

# The Affordable Application of Formal Methods to Software Engineering

James F. Davis  
University of Maryland University College  
3501 University Blvd. East  
Adelphia, MD 20783  
1-800-888-UMUC  
jim@thedavisnetwork.com

## ABSTRACT

The purpose of this research paper is to examine (1) why formal methods are required for software systems today; (2) the Praxis High Integrity Systems' Correctness-by-Construction methodology; and (3) an affordable application of a formal methods methodology to software engineering. The cultivated research for this paper included literature reviews of documents found across the Internet and in publications as well as reviews of conference proceedings including the 2004 High Confidence Software and Systems Conference and the 2004 Special Interest Group on Ada Conference. This research realized that (1) our reliance on software systems for national, business and personal critical processes outweighs the trust we have in our systems; (2) there is a growing demand for the ability to trust our software systems; (3) methodologies such as Praxis' Correctness-by-Construction are readily available and can provide this needed level of trust; (4) tools such as Praxis' SparkAda when appropriately applied can be an affordable approach to applying formal methods to a software system development process; (5) software users have a responsibility to demand correctness; and finally, (6) software engineers have the responsibility to provide this correctness. Further research is necessary to determine what other methodologies and tools are available to provide affordable approaches to applying formal methods to software engineering. In conclusion, formal methods provide an unprecedented ability to build trust in the correctness of a system or component. Through the development of methodologies such as Praxis' Correctness by Construction and tools such as SparkAda, it is becoming ever more cost advantageous to implement formal methods within the software engineering lifecycle. As the criticality of our IT systems continues to steadily increase, so must our trust that these systems will perform as expected. Software system clients, such as government, businesses and all other IT users, must demand that their IT systems be delivered with a proven level of correctness or trust commensurate to the criticality of the function they perform.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*SIGAda'05*, November 13–17, 2005, Atlanta, Georgia, USA.  
Copyright 2005 ACM 1-59593-185-6/05/0011...\$5.00.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*formal methods*.

## General Terms

Design, Reliability, Security, Verification

## Keywords

Correctness, Engineering, Formal, Proven, Trust

## 1. INTRODUCTION

It's just after 4pm on a Friday in New York City. Workers throughout the city are taking to the streets to head home for a peaceful weekend. Some have decided to take the subway, some taxis, cars, buses, and others simply walk home. Everyone is expecting a typical New York City commute home; long lines and gatherings in the subways, parking lots on the expressways, and crowded streets and buses. To everyone's surprise and amazement, this would be far from the typical commute; in fact the entire city will be coming to a screeching halt. Why? It's 14 August 2003, the day a majority of the Northeastern US felt the effects of a major power black-out. It wouldn't be until late November that the task force responsible for investigating the cause of the black-out determines that one of the most significant factors that caused this black-out was a "software failure at First Energy Corp."—One of the major energy providers for the Northeast [16].

Rewind to February 1991, Operation Desert Storm. To battle the barrage of SCUD missiles that Saddam's army lobbed across the Iraqi border towards US troops in Saudi Arabia, Kuwait as well as towards Israel, the US employs the Army's Patriot Missile System. This system, originally designed with a "track-via-missile" software guidance system to provide short-term defense of Soviet planes and missiles was now employed as a long-term defensive weapon to counter the Iraqi SCUD missiles; its software had to adapt [11]. Despite attempts to patch the software system in order to allow the system to correctly run beyond its original design limits, on 25 February 1991 a Patriot Missile battery that had been running for over 100 hours had fired to intercept an incoming SCUD and failed. As a result, the Patriot missile missed its target and fell on an Army barracks killing 28 Americans [11].

On January 25, 2003 the computer world experienced the effects of the "fastest computer worm in history," the Sapphire worm, also known as the Slammer Worm [10]. This worm exploited a

buffer overflow in Microsoft's SQL server and SQL Desktop Engine software that had been discovered in July 2002 [10]. The effects of the Sapphire Worm included "paralyz[ing] [South] Korea, disrupt[ing] 13,000 ATMs in the US and disable[ing] emergency services in Seattle" as well as interfering with elections and causing air travel delays [5,10].

