The State-of-the-Art in Formal Methods

by
Milica Barjaktarovic
Wilkes University
Wilkes Barre PA 18766
and
WetStone Technologies, Inc.
273 Ringwood Rd.
Freeville, NY 13068

January 1998

For
Michael Nassiff
Rome Research Site
AFRL/IFGB
525 Brooks Rd.
Rome, NY 13441-4505

Abstract

This report is predominantly based on observations and informal interviews during participation in the following events:

- Workshop on Integrating Formal Techniques (WIFT 98). Bocca Raton, FL, October 1998.
- Visit to SRI International. Palo Alto, CA, October 1998.
- Formal Methods Standardization Working Group Meeting. Palo Alto, CA, October 1998.
- Formal Methods PI Meeting with Hellen Gill. Palo Alto, CA, October 1998.
- Formal Methods in CAD (FMCAD 98). Palo Alto, CA, November 1998.

The overall impression from the trip is that industry needs assistance in dealing with the present realities of complex products and short time-to-market deadlines, and needs to consider formal methods as a systematic approach to dealing with the overwhelming amount of information. CAD industry seems the most willing and already uses many lightweight formal tools, such as model checkers and equivalence checkers. Telecommunications industry comes next. We have not encountered presentations in computer security and electronic commerce, but we could assume that their situation is very similar. The major task of the formal methods community will be to provide the assistance sought. Expressed needs include: more user-friendly tools; more powerful and robust tools; more real-life applications; more infrastructure such as verified libraries; more publicity of success stories and available technologies; and more user training.

Table of Contents

1 Introduction 1
1.1 Description of Events 2
2 Overview of the Existing State-of-the-Art 4
2.1 Overview of Existing Practice 4
2.1.1 Technology Transfer Issues 6
2.2 State-of-the-art in Industry 7
2.2.1 Practical Issues Related to Industrial Use of Formal Methods and Tools 8
2.2.2 State-of-the-art in Software Engineering 9
2.2.3 State-of-the-art in Hardware Engineering 12
3 Formal Methods Tools: State-of-the-art 14
3.1 Formal Tools in Industry 16
3.1.1 Industrial Experience 18
3.1.2 Integration of Formal Methods Tools 18
4 Proposed Solutions 20
5 Conclusion 22
Appendix A: Overview of Some Commonly Used Formal Tools 25
Appendix B: Formal Verification Tools in Hardware Toolkits 28
Appendix C: Summary of Related Projects 30
References 32
Glossary of Terms 34

1 Introduction

Currently, there are more than 75 different formal methods (FMs) listed on the de-facto formal methods repository at the World Wide Web Virtual Library on Formal Methods, http://www.comlab.ox.ac.uk/archive/formal-methods/. This is a de-facto database of anything relating to formal methods, with entries mostly from the USA and Europe. Formal methods are in different stages of development, in a wide spectrum from formal languages with no tool support, to internationally standardized languages with tool support and industrial users. The field of formal methods is in a great flux and evolving rapidly, leaving research laboratories and making inroads into industrial practice.

One of the major issues in the technology transfer from academic research to industrial practice is how to make formal methods more available to practitioners. First, formal methods are not appealing to industry unless there are adequate tools that support the methods. Second, since real problems require diverse approaches, and no single formal method is suitable for solving all classes of problems, it seems necessary to take advantage of different strengths of different formal methods. It would be highly desirable to integrate formal methods, especially those that have tool support.

We focus on formal tool integration as a practical step in a more widespread use of formal methods. In order to assess the latest developments in the field of formal methods and tools, we participated in several meetings and conferences. Our goal was to investigate possibilities for deriving a framework for integrating formal tools into a toolkit, either on its own, or as a part of existing toolkits.

We participated in the following events: two conferences focused on formal methods, one predominantly software oriented (WIFT'98), and another predominantly hardware oriented (FMCAD'98); a visit to a commercial research facility (SRI International); a research meeting (DARPA PI meeting); and the IEEE Computer Science Software Engineering Standards Committee (SESC) Formal Methods Planning Group. This wide variety of events exposed us to academic, industrial, and government formal methods communities, including tool makers, tool users, and funding agencies.

The overall impression is that industry is actively seeking new approaches that could assist in producing complex products while meeting short time-to-market deadlines. Industry is considering formal methods as a systematic approach to dealing with the overwhelming amount of information. CAD industry seems the most willing and already widely uses many lightweight formal tools such as model checkers and equivalence checkers. Telecommunications industry comes next. Some of the expressed needs include: more user-friendly, powerful and robust tools; more real-life applications; more infrastructure such as verified libraries; more publicity of success stories and available technologies; and more training.

1.1 Description of Events

Workshop on Integrating Formal Techniques (WIFT 98) was primarily oriented towards software engineering, and was attended equally by academia, government and industry, especially telecommunications industry, and was international in attendance. The physical proximity of Motorola's cellular phone facility contributed to industrial attendance. The workshop was unexpectedly small, because only about 30 people attended, although more than 60 were expected. Low attendance was beneficial for interaction, however, as most sessions were discussion- and hands-on oriented.

Visit to SRI International allowed for an opportunity to learn about SRI's latest projects, many of which are sponsored by the Rome Research Site IWGB group. SRI has several projects related to integration of formal methods:

- The Integrated Protocol Environment (TIPE), led by Jonathan Millen.
  TIPE is a framework that allows interoperability of tools in the application domain of computer security. The approach is based on the security-specific specification language called CAPSL, and the implementation language called CAPSL intermediate language (CIL). A specification of a cryptographic protocol is written in CAPSL and automatically translated into CIL. CIL specification is custom translated into input of various general tools, such as the PVS theorem prover or Murϕ model checker.
- Software Architecture Description Language (SADL), http://www.csl.sri.com/dsa/sadl-main.html, led by Victoria Stavridou.
  SADL is a language specializing in describing system software architecture hierarchies that are to be analyzed formally, for example for correctness and security properties. SADL can also specify data flow from high levels to low levels. SADL is intended primarily for describing structure, in which many components are connected in complex ways. SADL comes with a tool for syntactic checking. Although the SADL project is not directly related to formal tool integration, it can provide an entry point for it, e.g. SADL can model programs in most programming languages.
- Symbolic Analysis Laboratory (SAL), led by David Dill and Nataraj Shankar.
  SAL is a framework that allows interoperability of tools. The approach is to scale down problems instead of scaling up the tools, and to apply diverse tools and techniques. A common description is abstracted in ways suitable for input to various tools. Therefore , it is possible to prove large properties of complex systems using symbolic analysis of reduced abstractions. SAL accepts inputs in various forms, including Esterel, Statecharts, and JAVA, and inputs them into SVC, Murphy, PVS, SMV, and other tools. This effort started in October 1998.
- Maude effort, led by Jose Meseguer.
  This effort is a meta-logical framework based on rewriting logic. The premise is that there are underlying inference systems common to different tools, and these systems can be specified in rewriting logic. Maude is used as a meta-language in which other languages are defined with their own syntax. (An alternative way to achieve interoperability between languages would be to map from one logic to another using a meta-logical translator, which would be very difficult.) Maude is a wide spectrum logical language, based on rewriting logic, and is intended for declarative style

programming of concurrent systems. Maude specifications are executable in a way similar to model checking. In this project, Maude will be an interface to other theorem provers and model checkers. The project will develop a general formal language meta-tool, and a new theorem prover.

It would be interesting to see how these projects could be combined. For example, we could combine SADL and SAL, by having SADL rely on SAL for access to various tools. The projects most promising for contributing to the integration of a wide variety of formal tools are SAL and Maude.

IEEE Computer Science Software Engineering Standards Committee (SESC) Formal Methods Planning Group meeting, held at SRI International, was led by Dilia Rodriguez rodriguez@rl.af.mil from Rome Research Site and attended by researchers, industry and government. The consensus was that there is nothing that we can standardize in formal methods at the moment, except individual languages. Formal methods is a rapidly evolving field, and there are no “clear winners” yet in terms of methods and tools. The committee recommended that the formal methods community generate: taxonomy of terms used in formal methods; a classification of formal semantics, methods and tools; and a draft set of criteria that characterize formal methods. The draft will be discussed at a birds-of-a-feather session at the FM '99 World Congress on Formal Methods in Toulouse, France, in September 1999. An e-mail list was created to start the discussion.

Formal methods PI Meeting showed the progress of various projects led by Helen Gill from DARPA. Some of them were designed for tool integration. For example, a Cornell University effort, led by Edward Constable, is integrating several theorem provers. SRI International efforts SADL, SAL, and Maude are described above. The University of California at Berkeley Scalable Analysis Tool Kit (SAT) effort, led by Alex Aiken, is designed to help verify millions of lines of code and was used on a real-life industrial problem. There were several invited presentations. The presentation most related to formal tool integration was related to the Ptolemy Project from University of California at Berkley, led by Edward Lee. The project is described in Appendix C. The Ptolemy Project would be worth examining in more detail for the purpose of formal tool integration, because it integrates diverse non-formal tools and approaches, and it includes the MOCHA model checker as well.

During the PI meeting, two challenge problems were informally offered to the community. The first challenge problem was offered by Eric Hardner from the NSA Division of Information Security and Computer Science, ejh@tycho.ncsc.mil. The problem is providing secure, dynamic, and scalable (possibly reliable as well) multicast communications to a specified subset of users. One of the challenges with this problem is the large number of users in the specified subset, for example, one million users. The second challenge problem was presented by Victoria Stavridou from SRI International, victoria@sri.com, and deals with integrated modular avionics. The Avionics industry would like to allow different avionics functions to share the same hardware, which presents issues in combining independent components of various levels of criticality.

Formal Methods in CAD (FMCAD 98) was well attended by local industries from the Silicon Valley, both hardware manufacturers and tool vendors, as well as research community and nearby universities. The conference was not very much international in flavor. The tone of the conference was very practical and oriented towards industrial needs, with a sense of business urgency and importance.

2 Overview of the Existing State-of-the-Art

In this section, we will provide an overview of the current industrial, academic, and government practices pertaining to formal methods, and we will discuss some practical issues related to industrial application of formal methods.

2.1 Overview of Existing Practice

Formal methods are, at this time, making a transition from academic research to industrial use. Therefore, academia, industry, and government all influence the state of formal methods. We will discuss the interplay between these parties by discussing their own needs and capabilities.

