FORMAL VERIFICATION OF NUCLEAR SYSTEMS: PAST, PRESENT, AND FUTURE

Mark LAWFORD and Alan WASSYNG

Abstract: In this paper we review the Systematic Design Verification Process used on the computer controlled shutdown systems of the Darlington Nuclear Generating Station Shutdown Systems. The Software Requirements Specification (SRS) made extensive use of tabular expressions to document the requirements as did the Software Design Description (SDD). Systematic Design Verification was then performed based upon the 4-Variable Model to verify that the design was correct with respect to its requirements. Custom tools were developed to process the SRS and SDD documents to produce "block theorems" for the PVS theorem prover that were used to verify the majority of the functional requirements. We discuss how the formal methods were integrated into the forward going software development process and techniques that were used to manage the complexity of the verification task. We offer some lessons learned in the process and discuss the future of formal verification for nuclear systems.

Keywords: Formal methods, verification, software development process, software tools, safety-critical software.

Introduction

The two Shutdown Systems (SDS1 and SDS2) for the Darlington Nuclear Generating Station in Ontario, Canada, were the first computer controlled shutdown systems in Canada. This paper is not about the original Darlington Project, when Ontario Hydro was forced to reverse engineer tabular specifications for requirements and the code in order to get regulatory approval [1] to operate the Darlington station. People often cite the difficulty and cost of the original verification project when they want to dismiss tabular methods. No other formal method applied after the fact would have fared any better. Simply put, trying to apply formal methods after the fact to software that has been developed without formal verification in mind, is bound to be a difficult task regardless of the formal techniques employed.

This paper is about the Redesign Project, performed in the 1990's, in which formal techniques were integrated in the forward development process.[2] One of the essential components of the process was a Systematic Design Verification, conducted on each of the redesigned shutdown systems, SDS1 and SDS2. For each SDS, the Software Requirements Specification (SRS) made extensive use of tabular expressions to document the requirements as did the Software Design Descriptions (SDD). The Systematic Design Verification was based upon Parnas and Madey's 4-Variable Model [3] and was used to verify that the design was correct with respect to its requirements. Custom tools were developed to process the SRS and SDD documents to produce "block theorems" for the PVS theorem prover that were used to verify the majority of the functional requirements. (To complete the picture, a table-based verification process was also used to prove compliance of the code with respect to its design.)

What is a Shutdown System (SDS)?

A Shutdown System is a watchdog system that monitors reactor system parameters and shuts down (trips) the reactor if it observes "bad" behavior. The process control is performed by a separate Digital Control Computer (DCC) since it is not as critical. This design follows the principle of separation of safety and control. Each SDS consists of three computer systems (called "channels"), and the two SDS are diverse with respect to their hardware, programming languages, teams, and method of achieving a plant shutdown. The safety-critical component of each channel is a computer known as the "Trip Computer." All three Trip Computers in each SDS are identical.

Formal verification of each Trip Computer design was employed in the Darlington Redesign Project for a number of reasons. Unnecessary trips incur significant costs as coal or gas fired power plants have to be brought on line to meet demand. Without formal verification it can be difficult to make modifications and then get regulatory approval for the changes since even a relatively minor change results in another extensive and expensive round of testing and review. Failure of the system could have catastrophic consequences, but testing simply cannot cover all possible input cases for the system and there is simply too much detail for a person to catch everything by review alone.

Overview of the Darlington Redesign Project

The block diagram showing the overall structure of the SDS1 Trip Computer is shown in Figure 1. The system had 84 monitored variables (inputs) and 27 controlled variables (outputs). The software design consisted of 60 modules with 280 access programs implemented by 40,000 lines of code, including comments. Approximately 33,000 lines of code were written in FORTRAN and the remaining 7,000 were assembler.

