Pet Formalisms versus Industry-Proven Survivors: Issues on Formal Methods Education

S.D. Cha

Department of Computer Science and Advanced Information Technology Research Centre (AITrc)
Korea Advanced Institute of Science and Technology (KAIST)
373–1 Kusong-dong, Yusong-gu
Taejon, Korea
Phone: 82-42-869-3535
Fax: 82-42-869-3510
Email: cha@cs.kaist.ac.kr

Formal methods are gaining steady industrial acceptance as a promising approach to developing high-quality software. Although educators find it increasingly important to include formal methods in their software engineering curriculum, they face a dilemma: there are too large numbers and diverse flavors of formal methods to cover in a limited time, and little empirical data exists on relative strengths and weaknesses among them. How can we best develop a curriculum on formal methods given several practical constraints? In this paper, we describe our approaches, share experiences, and evaluate results. Our goal is to stimulate further discussion among software engineering education communities on how formal methods can be most effectively introduced to current and next generation software engineers.

Keywords: software engineering education, formal methods

1. INTRODUCTION

As the use of safety-critical systems in industrialised nations continues to grow, so does our society's dependency on high assurance software systems. There are some safety-critical applications where the most serious software mishaps must be avoided at all practical cost (Leveson, 1995). Examples include the fly-by-wire commercial jets, nuclear power plant shutdown systems, patient monitoring systems, and radiation therapy machines.

Formal methods are generally defined as techniques based on mathematical principles for the specification, development and verification of software and hardware systems. Although once regarded as a subject of research interest only to academics and considered impractical for industrial applications, formal methods are surely gaining slow but steady acceptance as a promising approach to overcome the quality ceiling problem commonly associated with traditional software development methods (NASA, 1995).

Formal methods are not, and should not be regarded as, "silver bullets" (Dean and Hinchey, 1996), and several issues, including their cost-effectiveness, are still being debated. However, widely accepted consensus among experts is that proper application of formal methods can increase

Manuscript received: 26 October 1999

quality of software and that conventional software engineering curricula (e.g. processes, design methodologies, object-orientation, and testing) are inadequate in providing sufficient technical expertise needed to develop the high quality software some safety-critical applications demand. Therefore, there is an urgent need to train current and next generations of software engineers not only on fundamental principles of software engineering but also on technical details of formal methods.

When developing a curriculum on formal methods, the initial decision to make is if the course will cover only underlying mathematical principles or details of specific formalisms. The former subject is often covered in courses such as discrete mathematics and computational logic, and most undergraduate computer science programs, including KAIST, teach them in required or elective core courses. Since knowledge of mathematical principles alone is insufficient to prepare students for the challenge of applying formal methods, we have decided to develop a formal method curriculum designed to provide students with hands-on experience and offer this initially to students enrolled in a postgraduate (e.g. masters) program.

Unfortunately, development of an effective curriculum on formal methods is a challenging task, and there are several practical and important issues to address. First is the time constraint. While integration of software engineering and formal methods curriculum would be clearly desirable, as advocated by Garlan (Foster and Barnett, 1996) and practiced in the CMU's MS program in software engineering, most universities will most likely be limited to offer a course lasting a semester or at most two. Furthermore, seamless integration of formal methods into the software development process still remains as a research topic. Another constraint is the number of academic credits associated with the course since it effectively determines the number of hours students can be reasonably expected to work. Finally, there are too large numbers and diverse flavours of formal methods proposed in the literature and each comes with various claims of advantages and strengths. For example, http://www. comlab.ox.ac.uk/archive/formal-methods.html, arguably the best-known site on the subject, list more than 70 different notations, methods, and tools for formal methods, and the list is certainly incomplete. In fact, Statecharts (Harel, 1987), a popular modelling language used to specify behaviours of reactive systems, is probably the most noteworthy omission from the list. Furthermore, little empirical data exists on relative strengths and weaknesses of various formal methods.

2. OUR APPROACH

Our curriculum development effort started with the formulation of a set of key questions related to course objectives and strategies. Our course was primarily intended for graduate students majoring in computer science who are likely to work in industry initially as programmers or research staff and later as project managers as their career advances. Like many universities, formal languages and discrete mathematics are part of required core courses in our undergraduate program, and our students are familiar with mathematical principles. Therefore, the primary course objective was to train our students with a sufficient amount of practical knowledge of formal methods so that they can readily apply formal methods themselves on industrial projects should the needs arise.