The three aforementioned brief case studies are examples of software engineering failures that resulted in the loss of life or critical mission failures—loss of revenue at a critically high level or a significant mission, corporate or national capability. The software systems that we thought were not critical systems have become so integrated into our business cultures and society that we simply cannot function without them or with them in a degraded state. For example, the US' gross domestic product "is over 98% dependent on IT" [5]. Bill Wulf, President of the National Academy of Engineering, during his talk at the 2<sup>nd</sup> Annual National Software Summit relayed the story of boiling a frog and how one must gradually heat the water up in the pot otherwise the frog would jump out [15]. He "fear[s] that our country is about to get boiled" due to the ever creeping criticality of our nation's software systems, the lack of quality in the development of these systems and the complacency and reduction of our own national engineering capability [15].

One common weakness in the above examples and as highlighted within the discussion that Mr. Wulf presented is that of a lack of verification and analysis of software systems. This is, of course, only one of many weaknesses in the above examples; however, it's the focus of this paper. In an effort to tackle the lack of verification and analysis throughout the industry, Microsoft's Tony Hoare issued a Grand Challenge for Computing Research, "The Verifying Compiler" [7]. The Grand Challenge is to build a "compiler [that] uses automated mathematical and logical reasoning to check the correctness of programs that it compiles" where one of its tests for success includes programs being "formally verified" [7]. This research paper focuses on the formal verification of software. In particular, on why formal methods are needed for today's software systems, introduce and examine the Praxis High Integrity Systems' "Correctness-by-Construction" methodology, and examine how this formal-methods based methodology can be applied to today's software systems.

## 2. Why Formal Methods

Formal methods are "mathematically based techniques for the specification, development and verification of software and hardware systems" [6]. Formal methods can provide mechanisms for mathematically proving that a software system or component does what it is supposed to do; nothing more, nothing less. These mechanisms are provided by way of formal specification languages such as: Z, Communicating Sequential Processes, Vienna Development Method, Larch as well as Formal Development Methodology [17]. These formal specification languages are typically used to develop unambiguous models to describe how their system is to operate [17]. One could then take these specified system models and run a mathematical proof against them to ensure their validity [17]. The result of the proof would then provide sufficient verification evidence to a systems developer that their system, as formally specified in their specification language of choice in their model, is correct.

Although formal methods may appear to be the *silver bullet* to the software crisis, it should be well understood that there is, of course, no *silver bullet*, and that there is a significant cost to the use of formal methods [6]. "Proofs of correctness require significant time to produce [and are of] limited utility other than [to provide] the assurance of correctness" and therefore, have been typically reserved for engineering fields where the benefits of such proofs make formal methods worth the resources [6].

If formal methods are to be reserved for safety-critical as well as mission-critical systems, as defined above, then we must clearly identify what systems fall into these categories. One could easily identify aerospace and military systems that would apply; however, as was discussed in the above Sapphire Worm case study, many of today's systems are slowly and quietly creeping into our critical business processes such as those responsible for national elections, banking systems and many more. These should now fall into the mission-critical systems category as well. Once these systems are identified as mission-critical systems a more appropriate risk analysis can be implemented to understand the relationship between the cost to formally verify their functionality versus the impact if their functionality was lost due to a system failure, or in this case a security flaw. As reported by John Chen, Sybase CEO, during the 2<sup>nd</sup> Annual National Software Summit, below is a list of estimated revenue losses by industry due to system downtime per hour:

- Shipping: \$28,000
- Teleticket Sales: \$69,000
- Airline Reservations: \$89,000
- Home Shopping: \$113,000
- Pay-per-view: \$150,000
- Credit Card Sales: \$2,650,000
- Financial Markets: \$6,450,000 [5].

As can be clearly seen here, the impact of one hour of downtime alone, which happened to be about the same amount of time before the first responses to the Sapphire Worm were implemented, can be quite costly and significantly impact mission capability [10].