Industry
Industry is dealing with increasingly shorter deadlines, such as 6 months to deliver an airplane box unit, or 30 days to test a product using English text as input. The products themselves are increasingly more complex, with several million lines of code or hundreds of millions of transistors. The current approach to design is often not systematic enough, since a common practice is to spend less time on design and more time on elaborate testing. For example, it is claimed that “80% of requirements specifications are English-text documents, and the rest is probably whiteboard “ [Comp98] p.71. Testing consumes about 50% of the product cycle and has become the bottleneck.

Conference papers, invited talks, and private conversations indicate that CAD and telecommunications industries, especially CAD, are running out of options for design and testing. Industry is actively seeking new ways to produce bug-free, standard-compliant products in timely manner, and is considering formal methods. Rolce Royce, Siemens, BMW, Daimler-Benz (“Mercedes”), Fujitsu Laboratories, Bell Laboratories, Lucent, Cadence Berkeley Laboratories, NEC laboratories, Hewlett-Packard, IBM, Intel, LSI Logic, Motorola, Rockwell-Collins, Texas Instruments, and Silicon Graphics have in-house formal methods groups, which often produce in-house formal tools. Many CASE tool vendors, for example Synopsys, sell lightweight formal tools such as equivalence checkers, and most CAD companies require their engineers to use such tools. Conferences such as FMCAD expose many success stories with the use of formal methods tools.

We will be discussing software and hardware industry in more detail in the later sections.

Academia

One of the major problems with technology transfer of formal tools from academic research to industrial use is that academia has little capacity to provide industrial strength tools. Academia in general is primarily concerned with research, i.e. development of new ideas, not making and marketing products. An additional intermediary is needed to bridge the gap between academic research and industry.

Academia has several characteristics that make the technology transfer difficult. The academic reward system applauds novel research but not development. Academic interests can be bent towards “intellectually interesting” problems and not necessarily real-life application. Academia might not have access to real-life industrial problems. Most formal methods academic research is currently done in computer science departments, on the theoretical side. “Formal methods” use to be a synonym for “formal verification” using heavyweight tools. Academics are beginning to adopt the notion that lightweight tools and aspects other than verification, such as specification, could be acceptable in academic circles and useful for popularization of formal methods. (For example, model checking was initially not “formal enough;” the World-Wide Web Virtual Library on Formal Methods has no listing of lightweight tools.) Formal methods are taught at few universities, and need to be somehow squeezed into already full academic curricula.

However, academia has the capability to produce novel research on which new industrial advances can be made. We feel that certain steps are needed to make the richness of academic research in formal methods more available to industrial use. Formal methods needs to spread into the engineering departments to gain a more pragmatic flavor; some work needs to be done on publicizing new and existing tools; and newly developed tools need to be given to a facility that will transform them into industrial products and market them. From a practical standpoint, we feel that the last item can be achieved by employing a commercially oriented team closely related to research laboratories, preferably in the close physical proximity that encourages cooperation. It is worth examining if the University of California at Berkeley has such facilities.

Academia can also aid in technology transfer if it trains more formal methods users, or at least train students in “systems” thinking. *21st Century Engineering Consortium Workshop*, held in March 1998 [21EC98], and its website, http://www.cs.indiana.edu/formal-methods-education deal with formal methods in higher education.

Government

Government has the ability to influence the widespread use of formal tools through government funding. Currently, most funding is not given for projects beyond research phase, i.e. once a tool is “invented” there is no more funding to take it to the production level. Some funding has to exist to bridge the gap between the research and production, when the tool is not ready to be commercially produced, but has such potential. Government can also give incentive to provide user training, either through existing academic curriculum or independent training.

2.1.1 Technology Transfer Issues

Most conference speakers and panelists agreed that technology transfer is a political and social process more than a process based on technical merit. It was pointed out that technology transfer is very much different than a research experiment. Some speakers estimated that technical merit is taken into consideration with the weight of less than 10%, and the remaining 90% is business needs. Business needs most commonly cited include: tool support, training, documentation and price; meeting deadlines; and financial advantage.

We feel that technology transfer of formal methods tools from academic research to industrial practice has several components: transfer of technical know-how to produce industrial strength tools; “selling” the idea to the users, i.e. getting them to use the tools; and transfer of instruction on how to run the tools.

The transition of formal tools from research labs to industrial use requires collaboration between industry and academia, which is difficult. There are numerous reports written on this subject. Some contributing factors are outlined below.

- In essence, collaboration relies on personal contact. Often, industry contact changes over time, and academic personnel, such as students, also changes.
- Industry complains that it is “hard to get the academics to produce what was desired.” The reason is possibly that academia and industry have different goals in mind. Academia is interested in publishing papers, and a large portion of industrial work belongs to the non-publishable, “grange,” work category. Industry needs to create products and profits, and it needs people who will listen to what industry has to say and produce a solution. An interviewee said: “Industry does not know what they want, but if you show them a solution they know if it is the right or wrong one.”
- Academics might find it difficult to gain access to industrial problems because it might be difficult to find an industrial contact, and/or industry might not be willing to disclose information.

The gap between academia and industry could be bridged if formal tools are not required to make a large leap between two very different settings, philosophy- and geography-wise. It is worth examining the practices at the University of California at Berkeley, where commercial products are developed next to the research labs.

Adoption of a tool depends on many factors. Tools must allow the engineers to accomplish their tasks in a timely manner. We will discuss issues pertaining to the tools in a later section. Success of tools, such as Bell Laboratories’ model checker SPIN, can be contributed to many factors, including adequate tool support, extensibility of the tool, accessible notation, and features which are useful for analysis, such as symbolic simulation.

Currently, a large portion of user training is provided by formal tool vendors, such as Synopsys Inc. Ideally, user training should happen in academia. Lack of trained formal methods users is partially due to lack of formal methods specific coursework as well as lack of systems approach in education in general. The lack of an organized system approach is one of the major drawbacks of current industrial practice, and a major barrier to the adoption of formal methods. Formal methods are an organized, systematic way to approach system engineering. It was often repeated in formal and informal talks during this trip that engineers lack the "big picture" approach, because education forces the "how to" and not "why" approach. It was stated that formal specification should "describe a forest before it describes the trees," and that few users have been trained to do so, i.e. they know how to write a specification, but not why. To help the users, we need to clearly outline the steps to be used for a particular formal method.

2.2 State-of-the-art in Industry

In general, industry has specific needs, some of which are outlined below:

- First and foremost, industry needs powerful, fast, robust tools, which can take executable specifications.
- Industry is mostly interested in tools that find bugs rather than tools that prove correctness. Companies state bluntly that they are certain that their products have bugs, and, therefore, need assistance in discovering them.
- Industry is dealing with constant change, and requires techniques and tools that allow for change even to the point of changing the already manufactured product. This property is called "dynamic requirements environment."
- Industry has no time to learn complicated new techniques. Pressured with short deadlines, engineers do not have time to experiment. (However, experience shows that, if the deadline is so short that the known techniques cannot possibly work, desperate engineers might try new techniques such as formal methods.) Not previously exposed to formal methods, industry needs some guidance in learning to use the tools.

We will focus on the state-of-the-art of formal methods in hardware and software industries. Currently, these industries are relatively disjointed, and formal methods application in them is disjoint as well. Most formal methods conferences are either hardware or software oriented. However, software and hardware are merging. For example, hardware/software co-design of systems-on-silicon has become a topic of great interest. There are several merging points between hardware and software:

- Complexities of hardware are being pushed into upper layers, i.e. software.
- Software can be embedded in hardware.
- Hardware can be designed and tested as software, e.g. in VHDL.
- Some hardware problems are similar to those encountered in the software industry. For example, protocols, such as hardware cache coherence protocols.
- Simulation is one of the primary means for testing hardware.

Therefore, it would be practical to combine the body of knowledge and experience in both fields.

2.2.1 Practical Issues Related to Industrial Use of Formal Methods and Tools

A complete product cycle includes requirements, design, specification on various levels, implementation, and testing. Currently, requirements are usually specified as informal English text, so that both customers and builders can understand them. Testing is currently done using visual inspection, simulation in hardware, running test vectors in software, and testing the final product.

Formal methods save resources when used constructively not retrospectively, i.e. when used for error detection during product development, not error elimination after product was designed. Experience shows that formal methods show the most payoff when used early in product cycle. For example, most errors occur in the requirements specification. These specifications tend to be relatively small compared to lower-level specifications, and lend themselves nicely to (semi-)formal analysis. The experience shows that the whole product cycle can be vastly improved if the requirements are made clear, complete, and correct using formal methods. Requirements specification can also be used for generating test vectors. Another example of productive use of formal methods is verification of systems that develop in time and/or are reused, since verified algorithms and modules can be reused.

Experience presents many successful and less successful examples of projects that used formal methods. Some projects did not succeed because they did not secure cooperation of application engineers and/or management, or exceeded allotted resources. The length of a product cycle is a key factor in industry. Lack of existing infrastructure, such as verified libraries and supporting theories, and lack of industrial strength formal tools can result in an unacceptably long product cycle. For example, Rockwell Collins' FM group verified the AAMP5 processor at register transfer level using PVS. Due to slowness of the tool, and the fact that the group had to develop libraries and other supporting theories, this task took an unacceptably long time of 308hrs/instruction. The next project, AAMP-FV, reused a lot of AAMP5 infrastructure. The tool makers collaborated closely with the users and improved the tool. Cost dropped to 10-20hrs/instruction. However, about 20 of the instructions were so complex that they required additional R&D and drove the cost back to 308hrs/instruction. We conclude that this kind of problem will be remedied in time, with further tool and infrastructure development.

Some companies, such as Motorola and Bell Laboratories, had positive results where use of formal methods either did not change the product cycle length, but increased the product quality, or shortened the cycle. Many such cases are documented in [ClWi96] and [Heit98]. One area in which formal methods can save time is testing. Formal methods practitioners agree that the product cycle operates on the "you pay now or you pay later" paradigm. In other words, we either spend short time on requirements, design, and specification and long time on testing, *or* we spend more time in the beginning design states and less on later testing stages. Currently, testing takes about one half of the entire product cycle. Formal methods are most useful in the beginning stages, to assure correct design, so that testing is reduced and simplified significantly. Test case generation alone can justify the cost of formal methods. This philosophy needs to be communicated to the users and management, and proven in practice. Use of formal methods also improves product quality, but this argument might not elicit enthusiastic response from the management unless it is quantified. What seems to be a universally accepted quantification is the number of bugs discovered, and the decrease in total time spent on the project, because they can be measured in terms of dollars saved/gained. For example, auto-manufacturing industry loses $3,000-4,000 every minute of factory downtime; Intel lost over $400 million on the Pentium FDIV bug. Particularly convincing are cases where formal tools discovered bugs not previously discovered in extensive testing.

