To appear in *IFIP Transactions A: Computer Science and Technology*

Formal or Informal, Practical or Impractical:

Towards Integrating Formal Methods with Informal Practices in Software Engineering Education*

T.H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam Road
Hong Kong

Abstract

Two conflicting schools of thought have been dominating software engineering education. One school stresses on the popular software development methodologies, but horror stories on poorly designed systems are not uncommon. The other school advocates formal methods, but most practitioners regard them as impractical. We recommend that we should bridge the gap between the formal and informal by bringing theory to existing practice. The formalism should be used as a working tool behind popular software development methodologies. Students should not be trained as craftsmen who consider software development as an art and learn only from past mistakes. Nor should they be trained as mathematicians who are more comfortable with theory than applications. Software engineers must be educated as real ''engineers'' who are competent with industrial practices as well as the mathematical foundation directly supporting them.

Keyword Codes: D.2, D.2.1, K.3.2

Keywords: Software Engineering, Formal Methods, Computer and Information Science Education

1. INTRODUCTION

Two conflicting schools of thought have dominated software engineering education [21]. One school stresses on the popular software development methodologies, such as structured analysis and design [8, 19, 29] and object-oriented analysis and design [3, 4, 1]. These methodologies are based on the experience of practitioners and consultants in the industry, and are supported by CASE tools with user-friendly graphical interfaces. However, horror stories on poorly designed systems using popular methodologies are not uncommon.

** Part of this research was done at the Programming Research Group of the University of Oxford under an ACU Visiting Fellowship. The research is also supported in part by a grant of the Research Grants Council.

The other school advocates formal methods, which help to specify unambiguously the exact requirements of software systems [9]. Examples are VDM [16] and Z [20, 28], which are based on abstract models; Larch [13, 27] and OBJ3 [10, 12], which are based on algebraic semantics; and CSP [7, 14] and CCS [17, 18], which model concurrency. Using correctness proofs, we can verify whether the systems have been implemented according to the specifications. Despite the rigour, however, success stories are few and far between, because formal methods are regarded as impractical and not accepted by software engineers in the industry [5]. Furthermore, even though commonly known as formal "methods", they are mostly stand-alone techniques rather than methodologies covering all the stages of software development.

In this paper, we look into the merits and dismerits of formal methods and recommend another alternative on software engineering education which may better serve the needs of the industry.

2. THE NEED FOR EDUCATION ON FORMAL METHODS

Software engineering is conventionally being taught as a craft. Students learn the techniques for supporting various phases of the software development life cycle, based on the recommendations and guidelines of experienced authors. These techniques are usually supported by CASE tools with user-friendly graphical interfaces, in the form of data flow diagrams, structure charts, entity-relationship diagrams, state-transition diagrams and/or client-server diagrams. They are be packaged under various "methodologies" such as structured analysis, structured design, object-oriented analysis, object-oriented design, information modelling, or combinations of the above. Each new methodology introduced is supposedly better than its predecessors. These methodologies are well accepted by educators and practising software engineers because:

(*a*) The straightforward guidelines make it easy for novices to learn their trade.

(*b*) They suggest partitioning complex systems into manageable units, such as objects or top-down hierarchies of modules.

(*c*) They are based on simple graphics, which can be understood by professionals as well as end-users.

(*d*) They are supported by user-friendly CASE tools, which help designers to toy with their proposals.

Despite the popularity and the user-acceptance of software engineering methodologies, horror stories are not uncommon. A latest example is the ambulance system in London, which resulted in unwarranted delays and unnecessary deaths on the first day of implementation. It was shelved indefinitely from the second day.

We may attribute the problems in software engineering to a mismatch between the scale of the target systems and the methodologies for deriving the solutions. Unlike craftsmen who make articles individually, the scale of production of software systems is similar to other engineering processes. A small error may mean the loss of millions of dollars or even human lives. The cost of correcting an error after implementation can be a thousand times that of correcting the same error at the time of specification [2]. It would be much more cost-effective if errors could be identified and removed at an earlier stage.