## 3. Correctness by Construction Methodology

Correctness by Construction is a software engineering methodology developed by Praxis High Integrity Systems and has been primarily geared towards the aerospace industry and their safety-critical systems [1]. This methodology, however, is also making significant impacts in a variety of industries where security-critical and mission-critical systems are of great concern [4]. The Correctness by Construction methodology has resulted in, as documented by the Praxis HIS company: (1) High productivity—savings felt during the testing phases; (2) Low defect software—very few remaining errors post-delivery; (3) Warranted software—Praxis' standard issuance of software warranties; (4) Low support costs—easily maintainable; and finally, (5) Satisfied customers—allows successful achievement of client's underlying business goals [13]. Case studies of the use of the Correctness by Construction methodology, such as the Multi-Application Operating System (MULTOS) Project—a 100KLOC smart-card operating system—achieved results such as: a defect

rate of 0.04 defects per KLOC (4 total); high-productivity levels, 28 lines of code per day; and most importantly the system satisfies its users, performs well and is highly reliable [12].

The Correctness by Construction methodology is composed of seven key principles: (1) Expect requirements to change; (2) Know why you're testing; (3) Eliminate errors before testing—"testing is the most expensive way of finding errors," second only to "your customers find[ing] them for you; (4) Write software that is easy to verify—reduce the verification burden by spending more effort on writing good code; (5) Develop incrementally—code a little, verify a little; (6) Some aspects of software development are just plain hard—there is no silver bullet and we should not "expect any tool or method to make everything easy;" (7) Software is not useful by itself—we must develop software in concert with and in response to the enterprise architecture, business processes, CONOPs, user manuals, design documentation, etc. [14]. There are also five distinct characteristics of Correctness by Construction systems: (1) Use of static verification; (2) Use of small, verifiable design steps; (3) Generation of certification and evaluation evidence as a side-effect to the development process; (4) Appropriate use of formality; and finally, (5) the use of right notations and tools for the job [2]. The most difficult of the principles and characteristics is that of being able to eliminate errors before testing or the use of static verification. Our ability to perform static verification—"techniques of verification that do not include execution of the software"—greatly "depends on the language or notation under development" [2,3]. Praxis recommends the use of their specially developed tool for this purpose, SparkAda [3]. SparkAda is a subset of the Ada programming language where certain functions and capabilities have been removed to greatly reduce the number of potential ways a certain function could be performed thus eliminating certain ambiguities [3]. The unique capability that SparkAda does provide, however, is the capability and standardized semantical notations to assist the theorem prover in formally verifying the SparkAda code [3]. These semantical notations are based on the formal language specification Z and axiomatic set theory [3].

#### **4. How to Affordably Apply Formal Methods Through the Use of C-by-C**

The Correctness by Construction methodology calls for a strict adherence to a formalized software engineering process. These steps must include the typical system development steps of enterprise architecture analysis all the way through system disposition. To apply the formal methods through Correctness by Construction, the entire process from start to finish and in order must be accomplished and strictly followed. No matter how much one attempts to affordably apply formal methods to an analysis, if the system validation fails then all other efforts have been fruitless as they will probably have to be completely reworked. Thus, without a solid software engineering process in place there is no affordable way to apply formal methods.

In accordance with the Correctness by Construction methodology, if a strict adherence to a formalized software engineering process is in place then a valid approach to applying formal methods would be to selectively apply, per some risk analysis, formal methods to subsets of the larger system. By strict adherence to a formalized software engineering process, requirements and design

defects should be greatly reduced and easily validated. This will allow the system developers to be able to concentrate on the verification and validation of the code itself. Now through the use of system risk analysis and cost benefit studies, the system developers can determine what functions and system areas are most at risk and would best benefit from a formal methods analysis to ensure correctness. For example, a system risk analysis might show that the system interfaces (human, machine and Internet) are high risk while an internal algorithm to sort data and its associated control code is low risk. In this example the system developers would want to concentrate the formal method tools on the system interfaces while, although maintaining software quality, the internal algorithm and its control code need not be formally analyzed.

