Integrating Formal Methods into a Professional Master of Software Engineering Program

David Garlan
Department of Computer Science
5000 Forbes Avenue
Carnegie Mellon University
Pittsburgh, PA 15213

Invited Paper
Proceedings of The 8th Z Users Meeting, June 1994

Abstract

A critical issue in the design of a professional software engineering degree program is the way in which formal methods are integrated into the curriculum. The approach taken by most programs is to teach formal techniques for software development in a separate course on formal methods. In this paper we detail some of the problems with that approach and describe an alternative in which formal methods are integrated across the curriculum. We illustrate the strengths and weaknesses of this alternative in terms of our experience of using it in the Master of Software Engineering Program at Carnegie Mellon University.

1 Introduction

A critical component of any engineering discipline is a collection of formal techniques for development and analysis of the artifacts produced by engineers. Civil engineers use structural analysis based on formalisms for characterizing strength of materials. Chemical engineers use formalisms based on unit operations.

What, then, do software engineers use? The sad fact is that there is little in the way of routine formalism that is applied throughout the industry. By and large, software engineers develop systems using informal methods and procedures based on accumulated experience building similar systems.

The situation can be partly explained by the fact the state of the science underlying large-scale, commercial software development is relatively immature [Sha90]. While there are many proposals for formal software development methods, (such as program verification [Hoa72], rigorous program development [Gri81, Jon86], abstract specifications of modules [GH80], and modeling of concurrency [Hoa78]) there is no well-established body of formal foundations that are uniformly recognized as fundamental to industrial software development. There are, of course, notable successes of formal methods in industry (such as [Bar89, NC88, DG90]). However, the fact remains that the primary proponents of systematic application of formal development have been academicians and those working in the areas of secure and safety-critical systems.

But a lack of widely-accepted, scientific underpinnings is (at best) only partly the reason. Indeed, the emerging examples of successful development and the application of special-purpose formalisms (eg., in areas such as protocol verification [MS91], testing, and real-time scheduling [SG90]) indicate that the benefits of existing formal methods are simply not being exploited.

For educators this presents both a problem and an opportunity. It is a problem because the lack of a coherent body of widely-applicable, formal methods makes it difficult for educators to know what and how to teach existing techniques. It is an opportunity because it allows educators to advance the state of practice by helping to produce fresh practitioners who are equipped with a new set of practical, formal skills, and who can speed the broader dissemination and adoption of formal methods in industry.

In response to this situation, the usual approach to the introduction of formal methods in software engineering curricula is to provide a special course in which a variety of formal techniques are surveyed, and perhaps partially mastered by the students. Such a course satisfies the need to make students aware of some formal approaches to software development. But it also has a number of problems, detailed later, the most serious of which is that it tends to isolate the use of formal methods from the main-stream activities of software development emphasized in the rest of the curriculum.

In this paper we describe a different approach. Instead of segregating instruction of formal methods, we attempt to integrate it across the curriculum. To illustrate this approach we describe the recently redesigned Master of Software Engineering curriculum at Carnegie Mellon University, and evaluate our experience in using it to date.

2 Current Approaches

Current approaches to the incorporation of formal methods in professional software engineering curricula generally fall into one of three categories.

The first category avoids teaching formal methods at all. The rationale for this approach is that formal methods are not sufficiently mature to be useful to the practicing software engineer. Programs that adopt this approach are often those that serve a population of local software firms that send their technicians to school for "refresher" classes on a part-time basis. These students are usually motivated by the pressures of current software development projects and the need to acquire skills that are directly applicable to their immediate predicaments.

The second category bases an entire curriculum around a specific formal method, such as VDM or Z. In such a program students learn about that method in their first term and are expected to apply those formal skills to all of their software development activities and all of their other courses. This approach is often used in an academic master's program, where students frequently go on to an more advanced degree in computer science.

The advantage of such a program is that students learn to use one, or perhaps several formal methods well and to apply them uniformly in their practice of software development. However, there are also a number of disadvantages. First, by focusing on a small number of specific methods, students may not be exposed to the broader spectrum of formal approaches. Second, while it would be wonderful if commercial software were developed in a formal, systematic way, as noted above, the state of the practice is far, far from this goal. Consequently, students who emerge from such programs will usually find that their formal skills cannot be directly applied, since the gap between practice and what they have learned is too great.