Using formal tools requires knowledge of formal methods as well as application domain. Currently, there are few people trained to use formal tools. The vast majority of the users have not been introduced to the paradigm of formal methods, and there are no handbooks outlining how to use formal methods. The approach that seems to work well within the daily industrial routine is that a small group of formal methods experts (sometimes only one person) works with practicing engineers and guides them in the use of formal methods. Clearly, this is a very slow rate of technology transfer and mass training of users, but is a prudent step at the moment. The engineers need to be trained in understanding the tools they are using. This seems to be the key to successful use of formal tools at Motorola. At Motorola, engineers write SDL specifications, and hand them over to the formal methods group for verification, code generation, and feedback. This loop is repeated until the engineers are satisfied with the product.

Another approach used by formal methods groups is called Independent Verification and Validation (IV&V). In this approach, an independent team of experts uses formal methods to “shadow” activity of the development process rather than participating in the development process.

Some companies are willing to hire formal methods consultants for product development, and this approach can work in some cases. For example, AMD hired formal methods experts for verification of the AMD$_5$86 processor, and the effort was successfully completed in nine weeks using ACL2 [Heit98]. Experience shows that most large companies are not willing to disclose their designs to outsiders, and therefore have internal formal methods groups.

2.2.2 State-of-the-art in Software Engineering

Software industry in general is experiencing increase in complexity and size in software, coupled with shorter time-to-market, lack of skilled labor and high personnel turnover rate. Currently, most software is designed using informal English specification and coded almost immediately. Test vectors are generated by hand from the informal requirements specification. Clear and complete documentation is a challenge. Software industry is characterized by myriad languages, platforms, applications, and practices.

The current state of the majority of commercial software engineering is best illustrated by the example of Microsoft – products are shipped with bugs, patches are (often electronically) distributed afterwards, and money is made by selling “new and improved” releases. Our impression is that a large portion of work in the software industry is to provide improvements to the bugs that should have not been there in the first place. This increases the cost to producers and consumers, because producers must spend time to produce and distribute patches, and consumers must spend time to patch their systems. For example, personal experience in the SGI UNIX system administration shows that at least 15% of system administrator’s time is spent on fixing system bugs. Therefore, most commercial software companies, such as Microsoft, are not showing much interest in formal methods.

However, companies that have safety, security, compliance to (often international) standards, certification, or product quality in mind are interested in formal methods. Examples are avionics, nuclear, oil and gas, rail and telecommunications industry. Organizations such as Motorola and Bell Laboratories are very interested in formal methods and have their own formal methods teams on board, which often produce formal methods tools. For example, the popular model checker SPIN comes from Bell Laboratories. Financial industry, medical industry, and the security industry are also beginning to use formal methods, and the list is growing. For example, very recently the author of this report was asked to join a newly formed and growing verification group at New York City Transit Authority, which is interested in verifying critical security applications.

Since software industry encompasses many different application domains, the range of preferred methods and tools is diverse. Currently, popular specification languages are English, Statecharts (because they are graphical) and increasingly UML. Popular formal methods tools are those based on Z, SDL, VDM, and various process algebras such as CCS and CSP. Testing is usually done using English-text specifications to derive test vectors. The telecommunications industry prefers to use the Test Suite Framework Standard (TTCN) for testing. TTCN is an International Standardization Organization (ISO) standardized language for specification of abstract test suite (ATS). ATS considers a system to be a “black box,” and describes the input and the output of the system. In general, software industry does not have many universally established CASE toolkits like the hardware industry. There are some formal methods toolkits such as VDM-SL and Telelogic Tau (containing SDL tool). There are some tools for hardware-software co-design, such as Eagle by Chrysalis. Some toolkits, such as Synopsys’ Telecommunications Workbench, exist to help telecommunications industry comply with standards. Please refer to the Appendices for further details.

The software community is currently considering the Universal Modeling Language (UML) as a specification language, because UML is becoming a de-facto standard in the software industry. UML is used for specifying, visualizing, and documenting objects of an object-oriented system. Its graphical, intuitive notation gained it the reputation of a “loose, user-friendly language” suitable for high-level design. Some commercial formal methods tools, like Telelogic’s toolkit, include tools for UML. UML v1.1. was released in September, 1997. The Second International Conference on UML will be held in October 1999 in the USA. Currently, UML is a set of notations without formal semantics. UML is being standardized by the Object Management Group (OMG). When UML becomes formally standardized, it might become the "VHDL" of the software world, and could allow for integration of tools into toolkits.

Regardless of what language or tools they use, software engineers agree that the two major features needed from the formal tools are:

· generation of test vectors
· generation of code.

Since software is more flexible than hardware in the sense that changes are more easily made in implementation, the software product cycle tends to be done in a circular stepwise refinement from requirements to implementation. A good example of stepwise refinement and collaboration between formal methods group and application engineers is provided by the Motorola Illinois formal methods group.

Motorola's formal methods group uses SDT toolkit, a part of Telelogic Tau toolkit by Telelogic Inc. (please refer to Appendix C). The SDT toolkit is based on formal language called Specification Description Language (SDL). Motorola's application engineers write specifications in a domain specific language (DSL), which is a formal specification language in current use. This specification is given to the FM group, which translates the specification into a domain independent language (DIL). DIL is based on SDL. Knowledge base, such as general-purpose programming knowledge, product specific knowledge, and implementation details, is taken into consideration; and a tool called Mousetrap is then used to generate code. The code is given back to the application engineers for revision, and the whole process might be repeated. For example, the time frame to implement 153 protocol data units is approximately 2.5 weeks: 1 week to create DSL specification; less than a week to translate it into DIL, about 3 days to encode the software and 1 day to generate code. When a new standard was issued for protocol data units, it took less than a day to implement the corrections; it is necessary to change only DSL specifications and rules for translating DSL into DIL.

Not every software company is prepared to introduce formal methods into their product cycle like Motorola. The Software Engineering Institute (SEI) has developed the Capability Maturity Model for Software (CMM) [Paul93], which specifies the characteristics of a mature, systematic, capable software process. CMM is currently used by DoD and other prime contractors to evaluate contractors who are qualified to perform the software work, or to monitor existing contracts.

> "The Capability Maturity Model for Software (CMM) is a framework that describes the key elements of an effective software process. The CMM describes an evolutionary improvement path from an ad hoc, immature process to a mature, disciplined process.
>
> The CMM covers practices for planning, engineering, and managing software development and maintenance. When followed, these key practices

> improve the ability of organizations to meet goals for cost, schedule, functionality, and product quality.
>
> The CMM establishes a yardstick against which it is possible to judge, in a repeatable way, the maturity of an organization's software process and compare it to the state of the practice of the industry [Kitson92]. The CMM can also be used by an organization to plan improvements to its software process" [Paul93].

Processes are rated based on their maturity, i.e. their level of a systematic approach to software engineering, in levels from 1 to 5. The levels can be informally described as follows. Level 1 means that the organization has little awareness of systematic approach to software engineering, works ad-hoc, usually "throws code together," abandons schedules and discipline in times of crisis, and relies on star programmers. Most software companies fit in this category, especially today. With the global competition, Y2K problem, computerization of most companies and growing complexity and size of software, there is a great need for information technology (IT) skills and a great demand for skilled labor. Unfortunately, these demands can result in acceptance of poor quality programming. Level 2 organizations have some ability to follow schedules and repeat successful practices. Level 3 organizations have a stable software practice, and can repeat earlier successes. Level 4 organizations have well established, managed practices and can produce predictable results. Level 5 organizations are run efficiently and continuously keep on improving. It is estimated that only organizations at levels 3 and higher are ready to have formal methods introduced in their product cycle.

2.2.3 State-of-the-art in Hardware Engineering

CAD industry is facing the pressures of global competition and increasingly larger and more complex designs. The number of transistors per chip doubles every 18 months. 600 MHz, 10 million-gate chips will be mass-produced in the near future. Implementation errors are difficult and costly to fix. For example, the famous Intel FDIV bug cost over $400 million dollars. Therefore, time-to-market and bug-free products are the main concern in CAD. CAD industry is advancing toward production of highly integrated systems, especially for multi-technology integration such as micro- electro-mechanical systems. Currently, hundred-million transistor chips are feasible, and [NSF98] predicts billion-transistor systems-on-sillicon that include embedded software, digital and analog electronic components, micromechanical systems, optoelectronic, fluidic and other components and microstructures. Design and test aids are needed for systems-on-a-chip, embedded systems, and high performance ASICs.

An overview of issues in CAD research is presented in [NSF96], with the following conclusion:

> "The link between academic CAD researchers and the systems industry is broken. This has led to an imbalance in the mix of research that favors incremental improvements to existing tools and paradigms. Proposals to funding agencies in the CAD area are dominated by this type of work.
> Changes in technology for electronic systems is causing a re-evaluation of the abstractions on which today's CAD systems are based. New compromises and

> methodologies need to be developed. Incremental design, re-use, and support for design-space exploration are changing the nature of the design cycle and require new approaches.
> The closed CAD systems provided by most of the CAD industry make it difficult to evaluate new point tools and hamper the technology transfer of research results.”

In regards to verification, [NSF98] concludes that “methods combining formal and simulation techniques will be required.” This statement is confirmed by current practice. Formal methods groups exist in most large chip manufacturing companies, and CAD tool vendors sell lightweight formal tools. FMCAD was sponsored (“with generous support,” as stated on the conference flyer) by Cadence Design Systems, Inc., Intel, Hewlett Packard, and Synopsys. Keynote speakers included high-profile speakers from Cadence and Intel.

Current CAD practice is based on hierarchical design, where each level is tested separately. High-level design is done is English. The next level is the register transfer level (RTL), designed using logic. RTL descriptions are synthesized into the next lower level, i.e. gate-level netlists. This last level is transistor level, designed using circuit analysis. Designs are accomplished using CASE tools and communicated between levels using VHDL or Verilog. VHDL and Verilog are universal languages for CAD tools, i.e. they are accepted by most, if not all, CAD tools. Testing on each level includes simulation. Since chips are more complex and design state space has exploded, simulation has become the bottleneck.