As for strategies of teaching formal methods, we decided to make extensive laboratory sessions an integral component of our curriculum. Our decision is based on the belief, shared by Dean and Hinchey, (1996) and by Foster and Barnett, (1996) that modelling skills are best learned through practice and that learning a formal method is more difficult and time consuming than learning a programming language. That is, many and diverse flavours of programming languages (e.g. ML, Prolog, C, Ada95, etc) can be successfully taught in a course for students to experience different programming paradigms. Although students' experiences would necessarily be limited to programs of small size, students can learn enough about syntax, semantics, and programming paradigms so that they can start developing larger size programs by themselves if they are allowed to consult reference materials. Our experience indicates that up to four or five different programming languages can be taught in a semester. However, becoming a "trained novice" on formal methods requires more training overhead than typical high-level programming languages do. For example, N. Shankar, a developer of PVS, estimates that learning PVS in sufficient detail to develop specifications for non-trivial problems and conduct proof requires one to six months of full-time (e.g. 40 work hours a week) usage. Therefore, we decided to carefully select two or three different formalisms and teach them in 16-week long semesters. As for the course projects, we selected a couple of small scale but realistic examples such as a controller for coin-operated coffee vending machines (CVM), the steam boiler problem (Abrial *et al*, 1996), and the water-level monitoring system. The latter two problems have been studied by various groups working on formal methods, and we felt that our experiences could be compared to the published ones so that students might obtain additional insights.

Selecting which formalisms to teach was an important but difficult decision to make since there were too many formal methods to choose from. We decided to narrow down our choices using the so-called "happy customer" test. That is, formal methods failing to meet the following criteria are excluded from further consideration:

1. It must have been successfully applied on medium to large scale and well-documented industrial projects;
2. It must be supported by stable or production-quality, as opposed to research prototypes, software tools; and
3. The tool must be supported by well-documented and comprehensive tutorial material.

We could have employed additional or different criteria. One possibility, referred to as the pet formalism approach in this paper, is to select the formalisms most favoured by the instructor. Such selection could be made based on diverse criteria. For example, it may come with an exceptionally powerful verification system. Perhaps it can represent asynchronous system behaviour well. Perhaps it is simple enough to cover in limited time. Another possibility is to categorise existing formal methods and to teach a representative sample from each group. For example, we could have selected one from each of process algebra-based, high order logic-based and finite state machine-based formal methods. An advantage of this approach is that students can gain familiarity with diverse formal methods and appreciate why various classes of formal methods have been proposed.

Although there would be certain educational merits associated with other approaches, we strongly believed that with those approaches students were unlikely to be best prepared for industrial application of formal methods. The initial survey indicated that there were quite a few industrial-strength formal methods that met our criteria mentioned above, and they included Petri Nets, Statecharts, B-methods, Boyer-Moore, RAISE, and PVS. Availability of either public domain or affordable CASE tools served as another practical constraint due to budgetary limitations. Formal methods that passed the initial screening still exhibited quite different flavours and are typically used in different phases of software development lifecycle. In order to compare solutions developed for the same problem using different formal methods, we have decided to require Coloured Petri Net (CPN) (Jensen, 1992; Jensen 1994) and Statecharts (Harel, 1987) as mandatory formalisms to study in depth. However, students were allowed and encouraged to study any formalism they were particularly interested in as alternatives provided that they met the same criteria. For example, a team of students seriously considered using PVS.

Our course consisted of lecture hours and laboratory sessions. The instructor made a deliberate effort to stimulate active discussion among all the participants. That is, conventional lectures in which students passively listened to presentations are minimised. Instead, students were required to critically review two or three assigned papers every week and to submit reading logs. It was strongly emphasised that reading logs must not simply summarise the papers. Instead, they were requested to list issues worthy of technical discussion in class so that the instructor may review them and distribute "discussion agenda" in advance. (The list of papers used in classroom discussion is available at http://salmosa.kaist.ac.kr/LAB/ MEMBER/CHA/COURSE/CS750/cs750-96f.html) Beginning with introductory papers on formal methods (Saidian, 1996), the first half of the semester was focused on papers that specifically dealt with CPN and Statecharts on topics such as overview, analysis methods, formal semantics, and industrial experiences. During the second half of the semester, Hinchey and Bowen, (1995) was used as the primary textbook so that students could learn more about the experience of industrial application of formal methods other than CPN or Statecharts.