The third, and most widespread, category adopts a single course specifically devoted to formal methods. In such a course students are usually exposed to a number of different formal techniques. (See [For91] or [Gar92] for examples.) This has the advantage that exposure can be broad, and the course can be tailored to the kinds of formal skills that are most directly applicable to practicing software engineers.

But it also has a number of serious drawbacks. First, such courses tend to emphasize notations over underlying principles. This is almost inevitable, since the breadth of coverage makes it difficult to deal adequately with the fundamental underlying principles. Second, the isolation of formal methods into a single course tends to lead students to segregate this knowledge from their other courses and software development activities. Students often feel that the formal methods course is a token gesture towards the way we would like things to be—i.e., development based on formal principles and notations—but that this has very little to do with the "practical" activities of producing industrial-scale software. Third, students taking such a course rarely become skilled users of *any* of the methods treated.

Recently we have been experimenting with a fourth category in which formal methods are integrated across the entire curriculum. Unlike the second approach outlined above, a number of formal techniques are introduced over the span of five core courses. But also, unlike the third approach, those techniques are taught in enough depth that students gain some facility in applying them to realistic problems. In the remainder of this paper we describe this approach.

3 The CMU Master of Software Engineering Program

The Carnegie Mellon University Master of Software Engineering Program (MSE) was founded in 1989 as a joint program between the School of Computer Science and the Software Engineering Institute. It is a four-semester, intensive program for professional software engineers, and leads to a terminal masters degree. The goal of the program is to develop technical leaders in industrial software engineering. These people should be able to act as agents of change in their respective organizations, and be able to apply to software development both the best of current practice as well as emerging technologies. Consequently, students who are accepted in the program must have both a strong background in computer science as well as two years industrial software development experience that indicates strong potential for leadership.

The incoming class size for the program is usually about 20 students. In the past these students have had an average of about 5 years industrial experience. About half of the students come from large corporations (Digital Equipment, HP, Westinghouse, General Motors, etc.), while the others represent a variety of smaller software development firms. Many of the students are supported by their employers, who expect them to return to the company after finishing their degree.

The MSE program is organized around three basic components: the Core Curriculum, a number of elective tracks, and the Software Development Studio. The Core Curriculum develops foundational skills in the fundamentals of software engineering, with an emphasis on design, analysis, and management of large-scale software systems [GBJ+93]. The elective tracks provide an opportunity for students to develop deeper expertise in one of several specialties, such as real time systems, human-computer interfaces, or the organizational environment of software systems. In the Software Development Studio students plan and implement a significant software project for an external client over the full duration of the program. Students work in a team environment under the guidance of faculty advisors to analyze a problem, plan a software development project, execute a solution, and evaluate their work. In that respect, the Studio is similar to the design studios that characterize architectural degree programs.

4 The Core Curriculum

The MSE Core Curriculum consists of five semester courses:

1. **Models of Software Systems:** This course treats foundations for software engineering based on the use of precise, abstract models and logics for characterizing and reasoning about properties of software systems. Specific notations are not emphasized, although some are introduced for concreteness. The main topics include state machines, algebraic models, process algebras, trace models, compositional mechanisms, abstraction relations, temporal logic. Illustrative examples are drawn from software applications.

2. **Methods of Software Development:** This course addresses the practical development of software using methods that help bridge the gap between a problem to be solved and a working software system. The intent of the course is to introduce students to comprehensive approaches to Requirements Analysis, Design, Creation, and Maintenance. Representative methods and notations include: object-oriented methods, JSD/JSP, VDM, Z, Larch, Structured Analysis/Design, Cleanroom development, and prototype-oriented development. In particular, students gain in-depth experience with three specific design methods, and are expected to understand the scope of applicability of each method. The course also introduces students to the use of supporting tools.

3. **Management of Software Development:** This course focuses on the management and organization of resources – both human and computational – for large-scale, long-lived software development projects. It treats the management of individual software development efforts and long-term capability improvement, including life cycle models, project management, process management, capability maturity models, product control (e.g., version and configuration management, change control), documentation standards, risk management, people management skills, organizational structures, product management, and requirements elicitation.