CAD industry has done model checking of low-level designs at the RTL level. There is a need to model high-level designs, and to relate high- and low- levels. Currently, it is difficult to prove that high-level design corresponds to low-level design. Engineers are expressing need for:
- tools that provide help in early design stages
- tools that provide timing and noise analysis as well as Boolean analysis
- methodology for designing for reuse.

The CAD community agrees that the major issue in CAD industry is complexity and size, which was confirmed by the FMCAD’98 presentations. CAD industry is producing systems on a chip and hardware with up to 10^7 bits/state and 500 000 lines of VHDL code. Hardware includes 1M-gate ASICs and PLAs with 300 inputs. The current practice is to test the hardware by simulation. The state size requires simulation with huge computational power, such as 200 CPUs on wide buses. Clearly, CAD industry is running out of possibilities for designing and testing such gigantic systems, and is willing to use formal methods.

The most difficult problems in hardware engineering are cache coherence, pipeline correctness, and floating point arithmetic. In hardware, experience shows that control-intensive circuits are much harder to debug than data-intensive circuits. This fact, coupled with the ease of use and learning, contributed to wide use of the model checkers versus theorem provers in hardware verification. Widely used tools are model checker SMV (from Cadence Berkeley Laboratories or CMU), and theorem provers PVS, ACL2, and HOL. Other tools used include Murϕ, VOSS, SVC, SCR, CV, and SPIN. We observed at FMCAD'98 that Silicon Valley favors "local" tools, such as the Cadence Berkeley SMV toolkit and PVS theorem prover. Several commercial toolkits, including model checkers and equivalence checkers exist for hardware: Checkoff (by Abstract Hardware), Design suite of tools (by Chrysalis Symbolic Design), Static Verifyer (by Synopsys Inc.), RuleBase (by IBM), Affirma (by Cadence Design Systems, Inc.) and FormalCheck (by Lucent/Cadence Berkeley Laboratories.) An overview is given in http://www.isdmag.com/VendorGuide/focus/designverform.html. Chrysalis' and Synopsis' formal tools are based on VHDL/Verilog. Many other smaller tool vendors sell such tools, but we were told by a practicing CAD engineer that these tools usually have more bugs although they are faster. Tools for hardware-software co-design exist as well, such as Eagle™ by Chrysalis.

A good example of the current capability of formal methods and tools is described in the paper that won the "best paper" award at FMCAD'98. The paper is titled "The Formal Design of 1M-gate ASICs," by Ásgeir _ór Eiríksson from Silicon Graphics, Inc. (SGI). The paper describes the successful refinement of a directory based cache coherence protocol specification to a pipelined hardware implementation. The protocol is used for SGI Origin 200 and Origin 2000 servers. Although these systems have been on the market for the last two years and no bugs have been found, formal analysis exposed several errors in sequencing and flow control. The significance of this project is also in the size and complexity of the protocol. The abstract of the paper states:

> "…The hardware analyzed is the most complex part of a 1M-gate ASIC. The design consists of 30,000 lines of synthesizable RTL Verilog code. The design contains a pipeline that is 5 levels deep and approximately 150 bits wide. It has a 16-entry, 150-bit wide, context addressable memory (CAM), and has a256x72 bit RAM. Refinement maps relate the high-level protocol model to the hardware implementation. We used the Cadence Berkeley Labs SMV model checker to create the maps and prove their correctness. Here are approximately 2,000 proof obligations. The formal model has been used for three tasks. First, to formally diagnose, and then fix broken features in a legacy version of the design. Second, to integrate the legacy sub-system design with a new system design. Finally, it has been used to formally design additional subsystem features required for the new system design. The same hardware designer enhanced the design, created the refinement maps, and formally proved the correctness of the refinements." [FMCAD98], p.49.

3 Formal Methods Tools: State-of-the-art

There are more than 75 formal methods listed in the World Wide Web Virtual Library on Formal Methods [WWFM]. The formal methods community is expressing the need to integrate some of these techniques, to provide industrial strength tool support, and to demonstrate how these tools can be used.

Possible choices of tools include:

- Heavyweight tools:
  Based on use of logic and automated assistance in (human-guided) proof steps. Powerful tools especially suitable for parametric verification. The major drawback is the long learning time. Additional research and training need to be done to bring these tools within the reach of application engineers.
  - Theorem provers.
    Currently not much used in practicing industry, except for safety critical industries such as Rockwell Collins. Formal methods groups at Intel, IBM, HP, and Motorola are using theorem provers to some degree. Academic research has been traditionally oriented towards theorem proving.

- "Medium-weight" tools:
  - Model checkers.
    Based on state space exploration. Relatively fast, automatic, and resemblant of simulation. The main use in practice is to provide counterexamples, symbolic simulation, and exhaustive checking of partial properties.
    Heavily used by CAD, telecommunications, and safety critical software industries. Formal methods groups from Intel, IBM, Motorola, HP, Bell Laboratories, Cadence Berkeley Laboratories, NEC Laboratories, and Fujitsu Laboratories are using model checkers.
    Observation: Feedback on user-friendliness of various model checkers is mixed. For example, Motorola's engineers thought that the model checker SPIN is too difficult to understand, and University of Cincinnati students thought it was extremely user friendly and easy to use. Therefore, the "weight" of various model checkers can be anywhere from "heavy" to "light," depending on the users' opinions.
  - Equivalence checkers.
    Check if two models are equivalent, usually by comparing state information. Widely used in hardware verification, in tools such as Chrysalis Design suite (refer to Appendix B.)
    Equivalence checking is considered "lightweight" by many, but we are including it here because model checkers can include equivalence checking.

- Lightweight tools:
  Used in industry to automate tedious tasks and quickly and easily obtain useful results. All industrial users agree that these kinds of semi-formal tools are very much needed and appreciated, and advocate further development of such tools.
  - Completeness checkers: check if all cases are covered.
  - Consistency checkers: check that there are no conflicting cases, such as the same transition from one state leading to two different states. Could be used for checking requirements, specifications, formal specifications, design models, code, tests, and link between them.
    For example, in a tabular specification with table entries A, B, and C, consistency means that the following holds true: (A B)   (A C)   (B C). Completeness means that A B C is true.

- Constraint checkers: check constraints on the system. Constraints are usually given as a part of requirements specification.
- Type checkers: check if data and other types are consistently used.
- Symbolic simulators: execute formal specification on indeterminant data. Domain engineers visually inspect the output of symbolic simulation and determine if it appears acceptable or not. In the current industrial practice, most errors are discovered during symbolic simulation.
- Pre- and post-condition specifyers: specify what conditions hold pre- and post- certain modules. These conditions can be useful for testing and debugging. Some heavy- and medium- weight tools are able to prove that these conditions hold.
- State invariant generation: state what properties hold in a given state. These properties can be proven using heavy- and medium-weight tools (e.g. by using assert statements.)
- Dependency graph browsers: visualize what dependencies hold among variables.

There are no "clear winners" among the tools used in industry, academia, and government. Some commonly used tools are PVS, ACL2, and SMV, and to a lesser extent SDL tools and HOL. Other tools include SPIN, LOTOS (mostly in Europe), SCR, FDR, and many others. A subset of tools is described in Appendices A and B.

3.1 Formal Tools in Industry

Industrial strength tools can be produced when tool vendors perceive enough incentive to build the tools. The time to build such tools is estimated to be 1-2 years (as heard at WIFT'98). Currently, the field of formal methods tools is still growing and there are many tools and no clear "winners." Many of the languages that the tools are based on are not standardized yet, which makes tool making, purchasing and using more difficult. Furthermore, once a language is standardized, the standard evolves in time. Some of the languages that are internationally standardized are VDM, LOTOS, SDL, and TTCN. Z and UML are presently being standardized.

Currently, industry uses mostly model checkers and lightweight tools. The key benefit is clarity and not analytical power, i.e. formal methods are used to provide key insights in a short time. For that reason, the notation used must be accessible to the users and tools should be as automated as possible. Model checkers and other lightweight tools also have the benefit of ease of learning and use, and similarity to the traditional practices. The most common use of model checkers is for symbolic simulation, where engineers can visually inspect the output and recognize errors.

Since no single tool is appropriate for dealing with all classes of problems, tools need to be combined. Additional research needs to be done to understand how to compose different views of a system. For example, model checkers are excellent in describing control-intensive behavior but cannot deal with data as well. Theorem provers are excellent in describing data, but cannot specify control sequence. Currently, model checkers and theorem provers are being used together in an increasing number of tools and projects, such as PVS at SRI International, or SteP at Stanford University. Common approaches are: to abstract a model into smaller pieces, model check each piece, and theorem prove the connection between the pieces and/or particular properties of the pieces or their parts; or to model-check the low-level design and theorem prove the high-level design.

Currently, tools need massive improvements to deal with the size of real-life problems. Model checkers run out of either memory space, if they are based on storing state space (e.g. using BDDs), or time, if they are based on techniques that traverse the event tree (e.g. using directed acyclic graphs (DAGs)). Many techniques are invented to deal with this resource explosion, mostly by careful abstractions. Theorem provers can verify data-intensive models via induction and other features of logic. However, running a theorem prover is slow because proofs are driven manually. In addition, when specification changes, as it is always the case, proofs break and have to be re-run. Formal methods users are asking if some techniques could be developed for re-using the proofs.

[FMCAD98] p.50 states: “The capacity of symbolic model checking, in most cases, is limited to 150-250 bits of state.” [Comp98], p.67-68 states: “Typical property-checking methods these days can handle 600-1000 bits per state. That is effectively about 1000 flip-flops … which is likely to support a 10,000 gate module. In a datapath oriented design it might support only 5,000 gates. … Most design chunks today contain 50,000 gate modules.” Therefore, the current approach to model checking large designs is to decompose the design into manageable partitions, to abstract away all unnecessary details, and/or to keep only a representative portion of the model (e.g. if there is symmetry or repetitiveness in the model). Ideally, object oriented paradigm can be used: a large model is broken into modules that are verified separately, and then the relationship between the modules is verified. The term for this kind of approach is “compositional model checking” and is considered a mature technology by some formal methods practitioners.