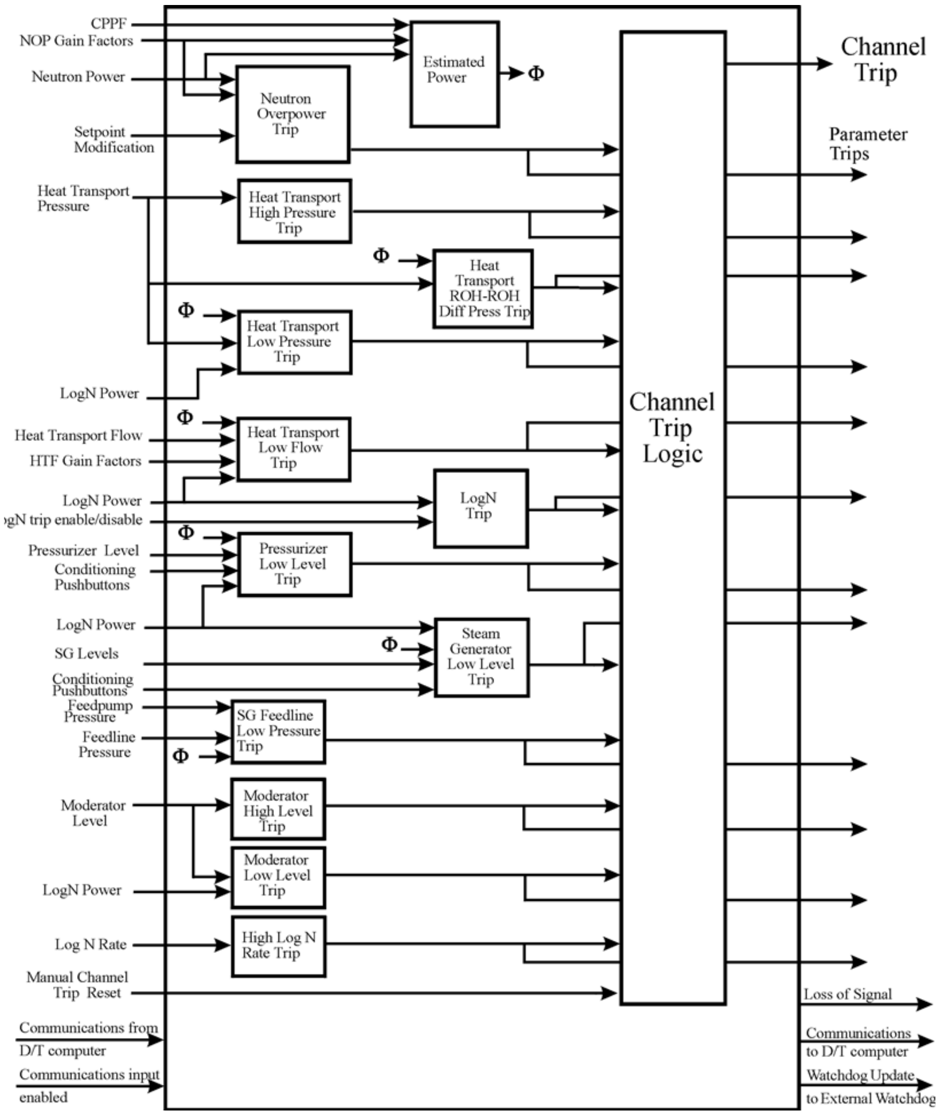

Figure 1: Block Diagram of the SDS1 Trip Computer.

The CANDU Computer Systems Engineering Centre for Excellence Standard for Software Engineering of Safety Critical Software first fundamental principle states: "The required behavior of the software shall be documented using mathematical functions in a notation which has well defined syntax and semantics."[4]

Functional specifications were used wherever possible for the following reasons:

Determinism: It is desirable to have unambiguous descriptions of safety critical behavior; *Clarity*: It is easier for domain experts and developers to understand functional requirements; *Preference*: Engineers prefer to specify precise behavior and appeal to tolerances when necessary; *Sufficient*: Functional methods are often sufficient and are easily automated.