4. **Analysis of Software Artifacts:** This course focuses on the analysis of software development products including delivered code, specifications, designs, documentation, prototypes, test suites. It treats both static and dynamic analyses, such as type checking, verification, testing, performance analysis, hazard analysis, reverse engineering, and program slicing. Tools for analysis are used where appropriate.

5. **Architectures of Software Systems:** This course is concerned with the design of complex software systems at an architectural level of abstraction. It treats organization of complex software based on system structure and assignment of

functionality to design components. The main topics include common patterns of architectural design, tradeoff analysis at an architectural level, domain-specific architectures, automated support for architectural design, and formal models of software architecture.

As illustrated below, the Management, Models, and Methods core courses are offered in the Fall semester. The Analysis and Architecture courses are offered in the Spring semester. (The *Directive* is a required course that is used to balance the core content in an individualized way for each student. Typically the directive will be a course that fills gaps in an entering student's background.)

Fall 1	Spring	Summer	Fall 2
Studio	*Studio*	*Studio*	*Studio*
Directive	*Electives*		
Models			*Electives*
Methods	**Analysis**		
Management	**Architectures**	*Elective*	

5 The Role of Formal Methods in the Core Curriculum

The Core was designed to allow us to integrate the use of formal methods throughout the curriculum.

Models of Software Systems is based on the conviction that it is essential that software engineers have a coherent understanding of the fundamental mathematical models that underlie most of the good abstractions of software systems. It is in this course that students learn the basic ideas of applying mathematics to software systems. Thus the Models course provides a "scientific" basis that can be exploited by the other courses in the program.

In this respect, the course fills the role of formal methods courses in most other curricula. But unlike courses in formal methods found in other programs, Models focuses less on specific notations than on the pervasive mathematical models on which those notations rest. Moreover, unlike many formal methods courses, it considers not only the use of mathematical models for software specification, but also models that apply to testing, analysis, process modeling, and design selection.

Methods of Software Development was designed to help students gain in-depth experience with a small number of techniques that span the gulf between a problem to be solved and a successful implementation. This course coalesces material often distributed across a number of distinct courses in other curricula: requirements specification, design, creation, maintenance. Among other things, Methods builds on the notations and concepts introduced in the Models course and demonstrates their practicality to real software systems development. In particular, among the methods treated in depth by the methods course, at least one focuses on the use of formal methods of software development. In addition, the methods course uses formal models to make distinctions between methods that are not themselves formal.

The Models and Methods courses have an important symbiotic relationship: Methods puts the use of formal models into perspective and builds skills in using particular formal notations. Hence, while Models might introduce a specific notation, such as Z or Larch, to illustrate a kind of formal model, Methods shows how those notations are used in the context of a complete software development project. It accomplishes this, in part, through the presentation of a number of case studies and by requiring students to carry out a project in which they gain experience at using a formal method to solve a non-trivial problem. (Appendix B contains a description of a project used recently in this course.)

Analysis of Software Artifacts integrates various techniques for understanding the things produced by software engineers. It adopts a broad view of analysis, including topics often divided into separate courses, such as testing, formal verification, and techniques of static analysis. The analysis course builds strongly on the models course by assuming that students have already learned the basic mathematical concepts that form the basis of many of the analytical techniques. Typical kinds of formal analysis considered in this course include topics such as precondition calculation for Z schemas, analysis of state machines using tools such as Statemate [Har87] and model checking [McM92], and fault tree analysis using Petri Nets [BAR+91].

Architectures of Software Systems tackles head-on the problem of structuring large-scale software systems. It is concerned with system composition in terms of high-level components and interactions between them, rather than the data structures and algorithms that lie below module boundaries [GSO+92].

Formal methods are used in the architectures course in several ways [Gar93]. First, they are used to illustrate how to formally model the architectures of a specific system [DG90]. Second, they are used to illustrate how to capture the essence of an architectural family or style [Gar91, AG92, AAG93]. Third, they are used to explore a theory of architectural "connection" [AG94]. To drive these points home, students are asked to extend a formal architectural model as a homework assignment. (Appendix A contains the text of the most recent assignment of this sort.)

6 The Role of Formal Methods in the Studio

The Studio runs throughout the four-semester program. In the first semester students interact with a "customer" to elicit requirements for the system that they will be constructing. In the second semester students produce a design for that system. In the third semester (typically during the summer) students implement the system. In the fourth semester, students maintain the system and reflect on their experience to try to understand in retrospect what they could have done better. (Cf., the phasing chart in Section 4.)