Although practical model checking might not verify full design, it fully verifies at least a part of a design and can discover subtle bugs not discovered by simulation. Simulation partially verifies full design. Industry is willing to have partially analyzed properties, because its goal is to discover errors and not to prove correctness. It is also understood that both simulation and formal models represent fictitious models of real systems, and bugs in the model may not exist in the real system and vice versa.

Partitioning a model into smaller parts requires domain-specific knowledge and formal methods knowledge. It requires (deep) insight into the design as well as choice of the right abstraction, and it is currently an art form performed by domain experts. The same expert will most likely verify the design. Therefore, a design engineer also becomes the test engineer, where traditionally these roles were separate. This fusion of roles can present a stumbling block in practice.

There is a push among academic researchers to automate abstraction as much as possible. There are several varieties of abstractions, including:

- Slicing: abstracting with respect to variables
- Control: abstracting with respect to control subsystem
- Boolean: replace state predicates with boolean variables; e.g. instead of keeping track of variables *a* and *b*, keep track of the value of $a>b$.
- Discrete: continuous behavior is reduced to discrete
- Parametric: reduce N processes to a fixed number of processes.

3.1.1 Industrial Experience

Rockwell Collins and the Naval Research Laboratory (NRL) obtained very positive feedback from engineers for connecting symbolic simulation to a mock up user interface. Engineers could see how the formal model relates to their product and this helped them to accept formal methods. We conclude that engineers are more likely to use tools that have the look and feel of the tools they are familiar with, or tools that clearly help them accomplish their tasks in timely manner.

Experience shows that engineers prefer:
- graphical tools
- informal tools that help automate tedious tasks
- executable specifications, preferably integrated with simulation of user interface. This allows for rapid prototyping and early review with the customers.

There is a lack of certain types of tools, and formal tools could fill the gap. Some specific industrial needs include:
- domain specific notations for niche markets
- tools that can provide good documentation
- some methodology to guide users. To address the lack of trained users, we could have handbooks, or “engineering cookbooks” with “how-to” instructions.
- CAD verification tools that provide not only Boolean analysis but also timing and noise analysis
- methodology for hardware design suitable for reuse
- generation of test vectors and code for software industry.

3.1.2 Integration of Formal Methods Tools

Integration of formal methods can happen on many levels, including tools, languages, models, notations, methods, and techniques. The most practical step would be to integrate tools, which would indirectly integrate other aspects of different formal methods. When integrating tools, we need to keep in mind that:
- Integration is different than translation.
- Information must coexist in both items being integrated.
- It would be desirable to create a method to integrate different formal methods.

Once we decide what kinds or which tools we desire to integrate, the technically non-trivial issue of tool integration remains. Tool integration can happen on many levels. Possible choices and related issues are:

- Language extension (i.e. embedding): cumbersome, slow and inconvenient
- Hand translation: error prone
- Point-to-point translation (i.e. features of one tool are added to another): does not produce elegant results, but could be acceptable.
- Generic frameworks: seems like the best approach to find some commonality between the tools and "glue" them together using this commonality. The information shared as the "glue" must exist in both tools. There are many possible commonalities between the tools, including:
    - tools could share a common language (e.g. VHDL);
    - tools could share common data;
    - tools could have underlying inference systems that can be specified in rewriting logic.

  Integrated tools ought to share common interface. One approach is to have one tool produce output to be fed into another tool; another would be to be able to call one tool without exiting the other (i.e. have shared data). The first approach is easier to implement but works more slowly in practice.

Formal methods tools can be integrated into the existing "non-formal" toolkits, or can be integrated into separate formal toolkits such as Telelogic Tau toolkit. "Non-formal" toolkits are widely accepted in industry. Hardware engineering has CAD/CASE toolkits, provided by companies such as Mentorgraphic or Cadence. Tools in such a toolkit can accept/produce VHDL or Verilog code, and this ability allows communication between the tools. Software engineering does not have such universally well-known, widespread toolkits and underlying standards. The software community is currently considering Universal Modeling Language (UML), which is becoming a de-facto standard in the software industry. Currently, it is a set of notations without formal semantics. When UML becomes formally standardized, it might become the "VHDL" of the software world, and could allow for integration of tools into toolkits. Currently, some lightweight formal tools are sold either with CAD/CASE toolkits or separately, by companies such as Cadence, Synopsys, IBM, and Chrysalis.

Some work on integration of formal tools into toolkits is already done or being done. Projects ESPRESS, UniForM, PROSPER, and many others are working in this area. Some projects, like Ptolemy, are not integrating exclusively formal tools, but have a rich experience in integrating diverse approaches. (Please refer to the Appendix C for more detailed descriptions of the projects.) We were also told that other commercial companies exist in the USA that have a rich experience in integrating formal tools, but cannot disclose their experience because the products are customer proprietary. An example of such a company would be Software Productivity Consortium. It would be worth the time to investigate what these projects have to offer. One commonality that we notice between these projects is that they are multi-year, collaborative efforts; and that the tools integrated are those either belonging to or closely related to the project participants. Integrating tools often requires access to tools' source code and tools' makers.

When asked if they would like formal tools integrated within a toolkit or separate, CAD users said that they would prefer customizable, stand alone formal tools, which can be integrated into the existing design flow. Tool vendors expressed a strong preference for stand alone tools, because that approach would increase their revenues.

4 Proposed Solutions

Based on our findings, we have constructed an outline of steps that can be taken to allow for integration of formal tools into existing industrial practice.

Step 1
The main questions to be answered are:
· What are the problems that users need solved?
· What kind of tools and features would be useful to the users?

To answer these questions, we must interview decision makers and cutting-edge researchers to find out what direction the field is moving in, and domain engineers to find out what the practicing engineers use and need in their daily work. In other words, we need to understand the current process flow and directions in which it is evolving, as well as the tools used. This task might be difficult to realize in practice, because commercial companies are very protective of their process flow as it contains trade secrets. Off-the-shelf tools are highly customized to fit a particular design flow. Companies that might be more willing to disclose their process flow are small and medium size companies.

More specific questions that need to be answered are:
· What application domain should the toolkit address?
  Since different application domains require different notations and features, it would be the best to develop a toolkit for one specific application. We narrow down the choices for relevant application domains to: telecommunications, CAD, and software industries. Telecommunications and CAD industries are the most mature application domains suitable for introduction of formal methods tools. Software industry in general is not expressing sufficient interest yet, except for the safety-critical software.
· What audience should be targeted as users of the formal toolkits?
  Possible choices are: academic or industrial researchers; design engineers; application domain engineers; testing engineers, or implementers. A related audience to keep in mind is tool developers. Our preference is that the toolkit should address working engineers and their daily needs.
· What level of abstraction should the toolkit address?
  For example, should the toolkit address requirements, specification, high-level design, intermediate-level design, low-level design, testing, and/or implementation. Ideally, the toolkit could be used all the way from top-level design to the implementation and testing, in a stack-like fashion, where tools communicate from one layer to another.
· What kinds of tools should be integrated in the toolkit?

The answer to this question depends on the answers to the above questions, as well as availability of tools and possibility for collaboration with tool makers. Possible choice of tools is outlined in the previous section. We feel that the toolkit must include lightweight tools, user-friendly interface (preferably graphical), and informal tools that help automate tedious tasks.

A question remains if the tools integrated should be general-purpose or specialized. It seems the most effective to include general purpose customizable tools, so that users can tailor them to their own needs and evolve the tools as their design flow evolves.

Based on stated industrial needs, we conclude that tools must address the issue of:

- Clarity: notation must be accessible to engineers and have power of expressiveness.
- Error finding: tools must identify bugs and mistakes, rather than to prove correctness. Errors must be clearly identified, i.e. what line of code, what statement, why it is an error.
- Effectiveness: tools must be fast, easy to use, and powerful.
- Robustness: the tool itself should be as bug-free as possible; proofs, analysis, and validation should also be robust, i.e. should not easily break when specification changes.
- Extensibility: users must have the ability to customize the tool to a particular design flow.
- Automation and methodology: tools should be as automated as possible. At least, users should have some methodology to guide them.
- Reusability and change: proofs/modules should be reusable during frequent changes in requirements and/or specification.
- Scalability: tools must have some means to organize large and/or growing specifications.
- Compatibility with existing practices/tools: compatibility comes in two forms: the tool must be able to “fit” into the existing design flow; and the tool should resemble existing tools.
- Integration of graphical and textual notations, acceptable user interfaces.
- Proving application specific properties: tools must be appropriate for the task at hand.
- Tool support: the tool must have good technical support, user manual, documentation, upgrades, and training.
- Stability: how likely is it that the tool and the tool vendor will exist in the next five years?

Step 2

The main questions to be answered are:

- What kind of functionality is already existing in the current formal tools?
- Which formal tools would satisfy requirements outlined in Step 1?
- How can these tools be integrated into the design flow outlined in Step 1?

Understanding formal tools requires creating taxonomy and classification of the current formal tools, clearly outlining capabilities, inputs and outputs. This work would take some time and effort, and is recognized by the SESC Formal Methods Planning Group as a very much needed step in identifying what formal tools we have available to work with. Tool makers are requesting this information in order to be able to integrate the tools. It would be most effective to start this work with a selected subset of tools.

We also need to evaluate a selected subset of tools to identify those suitable for inclusion in the toolkit. We must examine if a tool is robust enough for industrial use. This criteria includes:

- Efficient and bug-free execution: the tool must be sufficiently tested by a variety of users, reported bugs must be fixed quickly, efficiency needs to be improved, and the improved tool distributed. It is useful to have tools and updates downloadable from the web, and to have user mailing lists and user meetings.
- Documentation: usually one of the weakest links. User manuals are frequently not clear and complete, because they are written by tool makers who understand the tool so well that they take the information for granted, or do not have the time or the preference to write reports. Error messages generated by the tools also need to be improved, to show where and what occurred. This work requires many days of "grange" work that is not publishable and not well represented in grant funding. A commercial development team could implement these user-friendly features.
- Technical support: it is necessary to have dedicated personnel, well versed in the tool, available to answer user questions. Clearly, this takes personnel time.
- Learning curve and user friendliness: the tool has to use notations that are easy to learn by the users, because industrial users do not have the time to learn complex tools on the job. Until more users are trained in formal tools, we need to resort to this measure.

Other critical issues for success of the toolkit include:

- technology transfer
- publicity and marketing
- extensibility of the tool, i.e. users must be able to customize the tool.