Furthermore, we usually verify the correctness of an implementation using test data. Such data can, however, only show the presence of errors and not the absence of problems. The size and complexity of systems are increasing considerably in recent years. The reliance on test data alone for the correctness of a system have become unrealistic.

In other engineering disciplines, such as in mechanical or structural engineering, the engineers can provide users with a guaranteed degree of confidence by supporting their practice with theory, and hence ensure that the product is safe and free from errors. On the other hand, there is no theory in the popular methodologies which enables software engineers to prove the acceptability of a system. We must support practice by formal theory, so that the correctness of implementation can be proved and verified against user requirements.

The advantages of using formal methods include the following:

(*a*) The specifications will not be ambiguous or inconsistent. It can be verified to be syntactically correct and semantically consistent. It becomes a better contract among users, designers and implementors. Misunderstandings can be avoided.

(*b*) The time to code the system can be reduced despite a slightly longer time for specification.

(*c*) The implementation can be proved mathematically to be correct. Using formal derivations, we can predict the behaviour of the programs independently of how they are coded, and verify that the implementation indeed agrees with what is required.

(*d*) There is a guaranteed standard in the systems thus developed. This is particularly essential to systems which are safety-critical, or where heavy financial losses would result from errors.

(*e*) It will be easier to make future enhancements for the systems.

3. PROBLEMS WITH FORMAL METHODS

On the other hand, even though outcries have been made to popularize the formal methods, they have largely remained proposals from the academia. Practicing software engineers are not too eager to apply them. Students trained in formalism often find themselves having to learn and use informal methodologies after graduation. This may be attributed to the following:

(*a*) A specification is not only the basis for the implementation, testing and maintenance. It also serves many other purposes in the software development life cycle. It is a means of communication among users, designers and implementors. It helps people to arrive at the requirements of the target system. It enables users to visualize the proposal and compare it with their actual needs. For example, the popular methodologies are mostly based on graphical documentation, which are more suitable for giving users an intuitive idea of complex systems. It is in two dimensions, may be colour-coded and may consist of multiple levels. Hence it provides presenters with additional degrees of freedom. Formal methods do not support these functions very well. On the contrary, a formal specification consists of a substantial amount of jargon, which have to be translated verbally by the designer into laymen terms before it could be understood by end users. This is in sharp contrast to other engineering disciplines where blueprints presented to users consist of diagrams instead of differential equations.

(*b*) A user of a large complex system would not be familiar with all of its operations but only interested in an overview plus details of some selected aspects. Thus it is more natural to approach a system in a top-down manner, starting from a high level of abstraction and then filling in the details of the lower levels. This concept is strongly advocated by the popular methodologies. Most formal methods, however, require us to define the details at the beginning and grow a specification bottom-up.

(*c*) Most practising software engineers are not trained in formal methods, and may be overwhelmed simply by the thought of having to use mathematical notations. Experienced people are not willing to try their hands on new methodologies, especially if they involve formal jargon, for fear of losing out to young graduates.

(*d*) Practitioners find that quite a number of academics who advocate on formal methods are unfamiliar with existing methodologies and do not have much experience in the real world.* Many of the applications, until very recently, have been restricted to small academic problems such as stacks and queues. The most frequently quoted success story is still the application of Z in re-engineering CICS [6, 15], which reduced the development cost by 9% and the number of errors by 60%, and resulted in a Queen's Award for Technical Achievement for Oxford University Computing Laboratory and IBM Huxley.

(*e*) Most formal methods exist primarily as a technique which supports a formal notation, but not a development methodology for the full software development life cycle. Most formal tools are not linked up with an existing CASE environment.

(*f*) From the point of view of IT management, a methodology which is popular in undergraduate teaching and well tested in real-life applications would be more acceptable than unproven methods.