As a representative example, a recent Studio was responsible for constructing significant portions of a mobile robot project for NASA. In particular, when finished, the robot will automate critical aspects of the NASA Space Shuttle maintenance that is performed after each shuttle flight. This development was done as part of larger NASA-funded project involving Boeing, Rockwell, SRI, and the CMU Field Robotics Center.

In addition to being a challenging and realistic software development exercise in its own right, the Software Development Studio also provides a context in which students can apply the techniques they have learned in their courses. However, the extent to which students use specific material from the core courses is left up to them. This forces them to consider which of the many techniques that they have learned can be most appropriately applied to a specific problem. In particular, the use of formal methods is entirely optional.

Our experience has been that students have chosen to apply formal methods in different, but appropriate ways. While none of the teams has yet attempted to produce a full, formally-developed software product, each has found specific points of leverage for which formal methods have been useful.

In studio work associated with the NASA project mentioned above, for instance, students used formalism in the following ways:

1. The system architect for the master control subsystem produced a Z specification of the central control functions of the system. This helped identify a key design flaw in an earlier design for the system.

2. The student responsible for the arm motion of the robot developed a formal kinematic model and associated tool for analyzing the arm movements.

3. Another student analyzed the arm base control algorithm (designed by the Field Robotics Center) and discovered a substantially improved algorithm that minimized the need for gross repositioning of the arm.

4. Several students used finite state machine models to argue the satisfaction of certain safety-critical properties.

7 The Role of Formal Methods in the Elective Tracks

The elective tracks allow students to specialize in a particular subarea of software engineering, such as real-time systems. They also allow students to design independent study courses to pursue interests not otherwise covered by regular courses.

The real-time track makes use of formal modelling in two significant ways. First, students are introduced to real-time scheduling theory, and, in particular, they are expected to be able to apply rate-monotonic analysis to appropriate real-time problems. This allows them to determine schedulability for suites of periodic tasks, and to adjust various task parameters to improve processor utilization. Second, students learn various formal techniques for performance analysis. This permits them to reduce performance monitoring data to understandable results and to track down performance bottlenecks.

Several students have also developed independent study courses to explore formal methods in greater detail. For example, one student used Z to model a portion of the proposed POSIX real-time distributed systems communications standard. Although the student had little experience either with that domain or with the standards body, he was able to produce a specification that identified a number of ambiguities, inconsistencies, and points of incompleteness [RAMP94]. He then presented his findings to the POSIX working group (Working Group P1003.21), which plans to incorporate a version of his specification as an appendix to their final report. More recently a student from Ford Motor Company has used an independent study course to model the system architecture of Ford audio system product families.

8 Conclusion and Agenda for Future Development

To summarize, our primary goal in introducing formal methods into the MSE curriculum has been to give students the intellectual tools that will enable them to make concrete use of formalisms in their practice of software engineering. To do this we have adopted a two-pronged approach towards integrating the study of formal methods into the curriculum. First, we provide a foundational course that identifies common underlying themes and techniques that are the basis of many existing formal methods. Second, we integrate the use of formalisms into the other courses by introducing specific techniques for formal specification and analysis with which students can gain deeper experience.

Overall, our experience has been quite positive. We find that students learn to approach the use of formal methods with an open mind and a willingness to discover ways to exploit them on practical problems. By and large, they appear to have a realistic view as to the benefits and costs associated with the use of formal methods. This permits them to make judicious, cost-effective choice of formalisms. They appear receptive to new techniques and theories, but they hold those theories up to fairly high standards in terms of their ability to have an impact on industrial software development.

As an instructor in the program I have found this integrated approach to formal methods challenging but rewarding. The challenge comes from the need to identify the key underlying principles that will equip students with the ability to learn and apply a variety of formalisms. The reward comes from having had some success in doing this.

At the same time the approach we have followed is not without pitfalls and risks that should be kept in mind by others wishing to apply these ideas.

- **Lack of Textbooks:** Most texts on formal methods are organized around single notations and methods. While some do a good job in laying foundations in terms of logic, set theory, and even formal, symbolic systems, none that we have found concentrate primarily on the broader underlying principles. This makes it harder to teach what we would like in the Models course without making it seem like a series of units on specific notations.