Understanding how the tools can be integrated might be difficult. The major barrier is that it is impossible to obtain source code for many tools. Even if source code is obtained, often collaboration with the tool maker(s) is needed. In addition, it is common that each company highly customizes off-the-shelf tools to fit into the company's design flow. If we cannot obtain tools that we could build on, we could examine source code of CAD tools constructed for government projects.

5 Conclusion

Industry needs assistance in producing complex, bug-free products while meeting short time-to-market deadlines. Industry is considering, or already using, formal methods. For example, many large companies have internal formal methods groups, and CAD/CASE toolkits include model checkers and equivalence checkers. Formal methods are widely accepted in hardware engineering, and less so in software engineering. In any case, formal methods have not reached a maturity stage yet and are in transition from research to practice. The consensus is that this transition can be aided by producing industrial strength formal methods tools, more technical infrastructure such as verified libraries, more (publicized) industrial case studies, and increasing the pool of trained users. It is often stated that formal methods and tools must be accessible to engineers through their notation, features, and approach.

Therefore, we must come up with a strategy that will effectively use formal tools within the currently existing parameters, and work towards longer-term goals. We feel that, at this time, it seems most practical to have lightweight formal tools working in conjuction with the existing tools and practices. This approach will demonstrate usefulness of formal methods in addressing problems not easily addressable otherwise, and allow the users to get accustomed to formal methods' paradigms. More heavyweight tools would be used by a small group of experts. Meanwhile, a feedback cycle could be started: the need for better tools and positive experiences with the existing tools will cause greater industrial demand, so that academia will be willing to train more users; and vice versa. Unless there are more trained users and/or need, widespread use of formal methods will always stay at the lightweight level.

We can recommend several measures that could improve the current situation and increase industrial use of formal methods:

- Provide funding for tools that are transitioning from research environment into production environment. This would be an appropriate role for the government.
- Create opportunities for user training, either by influencing the academia or providing independent user training. An appropriate role for the government would be to provide training opportunities and thus break the chicken-and-the-egg cycle of academia waiting for a larger industrial demand, and industry demanding more when there are more trained users demanding more tools.
- Generate publicity for formal tools and research, so that the available knowledge can be disseminated through wider circles of users and different communities. For example, cross-pollinating research in hardware and software protocols would be useful to both communities. We recommend:
  - one-page fliers to be distributed at conferences
  - tool demos at a wide array of both academic and commercial conferences
  - a web page, linked to the World Wide Web Virtual Library on Formal Methods, with clear taxonomy of terms used in formal methods and classification of the tools
  - a web page, linked to the World Wide Web Virtual Library on Formal Methods page, with documentation of case studies, especially success stories. Publicized success stories could be used to justify use of formal methods.
  - a conference to present SRI International and other research of interest to the Rome Research Site to a wide community. The purpose of this conference would be to disseminate information and to provide a forum for discussion and exchange of ideas.
- Develop taxonomy of terms used in formal methods and classification of formal tools, clearly stating tools' capabilities. Ideally, we would include tools' inputs and

outputs. This survey needs to be well publicized, including: a web page linked to the World Wide Web Virtual Library on Formal Methods page; notices to e-mail lists; notices to newsgroups; and one-page fliers to be distributed at conferences. This survey would greatly benefit anyone who is searching for the right tool as well as toolkit makers. This effort would take a log time to complete fully, and, therefore, could be started for a subset of tools and/or subset of properties.

- Create verified libraries and other supporting theories, in order to build infrastructure to make formal tools easier to use. Additional research would determine the language and contents of the libraries.
- Integrate formal tools into existing tools that are already used in industry. This approach would require either access to tools' source code, or extensible tools. Industry might not be willing to disclose their design flow, and additional research needs to be done to determine what kinds of tools are needed.
- Create a framework to integrate formal tools into usable, industrial strength toolkits. Some work is already being done in this arena, mostly in Europe (please refer to the list of related projects). Integrating different tools is a difficult undertaking that might possibly requires access to the source code of various tools and collaboration with the original tool developers. The feasibility of the framework would be best illustrated by including parts of a tool into an existing and available toolkit.

Appendix A: Overview of Some Commonly Used Formal Tools

Toolkits Which Include Model Checkers

Model checkers are automated tools based on state space exploration. There are many model checkers in current use. They have become quite popular in the recent years, and are claimed to have “saved the reputation of formal methods” because of their ease of use and learning. Model checkers are automated tools suitable for users not trained in rigorous mathematics and theorem proving skills. Currently, model checkers are widely used in industry and in academia. Some of the most popular ones in the USA include, in alphabetical order: Murpϕ, SCR, SDT, SMV, and SPIN. Model checkers popular in Europe are FDR, LOTOS, SDT, and VDM tools.

The variety in model checking tools is due to the fact that tool makers had different use of the tools in mind. Roughly speaking, we can categorize model checkers based on the language supported (which is based on application domain, level of use, and intended audience), and measures used to achieve efficiency. Priorities for choosing or creating a language could include:

· Simple notation, such as: programming-like, tabular, or graphical
· Concurrency
· Message communication
· Message buffering
· Value-passing (i.e. use of data)
· Equivalence between various specifications
· Synchronous versus asynchronous modeling

The language is also catered to the specification level it will be applied to (e.g. requirement specification). Most model checkers are intended for describing control-intensive systems and usage by “regular” engineers. Most model checkers include a symbolic simulator (it is a byproduct of storing state space), and temporal logic for describing system properties.

Measures for increasing efficiency usually involve the language in which the tool is implemented (such as C, Lisp, or ML) and algorithms for reducing state space, such as exploiting symmetry (Murpϕ), hash reduction (SPIN), or automatic abstraction of unnecessary details (SCR*).

Murpϕ

http://verify/stanford.edu/dill/murphi.html

Murpϕ toolset is intended for asynchronous, interleaving modeling of concurrent systems, particularly hardware protocols. Murpϕ toolset is based on Murpϕ language, which is similar to Lisp programming language. Murpϕ language has constant, type, variable and procedure declarations, rule definitions, a description of the next state, and a collection of invariants (Boolean expressions that reference a variable.) Murpϕ includes a model checker. It is possible to check for invariant violations, error statements, assertion violations, and deadlock. It is possible to interface to SVC. Methods for increasing efficiency of Murpϕ include: state reduction by using symmetry, execution of action rules in reverse order, and keeping track of the number of processes in a particular state instead of the processes themselves.

Symbolic Model Verifier (SMV)
ftp://emc.cs.cmu.edu/pub/
SMV was intended for hardware systems. The SMV model checker is based on language SMV, which hierarchically describes finite-state machines at any level of detail, both synchronous and asynchronous. SMV uses CTL for verification. CTL is an extension of Boolean logic with four temporal formulas: atomic formulas, propositional logic, next-state logic, and "until" logic. CMU SMV is implemented using BDDs. There is also SMV tool from CadenceBerkeley Labs.

Software Cost Reduction (SCR*) toolkit
http://www.csr.ncl.ac.uk/projects/FME/InfRes/tools/fmtdb044.html,
http://www.itd.nrl.navy.mil/ITD/5540/personnel/heitmeyer.html.
SCR* is intended for modeling safety-critical software systems at the level of requirements specification. It is meant to be used by application engineers with no training in formal methods and no access to formal methods experts. SCR* uses SCR tabular notation to describe required system behavior as the synchronous composition of non-deterministic environment and deterministic system. Each SCR specification is organized into two parts: dictionaries (which define static information such as variable names), and tables (which specify how the variables change in response to the input events.) the SCR* toolkit contains the model checker SPIN and several lightweight tools:

- Specification editor
- Graphical dependency browser: aids I understanding dependencies among variables
- Consistency checker: exposes syntax errors, variable name discrepancies, missing cases, unwanted non-determinism, and circular definition.
- Simulator

It is possible to automatically generate state invariants (i.e. properties that hold true in a given state) from the requirements specification. Before calling SPIN, it is possible to automatically reduce state space by eliminating unnecessary details.

SPIN
http://www.dcs.gla.ac.uk/~tfm/fmt/hol.html
SPIN is Bell Labs' tool intended for concurrent systems, particularly communication protocols. The underlying language is PROMELA (Protocol or Process Meta Language), which is a C-like non-deterministic language based on Hoare's CSP. SPIN contains a symbolic simulator, model checker, and verification with linear temporal logic and invariants. SPIN uses hash algorithms to reduce state space. SPIN is implemented in C, and its source code is publicly available.

SDT toolkit is described in Appendix B as a part of Telelogic Tau toolkit.

CAD/CASE formal tools will be described in Appendix B.

Theorem Provers

What we call "theorem provers" are actually mechanized proof assistants. These automated tools are powerful in their ability to specify and verify large amounts of data, and require at least six months of learning time.

A Computational Logic for Applicative Common Lisp (ACL2)

http://www.cs.uwyo.edu/~cowles/acl2
ACL2 uses first-order logic of total recursive functions. The syntax is that of Common Lisp, which allows interface to programming in Lisp. The ACL2 theorem prover can be run in completely automatic mode or with user interaction.

Prototype Verification System (PVS)

http://pvs.csl.sri.com/
PVS uses a language based on a variant of higher-order logic. It contains a theorem prover, interface to temporal model checking, and consistency and completeness check tools for tabular specifications of the kind advocated by Parnas.

HOL

http://www.dcs.gla.ac.uk/~tfm/fmt/hol.html
The HOL theorem proving system is based on higher order logic, and uses Standard ML of New Jersey (SML/NJ) as the specification language. Since SML/NJ is public, HOL is extensible.

Stanford Validity Checker (SVC)

http://agamemnon.stanford.edu/~levitt/vc/index.html
SVC is intended to be used as a fast decision procedure for validating logical expressions, and was used successfully in that capacity in PVS. A PVS proof would be reduced to a smaller subgoal, and SVC can be used on that subgoal. SVC can be used for static analysis of formal, high-level specifications of safety critical software. SVC is based on first-order logic. The Web page has little information about the other features of the tool.