4. RECOMMENDATIONS

Instead of debating on which school to follow, we must recognize that neither formal nor informal methods are a complete solution to the problems in software development. We should compare ourselves with other engineering professions. Electrical engineers, for example, would not be satisfied with a design which is based entirely on circuit diagrams without supporting calculations. Nor would they present differential equations to users for verification. We recommend that we should bridge the gap between the formal and informal. The two camps are complementary to each other rather than in conflict.

Software engineering is indeed an engineering process. A specification is a model of the solution for the real world. Possible models must be analysed and evaluated until the most suitable is found. Graphics and mathematics are used because they are found to be more suitable for manipulation than English text. But neither of them may supersede the other. A diagram helps us with intuitive reasoning, as well as for presentation to users. The mathematical counterpart helps to prove our intuitive ideas.

Thus it is impractical to define a specification in terms of only a graphical or formal representation. We need a specification in more than one form. One representation must be convertible into the other so that users, designers and implementors can communicate in the most effective manner. A person would refer to the version appropriate to his background and needs. There must be a formal link between the two representations to avoid possible errors.

This cannot be achieved simply by adding adding "syntactic sugar" [11] to the formal specification, such as replacing mathematical symbols by English-like phrases. Nor should we be proposing *ad hoc* user-friendly graphical interfaces to formal specifications, with the hope that they will successfully replace the existing popular graphics.

The graphical notations in popular methodologies have proven track records of acceptance and practicality. They serve as excellent means of communicating ideas and as blueprints. The main

---

* I recall making a presentation on the application of formal methods using a realistic banking example. An academic in the audience quiped, "it looks like Cobol to me".

drawback lies in the lack of a mathematical rigour. We should train software engineers to use the popular methodologies, but at the same time, add formalism behind them. In this way, formal techniques and informal methodologies can be brought together.

Our experience at the Software Engineering Group, Department of Computer Science, The University of Hong Kong, have shown that this is indeed feasible. We have been successful in conducting a number of projects in bridging the gap between formal methods and popular software development methodologies. We have joint projects with the University of Oxford, York University, University of Melbourne and Jinan University. We have been awarded fundings of about $2,500,000 from various sources such as the Research Grants Council, the University and Polytechnic Grants Committee, the Science and Engineering Research Council, the Association of Commonwealth Universies, the Committee on International Cooperation in Higher Education, the International Conference on Information Systems, and the Hong Kong and China Gas Research Fund. More than 50 refereed international publications have been produced.

The following is a list of our projects on this aspect:

ACRONYM	**PROJECT TITLE**	**BRIDGING**	
		FORMAL METHOD	**POPULAR METHOD**
COD	a Communicating Objects Design Model (joint work with Dr Jeremy Jacob of University of Oxford and York University)	Process algebra (CSP)	Object-oriented analysis and design
FOOD	Functional Object-Oriented Design [25] (joint work with Professor Joseph Goguen of University of Oxford)	Functional programming (FOOPS)	Object-oriented analysis and design
NOODLE	a Net-based Object-Oriented DeveLopment Environment [24]	Petri nets	Object-oriented analysis and design
ALPSE	Application of Logic Programming to Software Engineering [23] (joint work with Dr T.Y. Chen of University of Melbourne and Professor H.Y. Chen of Jinan University)	Logic programming	Structured analysis and design
FDFD	Formal Data Flow Diagrams [26]	Petri nets	Structured analysis and design
SAD_FACT	a Structured Analysis and Design Framework using Algebraic semantics and Category Theory [22]	Initial algebra and category theory	Structured analysis and design

In order to support the above recommendation, we must provide a comprehensive software engineering education to students, which should include the following aspects:

(*a*) A software engineer must be trained in mathematics, mainly in the *appreciation* of formalisms such as abstract algebra, and the *application* of mathematical results to software development.

(*b*) He must be proficient in the popular analysis and design methodologies such as the structured methods and object-oriented methods. Emphasis should be made on the joint application of formalism and established methodologies to the solution of practical situation.