- **Lack of Tools:** Although we would love to expose students to practical tools that support the use of formal methods, we have not found it cost-effective to introduce many of the existing ones. The main problem is that each tool typically involves a large initial investment of time to get to the point where the tool is useful. A secondary problem is that few tools work together or share common interface designs. Consequently, each tool must be learned from scratch and operated in isolation.

- **Inter-course Coordination:** To make our approach successful it is important that the courses coordinate their content. In particular, since the Models course lays the foundation for the other courses the designer of that course must take into consideration the needs of the other courses. Similarly, the other courses must know the material in the Models course well enough to build on it in introducing their specific notations and methods. In our program, which is relatively small, this was not a problem, but it could be for a much larger or widely distributed program.

- **Able Students:** The students in our program are expected to develop concrete skills in using a number of formal methods and to be able to apply these to real problems in the Studio. Our experience has been that this requires students with a good background both in basic computer science and industrial software development.

- **Use of Studio:** A critical component of our approach is the use of the Studio as a test-bed in which students can experiment with the application of formal techniques on realistic problems. Many programs offer a project course that lasts a single semester. In such a short time frame it could be more difficult for students to have this kind of experience.

Several of these items suggest promising lines of educational development and further research. In particular, we would like to see new textbooks emerge that make it easier for students to quickly learn and use new formalisms. An important step in that direction would be to find a neutral, core, formal language that would allow many of the basic ideas of formal modelling, axiomatic specification, etc. to be explained without having to buy into a specific, complete formal specification method. Second, we would like to see further development of tools so that they are both more appropriate for student use and also more integrated with respect to each other. Finally, we would like to find more realistic case studies that we could use to introduce formal methods to students. Too many of the published examples are toy problems, and do not expose the more difficult issues of scalability and expressiveness for industrial software.

Acknowledgements

The CMU MSE Core Curriculum was designed by Alan Brown, Daniel Jackson, James Tomayko, Jeannette Wing, and myself. Mary Shaw and Nancy Mead also provided useful input during the curriculum design process. Gregory Abowd contributed substantially to the Methods course. Much of the success of the current curriculum is due to their course designs and delivery. The opinions and conclusions of this paper are, however, mine alone.

References

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to give meaning to software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 118(3), pages 9–20. ACM Press, December 1993.

[AG92] Robert Allen and David Garlan. A formal approach to software architectures. In Jan van Leeuwen, editor, *Proceedings of IFIP'92*. Elsevier Science Publishers B.V., September 1992. An expanded version appears as CMU School of Computer Science Technical Report CMU-CS-92-163, Towards Formalized Software Architectures.

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, May 1994.

[Bar89] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[BAR+91] W.C. Bowman, G.H. Archinoff, V.M. Raina, D.R. Tremaine, and N.G. Leveson. An application of fault tree analysis to safety critical software at Ontario Hydro. In *Conference on Probabilistic Safety Assessment and Management (PSAM)*, April 1991.

[DG90] Norman Delisle and David Garlan. Applying formal specification to industrial problems: A specification of an oscilloscope. *IEEE Software*, 7(5):29–37, September 1990.

[For91] Gary Ford. 1991 SEI report on graduate software engineering education. Technical Report CMU/SEI-911-TR-2, CMU Software Engineering Institute, April 1991.

[Gar91] David Garlan. Preconditions for understanding. In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 242–245. IEEE Society Press, October 1991.

[Gar92] David Garlan. Formal methods for software engineers: Tradeoffs in curriculum design. In *Proceedings of the Sixth SEI Conference on Software Engineering Education*. Springer Verlag, October 1992.

[Gar93] David Garlan. Formal approaches to software architecture. In David Alex Lamb and Sandra Crocker, editors, *Proceedings of the Workshop on Studies of Software Design*, number ISSN-0836-0227-93-352 in External Technical Report. Queen's University Department of Computing and Information Science, May 1993.

[GBJ$^+$93] David Garlan, Alan Brown, Daniel Jackson, Jim Tomayko, and Jeannette Wing. The CMU Masters in Software Engineering core curriculum. Technical Report CMU-CS-93-180, Carnegie Mellon University, August 1993.

[GH80] J.V. Guttag and J.J. Horning. Formal specification as a design tool. In *Seventh POPL*. ACM, 1980. (Also in *Software Specification Techniques*, pages 187-207).