3. MODELLING EXPERIENCES

Course projects were conducted by establishing teams of three or four students to facilitate effective learning and cooperation. When teams completed projects developing CPN and Statecharts models using the CASE tools, Design/CPN and STATEMATE, respectively, a couple of class sessions were used to present and compare diverse solutions each team developed. Students were given about three to four weeks to complete each modelling project. Two to three weeks were needed to become familiar with details of the CASE tools used, and developing and debugging models took, on the average, about 10 to 15 hours working as a team.

When carrying out course projects, students were allowed to select other problems if they wished provided that they were of similar complexity to the recommended problem set. However, most teams developed models for the recommended problem sets (see appendix A for a brief description and a simplified solution given in Statecharts). Several teams developed specifications for a traffic light controller, simplified version of a nuclear power plant shutdown system, or RISC processor called the DLX. In the next section, we describe the approaches taken in developing CPN and Statecharts models and how sufficiently the models were analysed.

Coloured Petri Nets

CPN, a popular high-level Petri net formalism, extends classical P/T Nets with advanced features such as token colours, substitution transition, and fusion places. While all tokens used in P/T Nets are of Boolean types, CPN tokens may have arbitrarily complex data types associated with them. Colour definition, a feature similar to the type definition in high-level programming languages, significantly improves expressiveness of the CPN formalism while maintaining analysis capability equivalent to that of P/T Nets. That is, any CPN model can be mechanically translated into behaviourally equivalent P/T Net models, and all types of analysis applicable on P/T Net can also be applied on the CPN models. Furthermore, CPN provides powerful features to enable hierarchical and modular development of models for complex systems. Using concepts of substitution transitions, one can organise CPN models hierarchically. When a transition is declared as being a substitution transition, input and output places associated with the "super transition" are considered input and output ports, respectively, and the behaviour of substitution transition is further expanded on another page. Students reported that hierarchy and modularity features combined with Standard ML declarations were useful in developing models that were easily understandable and extendible.

In addition to features supporting hierarchy and modularity, CPN also provides a strong type checking capability. CPN uses a functional programming language called the Standard ML to declare token types, to express versatile arc expressions, and to denote various computations associated with transitions. That is, a CPN model can be “compiled” into a behaviourally equivalent Standard ML program, data types checked, and model simulation performed.

Several students noted that strong type checking was useful in detecting trivial modelling errors and that rigorous type checking made a significant contribution in developing high-quality CPN models. (On the other hand, STATEMATE does not provide such checks. For example, typos in event declarations are interpreted as valid declarations of separate and distinct events.)

Unfortunately, our students encountered some difficulties partly due to incomplete understanding of the formalism and to shortcomings of the Design/CPN CASE tool.

- Timing analysis was difficult to perform. For example, the steam boiler problem has a requirement that each cycle (e.g. receiving messages from the physical unit, processing them, and sending messages back to the physical unit) must be completed within five seconds. It seems that the timing specification features of the CPN, based on timestamps associated with tokens, were not powerful and flexible enough to express and validate real-time aspects of the system.
- The tool could provide only limited assurance that the model satisfies the functional requirements expressed in natural language. Although the Design/CPN includes a capability to generate an occurrence graph (also known as the reachability graph) for a given model, the majority of our teams failed to generate the occurrence graph on our workstation equipped with 128 MB of main memory. More importantly, even when the complete occurrence graphs could be generated in a brute-force manner, they were not useful because students were quickly overwhelmed by the sheer amount of low-level and raw information generated. It appears that more research and development is needed to enhance user interfaces and to provide only the relevant information the user is particularly interested in. As an alternative to generating occurrence graphs, one can perform interactive step-by-step execution of the model. While interactive simulation can be an effective method to demonstrate that the model works properly on a typical scenario, it has serious shortcomings as a means of demonstrating that the system is free from exhibiting undesirable and unsafe behaviour under rare and exceptional cases. That is, interactive simulation of the CPN model has the same fundamental limitation that software testing has.
- Students encountered some difficulties due to specific ways the Design/CPN was implemented. For example, only one global declaration can be declared, and fusion places, places with the same name appearing on different pages, cannot be declared as ports. Such limitations occasionally forced rather artificial deviation from the original “intuitively appealing” models.

Nonetheless, the majority of our students felt that the CPN formalism and that the Design/CPN was useful. Graphical representation, hierarchy and modularity features, use of Standard ML, and strong type checking capabilities were key factors responsible for a positive experience.