(*c*) He must be trained in other aspects of computer software and hardware, to design the logical and physical aspects of the target system and to communicate with programmers and database administrators.

(*d*) He must be trained to communicate with users at various seniority levels, both to elicit user requirements and to present his proposals.

(*e*) He must be trained in psychology and sociology, in order to understand the real user requirements and appreciate the resistance in the introduction of new systems.

(*f*) He must be trained in business administration, in order to appreciate the financial and other impacts of the new system, and the management objectives behind.

(*g*) He must be trained in management science so that the project is developed effectively and managed with no hiccups. He must maintain the planned course of action despite setbacks, but must be adaptive to change if necessary.

Software engineering programmes should be designed to educate the students on the long term as a professional, rather than limiting to short-term practices. Students should not be trained as craftsmen who regard software development as an art and learn only from experience. Neither should they be trained as pure mathematicians. Students should be required to take fundamental courses in all the areas considered essential to the profession, and are allowed to concentrate in one or more specialized fields in their later years.

Emphasis should be made on applications by requiring students to take courses outside of software engineering or even computer science. These courses could be in a wide variety of areas such as business administration, economics, psychology, sociology, and electronic and electrical engineering. Realistic projects, done individually or in groups, should be undertaken by all students. These projects help students to consolidate the theories and methodologies and to apply them in practical situations.

Our philosophy is being applied to the design of the Software Engineering curriculum at the Department of Computer Science, the University of Hong Kong as from 1993. The progress and feedback from the both staff and students will be closely monitored.

5. CONCLUSION

We recommend that we should bridge the gap between the formal and informal by bringing theory to existing practice. We should train software engineers to use the popular methodologies, but at the same time, add formalism behind them. The popular graphical notations are extremely useful for conceptual development and as blueprints to users. These blueprints must be supported by mathematical rigour. The mathematics should be regarded as a tool for software development. It must be visualized as a slave rather than a master.

In this way, formal techniques and informal methodologies can go peacefully hand in hand in the software engineering curriculum. Students are not trained as craftsmen who regard software development as an art and learn only from past mistakes. Neither are they trained as mathematicians who are more comfortable with fundamentals than applications. Software engineers are educated as real ‘‘engineers’’, who are competent with industrial practices as well as the mathematical theory directly supporting them.

REFERENCES

[1] M. Blaha and W. Premerlani, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, Englewood Cliffs, New Jersey (1998).

[2] B.W. Boehm, ''Seven basic principles of software engineering'', *Journal of Systems and Software* **3**: 3−24 (1983).

[3] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, California (1994).

[4] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, New Jersey (1991).

[5] D. Coleman, ''The technology transfer of formal methods: what's going wrong'', Position statement, Workshop on Industrial Experience Using Formal Methods, Nice, France (1990).

[6] B.P. Collins, J.E. Nicholls, and I.H. Sorensen, ''Introducing formal methods: the CICS experience with Z'', Technical Report TR-12.260, IBM United Kingdom Laboratories, Hursley Park, UK (1987).

[7] J. Davies and S.A. Schneider, ''A brief history of timed CSP'', Programming Research Group, Oxford University Computing Laboratory, Oxford (1992).

[8] E. Downs, P. Clare, and I. Coe, *Structured Systems Analysis and Design Method: Application and Context*, Prentice Hall, Hemel Hempstead, Hertfordshire, UK (1992).

[9] G.A. Ford, ''SEI report on graduate software engineering education'', Technical Report CMU/SEI-91-TR-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania (1991).

[10] J.A. Goguen, C. Kirchner, H. Kirchner, A. Megrelis, and J. Meseguer, ''An introduction to OBJ3'', in *Conditional Term Rewriting Systems: Proceedings of the 1st Workshop*, S. Kaplan and J.-P. Jouannaud (eds.), Lecture Notes in Computer Science, vol. 308, Springer, Berlin, pp. 258−263 (1988).