[GN91] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44. Springer-Verlag, LNCS 551, October 1991.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[GSO$^+$92] David Garlan, Mary Shaw, Chris Okasaki, Curtis Scott, and Roy Swonger. Experience with a course on architectures for software systems. In *Proceedings of the Sixth SEI Conference on Software Engineering Education*. Springer Verlag, LNCS 376, October 1992. Also available as CMU/SEI technical report, CMU/SEI-92-TR-17.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, August 1978.

[Jon86] C.B. Jones. Systematic program development. In *Proc. Symposium on Mathematics and Computer Science*, 1986. (also in *Software Specification Techniques*, pages 89-108).

[McM92] K. L. McMillan. The SMV system, February 1992. Draft.

[MS91] K. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *International Symposium on Shared Memory Multiprocessors*, 1991.

[NC88] C.J. Nix and B.P. Collins. The use of software engineering, including the Z notation, in the development of CICS. *Quality Assurance*, 14(3):103–110, September 1988.

[Nii86a] H. Penny Nii. Blackboard systems Part 1: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(3):38–53, Summer 1986. Reprinted with corrections by AI Magazine.

[Nii86b] H. Penny Nii. Blackboard systems Part 2: Blackboard application systems and a knowledge engineering perspective. *AI Magazine*, 7(4):82–107, August 1986. Reprinted with corrections by AI Magazine.

[RAMP94] Neil R. Reizer, Gregory D. Abowd, B. Craig Meyers, and Patrick R.H. Place. Using formal methods for requirements specification of a proposed POSIX standard. In *Proceedings of the International Conference on Requirements Engineering*, 1994.

[SG90] Lui Sha and John B. Goodenough. Real-time scheduling theory and Ada*. *Computer*, pages 53–62, April 1990.

[Sha90] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, November 1990.

A Formal Modelling Assignment for Software Architectures

Formal Modelling of Architectural Styles
Architectures for Systems
Spring 93 (Garlan & Shaw)

This assignment is intended to help you develop some experience in manipulating a formal model of a software architecture. In this case you will be using the formal model of event systems presented in class. Following the pattern of specialization in [GN91] you are to formally characterize as event systems the two architectural idioms described below.

Blackboard Systems

Drawing on Nii's description [Nii86a, Nii86b] describe a blackboard system as a formal specialization of *EventSystem*. You may find it helpful to make the following simplifying assumptions:

- There are two kinds of components in a blackboard system: *BBdata* and *ksources*.
- The *BBdata* in the blackboard system are partitioned into a collection of *layers*.
- Each *ksourse* is associated with a some set of these layers.
- Each *ksourse* has a method *UpdateBB*, which allows it to update the blackboard when it is invoked.
- When the data in a blackboard changes, for each layer that is changed the system announces the *ChangedLayer* event to each of the knowledge sources that are associated with that layer.

You need not say anything about the run time mechanisms involved in carrying out the updates. In particular, you don't have to say how the knowledge sources update the blackboard, or how new data is added to the blackboard.

Spreadsheet Systems

Formally characterize a spreadsheet system as an event system. For the purposes of this assignment you can consider a spreadsheet to be an NxM matrix. Some of the entries in this matrix will have a *VALUE*. Some of the entries will also have an associated *Equation* that describes the value of that entry as a function over other entries in the spreadsheet. When spreadsheet entries are changed the equations that depend on those entries are implicitly reevaluated.

You might find the following definitions to be a useful starting point:

$[\mathit{VALUE}, \mathit{EQN}]$

$\mathit{Pos} == \mathbb{N} \times \mathbb{N}$

$$\begin{array}{|l}
Params : EQN \longrightarrow \mathbb{P}\, Pos \\
Eval : (EQN \times \mathrm{seq}\, VALUE) \longrightarrow VALUE \\
\hline
\forall\, e : EQN;\ vs : \mathrm{seq}\, VALUE \bullet (e, vs) \in \mathrm{dom}\, Eval \Rightarrow \#vs = \#(Params\ e)
\end{array}$$

In other words, we take *VALUE* and *EQN* to be primitive types, and a matrix position, *Pos*, to be a pair of natural numbers. We assume (axiomatically) that we can determine for each equation what its parameters are and also how to evaluate it for actual values. (The invariant guarantees that the number of formal parameters must match the number of actual parameters.)