Tolerances were taken into account on the system inputs and outputs where necessary, effectively making the specifications relational.[5] With mathematical requirements in place, it becomes possible to formally verify that the system design meets the requirements as a part of the development process.

Figure 2 shows the idealized development process together with the tools used in producing the documentation and software for the Darlington SDS Redesign. Part of the assurance case was implicitly embodied in the standard employed in the SDS Redesign, as follows:

1. The requirements are specified mathematically and checked for completeness and consistency. A hazards analysis is required to document risks and especially to identify sources of single point failures. These hazards have to be mitigated in the specified requirements.

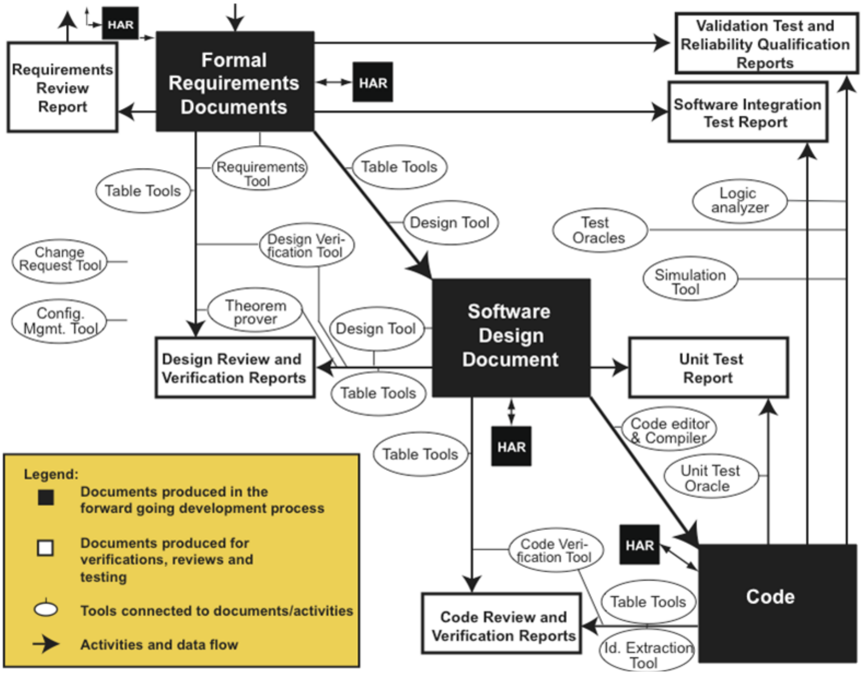

Figure 2: Idealized Development Process.

2. Compliance between requirements and software design is mathematically verified.
3. Compliance between the code and software design is verified through both mathematical verification and testing. Compliance between code and requirements is shown explicitly through testing. However, there is an implicit argument of compliance between code and requirements through the transitive closure of the mathematical verification – code to design, and design to requirements.

Some of the major benefits of using tool supported formal methods include:

- Independent checks which are unaffected by the verifier's expectations;
- Domain coverage through the use of tools that can often be used to check *all* input cases – something that is not always possible or practical with testing;
- Detection of implicit assumptions and ambiguous/inconsistent specifications;
- Additional capabilities such as the generation of counter-examples for debugging, type checking, verifying whole classes of systems, etc.

While these benefits are significant, one has to be careful to choose a formal method that results in specifications that are readable by domain experts and that can provide tool support that is integrated into the forward going software development process.

Tabular Expressions - A Useable Rigorous Method

For the redesign, the Software Requirements Specification (SRS) made extensive use of tabular expressions to document the requirements as did the Software Design Description (SDD). Ontario Hydro had some experience with tabular expressions since they were used to obtain the licence to operate Darlington in the original verification effort. Tabular methods are well suited to the documentation of the Trip Computer control functions that typically partition the input domain into discrete modes or operating regions. They were found to be readable by domain engineers, operators, testers and developers and there had been other success stories using tabular methods, such as the U.S. Navy's A-7 aircraft.[6] On the Darlington Redesign Project, tabular methods eventually showed significant benefits when used in a process with integrated tool support.