[11] J.A. Goguen and J. Meseguer, ''Unifying functional, object-oriented, and relational programming with logical semantics'', in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (eds.), MIT Press, Cambridge, Massachusetts, pp. 417−477 (1987).

[12] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, ''Introducing OBJ'', in *Software Engineering with OBJ: Algebraic Specification in Action*, J.A. Goguen and G. Malcolm (eds.), Kluwer Academic Publishers, Boston, Massachusetts (2000). The paper is also available at ''http://www-cse.ucsd.edu/users/goguen/ps/iobj.ps.gz/''.

[13] J.V. Guttag, J.J. Horning, and J.M. Wing, ''The Larch family of specification languages'', *IEEE Software* **2** (5): 24−36 (1985).

[14] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall International Series in Computer Science, Prentice Hall, London (1985).

[15] I. Houston and S. King, ''CICS project report: experiences and results from the use of Z'', in *VDM '91: Formal Software Development Methods: Proceedings of the 4th International Symposium of VDM Europe, vol. 1*, S. Prehn and W.J. Toetenel (eds.), Lecture Notes in Computer Science, vol. 551, Springer, Berlin (1991).

[16] C.B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International Series in Computer Science, Prentice Hall, Hemel Hempstead, Hertfordshire, UK (1990).

[17] R. Milner, ''A calculus of communicating systems'', Report ECS-LFCS-86-7, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh,

Edinburgh (1986).

[18] R. Milner, *Communication and Concurrency*, Prentice Hall International Series in Computer Science, Prentice Hall, Hemel Hempstead, Hertfordshire, UK (1989).

[19] M. Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, New Jersey (1988).

[20] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*, Prentice Hall International Series in Computer Science, Prentice Hall, Hemel Hempstead, Hertfordshire, UK (1991).

[21] M. Shaw and J.E. Tomayko, ''Models for undergraduate project courses in software engineering'', in *Proceedings of the 5th SEI Conference on Software Engineering Education* (*CSEE '91*), J.E. Tomayko (ed.), Lecture Notes in Computer Science, vol. 536, Springer, Berlin, pp. 33−71 (1991).

[22] T.H. Tse, *A Unifying Framework for Structured Analysis and Design Models: an Approach Using Initial Algebra Semantics and Category Theory*, Cambridge Tracts in Theoretical Computer Science, vol. 11, Cambridge University Press, Cambridge (1991).

[23] T.H. Tse, T.Y. Chen, and C.S. Kwok, ''The use of Prolog in the modelling and evaluation of structure charts'', *Information and Software Technology* **36** (1): 23−33 (1994).

[24] T.H. Tse and C.P. Cheng, ''Towards a 3-dimensional net-based object-oriented development environment (NOODLE)'', *The 5th International Congress on Computational and Applied Mathematics* (*ICCAM '92*), Katholieke Universiteit Leuven, Leuven, Belgium (1992).

[25] T.H. Tse and J.A. Goguen, ''Functional object-oriented design (FOOD)'', in *Foundations of Information Systems Specification and Design*, Dagstuhl Seminar Report No. 35, H.-D. Ehrich, A. Sernadas, and J.A. Goguen (eds.), International Conference and Research Center for Computer Science, Wadern, Germany (1992).

[26] T.H. Tse and L. Pong, ''Towards a formal foundation for DeMarco data flow diagrams'', *The Computer Journal* **32** (1): 1−12 (1989).

[27] J.M. Wing, ''Writing Larch interface language specifications'', *ACM Transactions on Programming Languages and Systems* **9** (1): 1−24 (1987).

[28] J.C.P. Woodcock, *Using Standard Z: Specification, Proof and Refinement*, Prentice Hall International Series in Computer Science, Prentice Hall, Hemel Hempstead, Hertfordshire, UK (1994).

[29] E. Yourdon, *Modern Structured Analysis*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, New Jersey (1989).