Consistency Tools

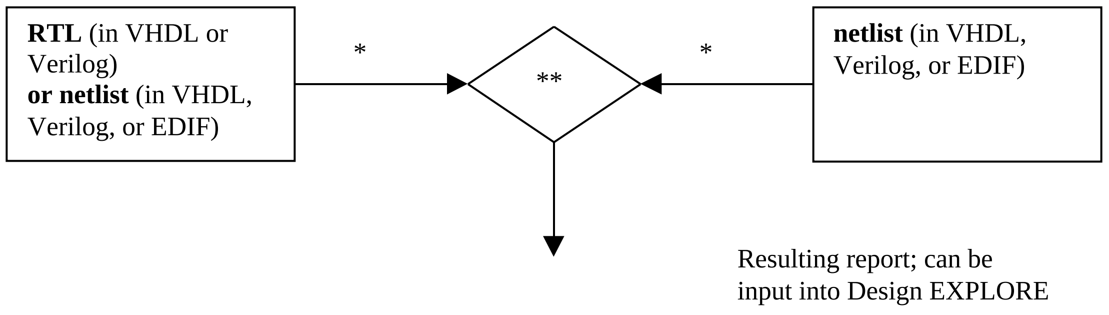
Most toolkits that include model checkers either include consistency tools or can be used for consistency checking. Theorem provers also can be used for consistency checking. Therefore, we could use SMV, PVS, SCR, HOL, and many other tools.

Appendix B: Formal Verification Tools in Hardware Toolkits

There are many formal hardware tools, but we will focus on several.

Design Suite of Tools, by Chrysalis Symbolic Design, Inc., http://www.chrysalis.com. Chrysalis specializes in formal tools for CAD, and its entire product line consists of three formal tools. The tools can operate independently or as a toolkit, either stand-alone or as a part of a CAD/CASE toolkit. The interoperability with regular CAD/CASE tools is achieved by using Verilog, VHDL and EDIF for input/output to Design tools.

Design VERIFYer is a formal equivalence checker. Chrysalis markets it as a replacement for gate-level regression simulation. The tool can perform hierarchical verification: RTL to RTL, RTL to gate, gate to gate or RTL to switch. The block diagram of the tool is given below.


RTL (in VHDL or Verilog) **or netlist** (in VHDL, Verilog, or EDIF)
*
**
*
netlist (in VHDL, Verilog, or EDIF)
Resulting report; can be input into Design EXPLORE

* convert into Chrysalis logic representation
** compare the logic driving each state bit

Design EXPLORE is an interactive debugger, which takes as input RTL, netlist, or library sources in VHDL, Verilog, EDIF, or HDL and converts them into Chrysalis logic.

Design INSIGHT Assertions is a model checker which uses assertions to prove desired properties. Assertions are Boolean statements that are true at certain points in code. Assertions can be used to state properties to be proved, such as input and output module requirements, and to state assumed properties that hold true. Properties can be expressed in VHDL, Verilog, or tool command language (tcl), Temporal operators are also available. Chrysalis is marketing this model checker as a time-saving tool, because modules can be tested before the whole design is operational. Most test benches operate at the chip level, so all modules have to working in order to test the logic.

The following is taken from Chrysalis' press release:

> November 16, 1998 - Toshiba America Electronics Corporation (TAEC) has selected Chrysalis' Design VERIFYer equivalency checker for their gate-to-gate formal verification. "We were spending thousands of hours on gate-level simulation for regression analysis and needed to streamline our design process through the use of formal verification. Chrysalis' Design VERIFYer is a proven solution for gate-to-gate comparisons," said Jeff Berkman, vice president, SLI engineering for TAEC. "Additionally, Chrysalis' application expertise and support is excellent."

FormalCheck, by Lucent Technologies Bell Labs Design Automation, http://www.lucent.com/. The tool has been given to Cadence Berkeley Laboratories. The following is an excerpt from the hardware industry news:

> "Lucent Technologies Inc.'s Bell Labs Design Automation (Murray Hill, N.J) has been awarded a contract for more than $1.25 million by Silicon Graphics/Cray Research (Mountain View, Calif.) for the follow-up purchase of additional licenses of FormalCheck, its system-level model-checking technology."

Static Verifyer, by Synopsys Inc., http://www.synopsys.com/products/staticverif/staticverif.html. Synopsys is marketing Static Verifyer toolkit approach as the substitute for the current practice. Currently, engineers do not simulate much at the RT level, but simulate extensively at the gate-level. Full chip gate-level simulations run for days, up to several weeks. Synopsys' solution is to: simulate at the RT level using Synopsys' **VCS** (Verilog simulator) and **Cyclone** (VHDL simulator), and perform static verification at the gate level using Static Verifyer. Static verification consists of static timing and equivalence checking. **Prime Time** is Synopsys' static timing tool, which checks if the delays of all paths in a circuit satisfy timing assertions. Synopsys **Formality** is equivalence checker, which compares functional equivalency of the RTL source to the post-synhesis netlist. The last part of the Static Verifyer toolkit are tools used for testing: Design Compiler Expert Plus, TestGen, and PathTest.

Static Verifyer requires synthesis-based design flow and synchronous design.

Eagle™, by Synopsys Inc., http://www.synopsys.com/products/hwsw/eagle_ds.html. Eagle™ is a toolkit for hardware-software co-design, intended for be used as a part of XRAY® Debugger software development toolkit from Mentor Graphic Corp.. Debugger works with Microtec C and C++ compilers. Eagle™ toolkit consists of VHDL and Verilog simulators and VERA™

Appendix C: Summary of Related Projects

Telelogic Tau toolkit, by Telelogic Inc., http://www.telelogic.com. This is a commercial toolkit for all phases of software development. Telelogic Inc. is a Sweden-based company with offices in New Jersey, California and Illinois. The entire toolkit costs about $10,000. The tools are: ORCA for UML-based requirement specification, SDT for system design and implementation in SDL, and ITEX for testing with TTCN. SDL is a graphical, internationally standardized formal language by ITU. SDL is an object oriented language that can describe concurrent finite state machines. It accepts data inputs as abstract data type (ADT) or Abstract Syntax Notation One (ASN.1). The Telelogic Tau toolkit automatically generates executable code, and automatically generates executable test vectors. SDL and TTCN have the ability to work with ASN.1. SDT supports writing specifications of CORBA objects in Interface Description Language (IDL). An observation: several months after viewing the Telelogic Tau demo at WIFT'98, we were contacted by a sales person. We were impressed by Telelogic's commitment to sales and marketing.
A similar toolkit, called **ObjectGEODE**, based on OMT, SDL, and MSC, is produced by Verilog, USA/France.

Open Mechanized Reasoning Systems (OMRS) project, by John McCarthy, at all.
http://www.mrg.dist.unige.it/omrs/index.html

The 15th International Conference on Automated Deduction (CADE-15), July 5-10, 1998, Lindau, Germany. Topic: Integration of Deduction Systems.
http://i12www.ira.uka.de/Workshop/cfp.html.

Proof and Specification Assisted Design Environments (PROSPER) Project, http://www.dcs.gla.ac.uk/prosper. This is a recently started 3-year joint project between the Universities of Glasgow, Cambridge, Edinburgh, and Karlsruhe, and commercial tool builders Prover Technology AB (Sweden) and IFAD (Denmark). Prover Technology AB sells theorem proving technology, training and research. Prover Technology AB invented Stålmarck's algorithm, which became very popular in Europe because it provides a fast and efficient alternative to BDDs for verification purposes. Prover Technology AB implements this algorithm in its Otter theorem prover, and was able to verify formulas up to 350,000 connectives [FMCAD'98], p.82. IFAD produces the VDM-SL Toolbox, currently used at more than 50 sites, mostly in large European companies. Both companies plan to use the PROSPER toolkit to improve their commercial products. The goal of PROSPER project is to have a user-friendly, flexible, extensible toolkit that can easily integrate with various commercial tools. The PROSPER toolkit will provide a GUI interface and a common theorem proving support for a software and a hardware toolkit. The software toolkit is the VDM-SL toolbox produced by IFAD. The hardware CAD toolkit will be built by PROSPER and will be based on VHDL/Verilog. Other features of the project include: next generation interfaces and tools for requirements specification, using natural language and timing diagrams; and open proof architecture, containing an API for CAD/CASE tool integration, the core proof engine, and a plug-in interface.

Universal Formal Methods Workbench (UniForM), http://www.informatik.uni-bremen.de/~agbkb/UniForM/. This is a (completed?) 3-year joint project between the University of Bremen and Carnegie Mellon and Stanford Universities, and companies SRI International and the Kestrel Institute. The goal is to perform application-driven basic research and develop reliable software for industry. The project will result in a tool kit with a common user interface, to be available in public domain in 1998. "UniForM is a specific framework instantiated with specific tools to handle communicating distributed systems and real time requirements." The project includes a case study, control of a safety critical system, and using the example of developing a decentralized central unit for a single track tramway networks. This project follows the V-model, developed by the German Ministry of Interior and accepted as standard in German software industry. V-model is a standard development process model for planning and executing information technology projects. UniForM combines duration calculi, CSP (with FDR), Z (with type checker) and PLC. The tools are integrated using development by transformation from one tool to another.

ESPRESS project, http://www.first.gmd.de/~espress. This project was sponsored by Daimler-Benz AG, and carried through cooperation between German government and universities. ESPRESS is designed for developing software for complex, safety-critical embedded systems, such as intelligent cruise control systems. The toolkit covers the entire product cycle in application areas of automobile electronics and traffic light control. ESPRESS toolkit provides one interface to many tools: Statecharts and Z notation are used to represent requirements, Isabelle/HOL-Z tool is used for validation, verification and generation of test cases, and commercial tools such as Statemate are used for editing, type checking, and specification validation. ESSPRESS was presented at the Z User meeting (ZUM), International Conference of Z Users, September 1998, Berlin, Germany. The conference was sponsored by Daimler-Benz.

Project KorSys (Korrekte Software für Sicherheitskritische Systeme, or Correct Software for Safety-critical Systems), http://www4.informatik.tu-muenchen.de/proj/korsys/all/index.html, is a joint effort between companies BMW, Siemens AG, FZI, and ESG, and universities Technical University of Munich, and the University of Oldenburg. This project deals with methods and tools for reactive and finite-state systems. Application areas include avionics system at ESG and locking and hybrid systems at BMW. The project page contains little information in English, and a paper in German.

Ptolemy Project, http://ptolemy.eecs.berkeley.edu/. This is University of California at Berkeley ongoing effort, led by Edward Lee. The project includes a model checker MOCHA by Tom Henzinger from UC Berkeley, but is mainly integrating non-formal tools. Nevertheless, the project is of interest to us because it provides expertise in integration of diverse methods and tools. The project's goal is to develop techniques ad tools for modeling heterogeneous, reactive systems, particularly embedded systems.