Referring to the case studies discussed briefly in the introduction, if one were to attempt to apply an affordable measure of formal methods they would have to first enter the systems engineering lifecycle back at the point of analysis of the enterprise architecture and concept exploration/development. Assuming the process runs smoothly through requirements development, a system risk analysis for each particular system could then be performed to identify those system areas most at risk of causing system failure or system degradation at a level high enough to cause mission/business failure. In the Northeast Black-out case study, because of the significant importance and the impact that any fault might have in the overall electrical system it may be determined that a formal analysis is acceptable for the entire software system. For the Patriot Missile System case study, the timing code and other code related to the trajectory and tracking of targets may be deemed the most mission-critical portions of the system and thus be formally analyzed to ensure correctness. It is important to note, however, that this case study also highlights a systems engineering process failure when adapting the system to a new environment and that is to reassess and re-enter the system development process from the beginning and get the requirements right first. Finally, the Sapphire Worm case study is somewhat different and much more complex. There is no one system or responsible person who would be best to determine where the formal methods analysis would be best applied. In this case, businesses, organizations and users must identify their mission-critical processes and the IT solutions that support those processes. Once identified, they must demand from their IT solution providers that they provide proven solutions and warranties that state the developer is responsible for the correctness of their software components. The IT solution providers must know their customers, know how their solutions are used and perform cost-benefit and risk analyses to determine where formal methods would be appropriate in the development of their individual solutions.

#### **5. Recommendations**

The affordable application of formal methods such as the methods inherent to the Praxis HIS Correctness-by-Construction methodology is capable and available. By front-loading a development effort to ensure system requirements are validated prior to implementation and further by selecting critical components to undergo formal verification, the cost required to perform system test and maintenance is greatly reduced. To enable this, software engineers and their clients both have specific responsibilities. Before this can be done, however, the ability to

apply methodologies such as Correctness-by-Construction requires system and software engineers to strictly adhere to formalized engineering processes that allow for the system validations at each step throughout the process.

Software engineers must know their clients, must understand how their systems will be used and what portions of their systems are critical to their clients. Software Engineers must adhere to formal engineering development processes. Software Engineers must further implement their systems with the appropriate formal analysis as identified by system risk analysis and cost-benefit tradeoffs. Software Engineers must ensure their clients are informed and involved during a system risk or tradeoff analysis.

Clients must know their businesses and what portions of their business processes are the most critical to achieve or maintain mission success. Clients must ensure the software systems they utilize are built with the appropriate strengths to support their critical business processes. Clients must demand software warranties that state that the software will meet the documented need of the client within the documented environment and conditions.

## 6. Conclusion

Formal methods when properly utilized provide an unprecedented ability to build trust in the correctness of a system or component. Through the development of methodologies such as Praxis' Correctness by Construction and tools such as SparkAda, it is becoming ever more cost advantageous to implement formal methods within the software engineering lifecycle. As the criticality of our IT systems continues to steadily increase, so must our trust that these systems will perform as expected. Software system clients, such as government, businesses and all other IT users, must demand that their IT systems be delivered with a proven level of correctness or trust commensurate to the criticality of the function they perform. Software engineers must have the tools and capabilities to build this level of trust.

Our dependence on software systems has reached such an extremely high level that the past arguments that formal methods were too expensive in regards to time and money is transitioning to the fact that it costs too much in downtime and maintenance not to formally prove these systems correct [6]. As discussed in the above case studies, the systems involved clearly were critical to the safety-of-life, success of the mission or success of an essential business or national capability. These systems were delivered without a measurable level of correctness and ultimately failed causing significant impacts to their system owners. The new system's "*-ility*" that must be required, measured and evaluated is correctness and this can only truly be done through the appropriate application of formal methods. The ability to apply formal methods to software engineering is available. The relative cost to its benefit of applying formal methods to software engineering is quickly becoming affordable. Until our dependence on software systems in fulfilling our national, business and personal strategic goals and objectives diminishes, our need for trust in these software systems will continue to grow.

## 7. ACKNOWLEDGMENTS

My thanks to the University of Maryland University College's Professor Louis Blazy for his assistance and teachings during his Spring 2005 System Development course for which this paper was originally written.