Statecharts

Statecharts (Harel, 1987) are gaining popularity as a specification language for reactive systems. Statecharts extend the classical finite state machine in terms of hierarchy (OR state), concurrency

(AND state), and broadcasting communication. Various research groups have worked on either extending Statecharts or defining operational semantics. According to one report, more than 20 variations of Statecharts semantics have been proposed in literature (Beeck, 1994). However, STATEMATE is the only commercial CASE tool supporting Statecharts modelling and analysis. STATEMATE offers a rich variety of tools in addition to the basic Statecharts editor and simulator. For example, panel editor enables rapid prototypes to be developed by using various icons and by associating specific events to icons. Furthermore, a code generation tool can produce executable C or Ada skeleton programs. In addition, STATEMATE offers interactive or batch simulation capability. The latter feature, simulation scenario programming language, is not found in the Design/CPN.

Statecharts are generally considered to be easy to understand not only to computer scientists but also to application experts (e.g. non-computer scientists). Leveson *et al*, (1994) is a well-publicised example on how Statecharts, actually a variant of Statecharts called RSML, was successfully applied to specify behaviour of complex avionics equipment called TCAS II. Our experience is consistent with that reported in Leveson *et al,* (1994) in that students found little problem in understanding Statecharts and simulating behaviour interactively.

Despite generally positive experiences reported by our students, there were several “wish lists” that became apparent while using the STATEMATE. First, many felt that the interactive simulation capability provided by the STATEMATE was inadequate to test the model sufficiently. This observation is similar to the one made regarding Design/CPN in that exceptional cases were difficult to simulate, and assurance that the model will work properly under exceptional cases was difficult to deliver. Another difficulty was due to conceptual conflict between well-known global broadcasting communication mechanisms and the STATEMATE feature to declare scopes for variables and events.

4. CONCLUSIONS

Formal methods is a broad, diverse, and interdisciplinary subject. Characteristics of effective formal methods curricula vary depending upon several factors such as student backgrounds, intended application domains, and most importantly the course objectives. There are temporal (e.g. quarter or semester) and financial (e.g. funds available to purchase necessary CASE tools) constraints as well.

Since there are no widely accepted metrics to quantify superiority of one curriculum over another, we make no claims that our formal methods curriculum is superior to others. However, an informal survey taken among enrolled students indicated that the course was effective in accomplishing the objectives. The following contributed to the success: decision to cover only a small number of industrial-strength formal methods in depth and course projects which provided students with experience in developing solutions to the same problem set using different formal methods.

Students felt that they were able to better understand system behaviours, especially the subtle ones. In other words, formal methods served, as intended, as excellent “discovery” tools. In fact, a couple of students successfully used STATEMATE the following year to specify reactive behaviour of an industrial traffic light controller whose size and complexity is substantially greater than the problems the course dealt with. For example, whereas the Statecharts models developed as a course project usually consisted of one or two pages and less than 30 states, the latter example consisted of 19 hierarchically organised sets of pages and 57 pages including data dictionary definitions.

Despite a generally favourable experience, enhancements seem possible to provide a more effective learning environment and to more accurately measure effectiveness. Our experience confirmed that learning formal methods is a difficult and time-consuming experience. Some students initially felt lost because they did not know how to start developing models. Existing guidelines largely deal with layout issues. For example, Jensen (1992) suggests that models be organised so that inputs generally appear on the left or top portion of the diagram and outputs on the bottom or right portion. Such a placement would enable top-down and left-to-right reading of the model. Another guideline is to avoid overlapping arcs whenever possible. While such "guidelines" might be useful in better organising the model already developed, they provide little help on how to start modelling activities.

Our students could have learned formal methods more effectively had there been practical modelling guidelines and comprehensive tutorial material in which several pre-developed models were included. Such models need not necessarily be polished in content and presentation. Models that span both sides of the quality spectrum can serve as equally effective educational material. Unfortunately, being the first time the course was offered, our students did not have any models they could consult. A book on the steam boiler problem and the associated CD-ROM (Abrial *et al*, 1996) is a useful addition, and we plan to further polish and eventually share the CPN and Statecharts models developed in our course via Internet.

Measurement of educational effectiveness is another area in which the curriculum needs improvement. This paper is based on an informal interview conducted at the end of the semester. Although the details are yet to be worked out, a questionnaire needs to be developed and more systematic surveys could to be conducted at the beginning and end of the semester to find out how effectively we have met the challenge.

ACKNOWLEDGEMENT

This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

REFERENCES

ABRIAL, J.-R., BORGER, E., and LANGMAAK, H., eds. (1996): *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, LNCS 1165, Springer-Verlag.