Ptolemy can model heterogeneous systems such as those including hardware and software, analog and digital, and electrical and mechanical devices. Ptolemy can also model systems that are complex in the sense that they mix widely different operations, such as signal processing, feedback control, sequential decision making, and user interfaces. The project's web page claims that "using Ptolemy software, a high-level dataflow model of a signal processing system can be connected to a hardware simulator that in turn may be connected to a discrete-event model of a communication network." It would be worth further research to investigate how the Ptolemy Project could provide us with a wealth of experience in tool integration. Ptolemy research influenced products by Cadence, Motorola, HP, and many other companies.

Formal Methods Europe (FME), http://www.cs.tcd.ie/FME/, is a European organization supported by the Commission of the European Union, with the mission of promoting and supporting the industrial use of formal methods for computer systems development.

Forschungszentrum Informatik (the Research Center for Information Technologies, FZI), at the University of Karlsruhe, Germany, http://www.fzi.de/prost/prost_e.html. FZI is an industry-oriented research institution with the mission to implement technology transfer from the university to the industry. It is not entirely clear from the web page, but it appears that FZI is a commercial company rather than government or university sponsored center. The FZI Software Engineering Department focuses on: the development of methods and tools for the economical development and use of software; the methodology of design and constitution of reusable software; development of open programming environments; integration strategies of software tools; practical applicability of formal methods; and the reorganization and re-engineering of legacy systems.

References

[CD98] "Formal Verification: Essential for Complex Designs." *Computer Design* 37(6), June 1998.

[ClWi96] Edmund M. Clarke, Jeannette M. Wing, et. al. "Formal methods: State of the Art and Future Directions." *ACM Computing Surveys*, 28(4):626-643.

[21EC98] *21st Century Engineering Consortium Workshop: a forum on formal methods education*. March 1998, Melbourne, FL. Sponsored by Air Force Rome Research Site, organized by Michael Nassiff. For a draft of the workshop report, contact Steven Johnson, Indiana University. Affiliated web site: http://www.cs.indiana.edu/formal-methods-education.

[Heit98] Constance L. Heitmeyer. "On the Need for 'Practical' Formal Methods," Formal Techniques in Real-Time and Real-Time Fault-Tolerant Systems, *Proceedings of the 5th Intern. Symposium* (FTRTFT'98), Lyngby, Denmark, September 14-18, 1998, LICS 1486, pp. 18-26, (invited paper).

[Kits92] D.H. Kitson and S. Masters, *An Analysis of SEI Software Process Assessment Results: 1987-1991*, Software Engineering Institute, CMU/SEI-92-TR-24, July 1992.

[NSF96] Gaetano Borriello (ed.), Future Research Directions in CAD for Electronic Systems. Workshop sponsored by NSF/CISE/MIPS, Seattle, May 1996. http://www.cise.nsf.gov/ccr/nsf-workshop/index.html

[NSF98] "Final report: NSF Workshop on Billion-Transistor Systems," Princeton, NJ, March 1998. http://www.ee.princeton.edu/~wolf/nsf-workshop/final-report.html

[Paul93] Mark C. Paulk, at all., "Key Practices of the Capability Maturity Model, Version 1.1." Software Engineering Institute, CMU/SEI-93-TR-25, February 1993. http://rbse.ics.nasa.gov/CMM/TR25/tr25_o1.html

[FMCAD98]Ganesh Gopalakrishnan, Phillip Windley (eds.), *Proceedings of Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, Palo Alto, CA, USA, November 1998. Lecture Notes in Computer Science 1522, Springer.

[PROSP] "Proof and Specification Assisted Design Environments (PROSPER) Project Programme." ESPRIT LTR Project 26241, March 1998. Contact: T.F. Melham, Department of Computing Science, University of Glasgow, G128QQ, Scotland. http://www.dcs.gla.ac.uk/prosper.

[SDL92] "Specification and Description Language (SDL-92)." Z.100 (3/93). "SDL Combined with ASN.1." Z.105 (3/95). ITU General Secretariat, Sales Section, Place de Nations, CH-1211 Geneva 20.

[STTT] Springer International Journal on Software Tools for Technology Transfer (STTT). http://sttt.cs.uni-dortmund.de. Since September 1997.

[TTCN] "Framework and Methodology for Conformance Testing of Implementation of OSI and CCITT Protocols: TTCN Standard Definition." X.290, ISO/IEC-9646-3.

[WWWFM] Jonathan Bowen (webmaster), "The World-Wide Web Virtual Library on Formal Methods." http://www.comlab.ox.ac.uk/archive/formal-methods.html.

[WWWSC] Jonathan Bowen (webmaster), "The World-Wide Web Virtual Library on Safety Critical Systems." http://www.comlab.ox.ac.uk/archive/safety.html.

Glossary of Terms

ADT (abstract data type): a data type with no specified data structure. Instead, it specifies a set of values, a set of operations allowed on the values, and a set of equations that operations must fulfill.

ASN.1 (Abstract Syntax Notation One): a language specifically designed for describing structured information that is conveyed across some interface or communications medium. ASN.1 is an international standard ISO/IEC 8824 and is a key ingredient of OSI.

Cache coherence protocol: protocol used in hardware engineering to maintain cache coherence in shared memory application. Described in "Protocol Verification as a Hardware Design Aid," by David L. Dill, Andreas J. Drexler, Alan J. Hu and C. Han Yang, *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992, pp. 522-525. as:

> "Directory-based cache coherence is a way of implementing a shared memory abstraction on top of a message-passing network, by recording in a central directory which processors have cached readable or writeable copies of a memory location. Maintaining cache coherence can be complicated.
>
> For example, if a processor *p* wants a writeabe copy of a location which is cached read-only by processors $\{q_i\}$, a request for a writeable copy is sent from *p* to the directory. The directory then sends a writeable copy to *p* (which can then proceed) and an invalidation message to every $\{q_i\}$. Each q_i then invalidates its copy and sends an acknowledgment back to the directory, which is waiting for all the invalidations to arrive before processing any more transactions on that location.
>
> Although this simple transaction sounds simple enough, the problem becomes more complicated when one considers scenarios in which several different transactions on the same location have been initiated at the same time, especially when messages are not guaranteed to arrive in the same order they were sent."

CCITT, now known as ITU-T (The Consultative Committee for International Telegraph and Telephony, now known as ITU- Telecommunications Standardization Sector): the primary international body for fostering cooperative standards for telecommunications equipment and systems.

CMM (Capability Maturity Model for Software): a framework that describes the key elements of an effective software process. The CMM describes an evolutionary improvement path from an ad hoc, immature process to a mature, disciplined process, and establishes a yardstick against which it is possible to judge, in a repeatable way, the maturity of an organization's software process and compare it to the state of the practice of the industry [Kitson92]. The CMM can also be used by an organization to plan improvements to its software process. [Paul93].

CORBA (Common Object Request Broker Architecture): a standard set by OMG for specifying ways of interaction between a server and a client. CORBA uses IDL.

IDL (Interface Description Language): textual language used in CORBA, to specify which services are offered and available. IDL is based on C++ and extended with constructs for distributed objects.

IEC (International Engineering Consortium): conducts a broad range of university and industry cooperative programs consisting of educational forums and workshops, research studies, publications, Web education, and management services. http://www.iec.org.

ISO (The International Organization for Standardization): is a worldwide federation of national standards bodies from some 130 countries, one from each country. ISO is a non-governmental organization established in 1947. The mission of ISO is to promote the development of standardization and related activities in the world with a view to facilitating the international exchange of goods and services, and to developing cooperation in the spheres of intellectual, scientific, technological and economic activity. ISO's work results in international agreements which are published as International Standards. http://www.iso.ch.

ITU (International Telecommunication Union, formerly CCITT): an international organization within which governments and the private sector coordinate global telecom networks and services. http://www.itu.int.

OMG (Object Management Group): an international standardization body which develops commercially viable and vendor independent specifications for the software industry. The goal is to develop a heterogeneous computing environment by introducing standardized object-oriented software. OMG developed CORBA. The consortium includes over 800 members, predominantly from the USA. http://www.omg.org.

OSI (Open Systems Interconnection): a set of ISO/IEC standards for layered system architecture that allows computer networking. A competing standard is TCP/IP. http://ganges.cs.tcd.ie/4ba2/.

SDL (Specification Description Language): is a graphical, internationally standardized formal language by ITU [SDL92]. SDL is an object oriented language that can describe concurrent finite state machines. It accepts data inputs as abstract data type (ADT) or Abstract Syntax Notation One (ASN.1).

TTCN (Test Suite Framework Standard): ISO standardized language for specification of abstract test suite (ATS). ATS considers a system to be a “black box” and describes the input and the output of the system [TTCN].

Murϕ: model checker from Stanford University.

SMV (Symbolic Model Verifier): model checker, either from Carnegie Mellon University or from Cadence Berkeley Labs. Cadence SMV model checker uses languages Verilog and SMV. FormalCheck is the commercial version of Cadence SMV tool.

SPIN: model checker from Bell Laboratories.

VDM (The Vienna Development Method): a collection of techniques for the formal specification and development of computing systems. It is based on a specification language called VDM- SL. Standardized as ISO/IEC 13187-1996.

UML (Universal Modeling Language): specification language which is becoming a de-facto standard in software industry. UML v1.1. was released in September 1997. Second International Conference on UML will be held in October 1999 in the USA. Currently, UML is a set of notations without formal semantics. UML is being standardized by the OMG.

Acknowledgments

The author wishes to thank all the people who volunteered their opinions and experience when informally interviewed during this trip. The list is too long to enumerate everyone. Thanks to Frank Weil, Marc Blackburn, Bob Busser, Anthony Hall, Perry Alexander, Steve Miller, Constance Heitmeyer, David Dill and his students, John Rushby, Paul Miner, David Greve, Steven Johnson, Shiu-Kai Chin, Damir Jamsek, Dan Craigen, Warren Hunt, Rajeev Ranjan, Dang-Jae Kim, Joseph Lu, Tom Melham, and numerous others. Foremost, thanks to Michael Nassif for the insight to arrange this trip and all the work involved with it, to John Faust for doing the administrative work, and to WetStone Technologies, Wilkes University and AFOSR Summer Research Program for providing time for it.