference manual, compilers, and so on.

The complexity mainly stems from the rather large number of language concepts and features and, in particular, their general interaction. An often-noted problem is, as an example, that concurrency (tasking) interfere with the semantics of otherwise well-understood constructs such as function calls in a rather non-transparent way: the effect of tasking is not clearly bound to the syntax of Ada. It is to be feared that the complexity of Ada may impart a serious threat on the ability to construct and maintain correct and reliable software systems. With the widespread acceptance of Ada as the preferred programming language for military and space applications it is more urgent than ever to be serious about true engineering techniques and tools that will enable industrial construction of correct and reliable software.

We believe that there are two (complementary) lines of development to be pursued: adoption of the transformational programming paradigm, and providing usable techniques and tools for analysis (including verification) of programs. These two lines will probably be effective at different points in time: although powerful transformational programming systems are currently being developed, it will invariably take some time before such systems come into widespread use -- hence there is an extremely urgent need for providing tools that can assist in analysing Ada programs having been produced by more traditional techniques.

If such tools are to be of an interesting quality they must be based on a formal understanding of Ada. It is hoped that the completion of the Draft Formal Definition of Ada [Hansen 86] will provide the necessary foundation.

6. References

[Bauer 76] F.L. Bauer: "*Programming as an Evolutionary Process*"; in: Lecture Notes in Computer Science, Vol. 46, Springer Verlag , 1976

[Bekic 74] H. Bekic et.al.: "*A Formal Definition of a PL/I Subset*"; IBM Vienna Laboratories TR25.139, December 1974

[Bjørner 82] D. Bjørner & C.B. Jones: "*Formal Specification and Software Development*"; Prentice-Hall International Series in Computer Science, 1982

[Bjørner 83] D. Bjørner & S. Prehn: "*Software Engineering Aspects of VDM*"; in: D. Ferrari et.al. (eds.): "Theory and Practice of Software Technology", North-Holland Publishing Company 1983

[CIP 85] F.L. Bauer et.al.: "*The Munich Project CIP - Volume I: The Wide Spectrum Language CIP-L*"; Lecture Notes in Computer Science, Vol. 183, Springer Verlag ,1985

[ESPRIT 86] "*ESPRIT Project Synopses, Software Technology*"; Commission of the European Communities, January 1986

[Hansen 86] K.W. Hansen: "*Structuring the Formal Definition of Ada*"; these proceedings

[Jones 80] C.B. Jones: "*Software Development - A Rigorous Approach*"; Prentice-Hall International Series in Computer Science, 1980

[Meiling 85] E. Meiling et.al.: "*RAISE Project: Fundamental Issues and Requirements*"; RAISE/DDC/EM1/v6, 1985-12-10; Dansk Datamatik Center, 1985