With this as a basis you can then define a spreadsheet along the following lines:

$$\begin{array}{|l}
\hline
SpreadSheet \\
EventSystem \\
height, width : \mathbb{N} \\
boxes : Pos \pinj Component \\
eqns : Pos \pfun Eqn \\
vals : Pos \pfun VALUE \\
\ldots \\
\hline
\ldots \\
\hline
\end{array}$$

The symbol $\pinj$ indicates that each position is associated with a unique component.

You may assume that each Component in a spreadsheet (associated with a box via *boxes*) can update its value using the method *Update* whenever it gets the *Revaluate* event. Your task is to add any appropriate additional state and the state invariants. In particular, the state invariant should explain how EM is determined by the other parts of the spreadsheet.

What to Hand In

You should hand in a:

- A description of the two formal models outlined above. Ideally this should be formatted and checked using Fuzz, but it need not be. As with all Z documents, the formalism should be accompanied by enough prose to explain what is going on. You may work in groups to produce this document.

- As individuals you should also turn in commentary addressing the following questions:

  1. What important aspects of the modelled architectures are (intentionally) left out of the model.
  2. For the blackboard system, would it be possible to model some notion of “non-interference”? (You need not model it, but you should explain why or why not you answered the question in the way you did.)
  3. For the spreadsheet system, is the *Circular* property defined in the events paper a useful concept? Why or why not?
  4. Based on the formal models, briefly compare each of the two new systems with the other ones that were formally modelled. For example, you might explain which of the other systems are they most similar to?

We will mail to you a copy of the Z description for the event system described in [GN91].

Grading Criteria

Your solutions and commentary will be graded by the following criteria:

- Whether or not you are able to model the requested specializations.
- Your ability to understand and explain the formalisms in the accompanying prose.
- Your reflective commentary.

B Formal Methods Project for the Methods Course

CMU MSE 15-772
Case Study
Methods of Software Development Fall 1993
Fall 93 (Daniel Jackson)

Terminator 3: A Toy Air Traffic Control System

Controlled airspace is divided into regions. Aircraft fly along corridors called airways that connect terminal control areas (TMAs). Each TMA contains one or more airports. The airways meet the TMA at gateways. From a gateway, an aircraft travels along a glide path to a landing strip. So that planes can land against the wind, there may be several glide paths from a gateway to a strip.

The job of the controllers is to ensure that planes land safely by keeping minimal separations on glide paths and by scheduling landings into slots of fixed time width. The minimal separation of two aircraft depends on their relative speed. If the flow of aircraft into a gateway is too great, the controllers may delay aircraft by holding them in the gateways stack.

A plane entering the gateway is handed-off between the region controllers and the TMA controllers. The pilot requests clearance to enter a glide path. He may instead be given clearance to enter the stack at some level. As aircraft at the bottom of the stack are cleared to glide, aircraft above are moved down the stack, one level at a time.

The Terminator system helps control landings in a single TMA with one airport and one landing strip. It reads radar messages giving positions of aircraft and maintains for each aircraft a flight strip containing a log of the clearances it has been given and its progress through airspace. It answers queries about aircraft positions and the holding patterns of stacks. It does not issue clearances, but determines whether a requested clearance is safe. It also warns controllers if aircraft deviate from their clearances.

The system thus receives four kinds of input: radar messages, clearance requests, clearance notifications and status enquiries. It produces three kinds of output: responses to clearance requests, deviation warnings and reports. There are four kinds of clearance: to enter a stack at some level, to descend in a stack, to enter a glide path from a stack and to enter a glide path from a gateway. Radar information arrives already processed in a single stream of messages. There are two kinds of messages: that an aircraft is holding at some level in a stack, and that the aircraft is at some point along a glide path.

Complications and Elaborations

- Separation rules dependent on weather and aircraft type.
- Priority for air ambulances allows aircraft to enter glide path when stack is non-empty.
- Support function to change landing slots.
- Maintain information about controllers: who gives which clearances.
- Increase flow by reducing aircraft speed as well as by stacking.

- Allow intersecting glide paths.
- Generate clearances automatically.
- Filter out planes that are detected by radar but are overflying TMA or not in controlled airspace.
- More than one airport/TMA.