## 8. REFERENCES

- [1] Amey, P. (2002, March). Correctness by Construction: Better can also be cheaper. Crosstalk Magazine. Retrieved 20 April 2005, from [http://www.praxis-his.com/pdfs/c\\_by\\_c\\_better\\_cheaper.pdf](http://www.praxis-his.com/pdfs/c_by_c_better_cheaper.pdf).
- [2] Chapman, R. (2004). Correctness by Construction: A TSP-Friendly Approach to Ultra-Low Defect Software. Retrieved 20 April 2005, from <http://www.sei.cmu.edu/tsp/tug-2004-presentations/chapman.pdf>.
- [3] Chapman, R. (2004, November). SPARK, an Intensive Overview. Proceedings of the SIGADA Conference, 14-18 November.
- [4] Chapman, R. & Hall, A. (2002, Jan). Correctness by Construction: Developing a Commercial Secure System. IEEE Software. Retrieved 20 April 2005, from [http://www.praxis-his.com/pdfs/c\\_by\\_c\\_secure\\_system.pdf](http://www.praxis-his.com/pdfs/c_by_c_secure_system.pdf).
- [5] Chen, J. (2004, May). The Enterprise: Unwired. 2nd National Software Summit, Reston, VA. Retrieved 20 April 2005, from <http://www.cnsoftware.org/nss2report/Chen-NSS2v.3.pdf>.
- [6] "Formal Methods." (n.d.). Retrieved 20 April 2005, from [http://en.wikipedia.org/wiki/Formal\\_methods](http://en.wikipedia.org/wiki/Formal_methods).
- [7] Hoare, T. (2004, August). The Verifying Compiler: a Grand Challenge for Computing Research. High Confidence Software & Systems Conference, March. Retrieved 20 April 2005, from [http://research.microsoft.com/~thoare/The\\_Verifying\\_Compiler.ppt](http://research.microsoft.com/~thoare/The_Verifying_Compiler.ppt).
- [8] Knight, J.C. (2004). Correctness by Construction: An Introduction to SPARK Ada. Retrieved 20 April 2005, from [http://www.cs.virginia.edu/~jck/cs651/slides/14.correctness\\_by.construction.pdf](http://www.cs.virginia.edu/~jck/cs651/slides/14.correctness_by.construction.pdf).
- [9] Michael, J.B., Voas, J.M., & Linger, R.C. (2004, May). Proceedings of the Center for National Software Studies Workshop on Trustworthy Software, 8-9 April. Retrieved 20 April 2005, from <http://www.cnsoftware.org/nss2report/NSS2FinalReport04-29-05PDF.pdf>.
- [10] Moore, D., Paxson, V., Savage, S., et. al. (2003). The Spread of the Sapphire/Slammer Worm. Retrieved 20 April 2005, from <http://www.cs.berkeley.edu/~nweaver/sapphire/>.
- [11] Morgan, T. & Roberts, J. (2002). An Analysis of the Patriot Missile System. Retrieved 20 April 2005, from <http://seeri.etsu.edu/SECCodeCases/ethicsC/PatriotMissile.htm>.
- [12] Praxis High Integrity Systems. (n.d.). Case Study: MULTOS Certification Authority. Retrieved 20 April 2005, from <http://www.praxis-his.com/services/software/casestudy.asp>.

- [13] Praxis High Integrity Systems. (n.d.). Correctness by Construction Approach. Retrieved 20 April 2005, from <http://www.praxis-his.com/services/software/approach.asp>.
- [14] Praxis High Integrity Systems. (n.d.). Correctness by Construction Principles. Retrieved 20 April 2005, from <http://www.praxis-his.com/services/software/principles.asp>.
- [15] Wulf, W.A. (2004, May). Do we have a creeping crisis and are we losing our edge? 2nd National Software Summit, Reston, VA. Retrieved 20 April 2005, from <http://www.cnsoftware.org/nss2report/Wulf%20Luncheon%20Presentation.pdf>.
- [16] Verton, D. (2003, November). Software failure cited in August blackout investigation. ComputerWorld. Retrieved 20 April 2004, from <http://www.computerworld.com/securitytopics/security/recovery/story/0,10801,87400,00.html>.
- [17] Vienneau, R. (1993, May). A Review of Formal Methods. Retrieved 20 April 2005, from <http://www.dacs.dtic.mil/techs/fmreview/definition.html>.