DEAN, C.N. and HINCHEY, M.G. (1996): Formal Methods and Modelling in context, in DEAN, C.N. and HINCHEY, M.G. (eds.): *Teaching and Learning Formal Methods*, Academic Press.

FOSTER, J.A. and BARNETT, M. (1996), Moore Formal Methods in the Classroom: A how-to manual, in DEAN, C.N. and HINCHEY, M.G. (eds.): *Teaching and Learning Formal Methods*, Academic Press.

GERHART, S., CRAIGEN, D. and RALSTON. T. (1994): Experience with Formal Methods in Critical Systems, *IEEE Software*, 11 (1): 21-28.

HAREL, D. (1987): Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 8: 231-271.

HINCHEY, M.G. and BOWEN, P.B, eds. (1995): Application of Formal Methods, Prentice-Hall.

JENSEN, K. (1992): *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use*, Volume 1, Springer-Verlag.

JENSEN, K. (1994): *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use*, Volume 2, Springer-Verlag.

LEVESON, N.G., HEIMDAHL, M.P.E., HILDRETH, H., and REESE. J.D. (1994): Requirements Specification for Process-Control Systems, *IEEE Trans. Soft. Eng.*, 20(9):684-707.

LEVESON, N.G. (1995): *Safeware: System Safety and Computers*, Addison-Wesley.

NASA OFFICE OF SAFETY AND MISSION ASSURANCE (1995): *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Vol 1, Planning and Technology Insertion*, NASA.

SAIDIAN, H. (1996): An Invitation to Formal Methods, *IEEE Computer*, 29(4):16-30.

VON DER BEECK, M. (1994): A Comparison of Statecharts Variants, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '94)*, LNCS 863, Springer-Verlag.

BIOGRAPHICAL NOTES

Sungdeok (Stephen) Cha received his PhD degree in 1991, in Information and Computer Science from the University of California, Irvine.

From 1990 to 1994, he was a member of the technical staff at Hughes Aircraft Company and the Aerospace Corporation where he worked on various projects on software safety and computer security. In 1994, he became a faculty member in the computer science department of the Korea Advanced Institute of Science and Technology. His research interests include software safety, formal methods and computer security. Further information on him and his research can be obtained at http://salmosa,kaist.ac.kr/.

APPENDIX: COIN-OPERATED COFFEE VENDING MACHINE(CVM)

CVM accepts inputs from users, such as coins and various buttons requesting different types of coffee, and coordinates internal activities so that user requests are satisfied. Several internal actions must be coordinated (e.g. a cup must be in place prior to releasing coffee, cream, or sugar), and several exceptional events or conditions must be dealt with. For example, one or more of the raw materials used in producing coffee of the requested type may need to be refilled. In such a case, although we gave some degree of specification freedom to students, an operator intervention may be required. Priorities among different events may need to be assigned. For example, if a user pushes buttons requesting coffee to be dispensed and demanding coins to be returned, the system must prioritise various events and decide the proper reactions. Further details were intentionally left unspecified, and students were given freedom to decide the detailed requirements. For example, some students modelled a water level monitoring system and a heater controller as embedded components so that the machine will be able to maintain coffee in a certain temperature range (e.g. between 91 and 95 degrees in Celsius). The following is a simplified Statecharts specification model of the controller for a coin-operated coffee vending machine.

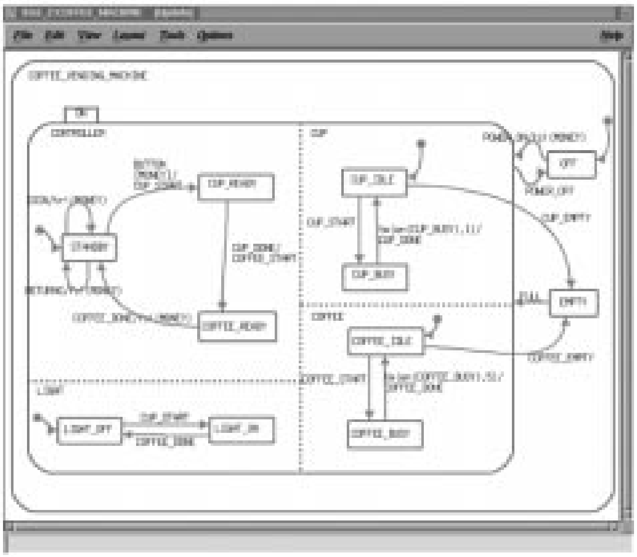

Figure 1: Statecharts Model for Coffee Vending Machine