Tabular Expressions Semantics

Consider the following example table. Here each c_i is a Boolean expression, when c_i is true f returns e_i.

$$f(x_1, \ldots, x_m) = \begin{array}{|c|c|c|c|}\hline c_1 & c_2 & \ldots & c_n \\ \hline e_1 & e_2 & \ldots & e_n \\ \hline \end{array}$$

In order for the table to be *proper*, it must satisfy two properties:

1) *Disjointness:* $i \neq j \rightarrow (c_i \wedge c_j \leftrightarrow \bot)$;
2) *Completeness:* $(c_1 \vee c_2 \vee \ldots \vee c_n) \leftrightarrow \top$.

The following example illustrates why tables are effective.

$$\mathrm{f}(x, y) \overset{\mathrm{df}}{=} \begin{cases} x + y & \text{if } x > 1 \wedge y < 0 \\ x - y & \text{if } x \leq 1 \wedge y < 0 \\ x & \text{if } x > 1 \wedge y = 0 \\ xy & \text{if } x \leq 1 \wedge y = 0 \\ y & \text{if } x > 1 \wedge y > 0 \\ x/y & \text{if } x \leq 1 \wedge y > 0 \end{cases}$$

	$x > 1$	$x \leq 1$
$y < 0$	$x + y$	$x - y$
$y = 0$	x	xy
$y > 0$	y	x/y

The logical expression on the left is equivalent to the tabular expression on the right, but generally people find the table much easier to understand.

Formal Verification Used in "Certifying" Darlington

This section provides an overview of the Systematic Design Verification (SDV) procedure and corresponding tool support employed on the SDS Redesign Project. We highlight elements of the process, such as the decomposition of proof obligations, that facilitate tool support and reduce the effort required to perform rigorous design verification, including creation and maintenance of the process documents. In particular, we concentrate on the verification of functional properties utilizing tabular notation. Details of the software process and notation,[7] the decomposition of the proof obligations [8] and the complete procedure [9] can be found in the provided references.

The objective of SDV is to verify, using mathematical techniques or rigorous arguments, that the behavior of every output defined in the SDD (the design document) is in compliance with the requirements for the behavior of that output as specified in the SRS (the requirements document). It is based upon a specialization of the 4-variable model that verifies the functional equivalence of the SRS and SDD by comparing their respective one step transition functions. The resulting proof obligation in this special case is:

$$REQ = OUT \circ SOF \circ IN \tag{1}$$

and is illustrated in the commutative diagram of Figure 3. Here *REQ* and *SOF* are the one step transition functions of the requirements and design respectively.

"Vertical" Decomposition of Proof Obligations

To facilitate verification and good design, we decompose the proof obligation (3) "vertically" taking into account hardware hiding modules (see Figure 4). By creating

$$\begin{tikzcd}
\mathbf{M} \arrow[r, "REQ"] \arrow[d, "IN"'] & \mathbf{C} \\
\mathbf{I} \arrow[r, "SOF"] & \mathbf{O} \arrow[u, "OUT"']
\end{tikzcd}$$

M - Monitored Variable statespace **C** - Controlled Variable statespace
I - Input Variable statespace **O** - Output Variable statespace

Figure 3: 4-Variable Model of Parnas & Madey.

“pseudo” representations of the monitored and controlled quantities within the software, we allow for a more direct implementation of the requirements within the software, resulting in the decomposed proof obligations:

$$Abst_C \circ REQ = SOF_{req} \circ Abst_M \tag{2}$$

$$Abst_M = SOF_{in} \circ IN \tag{3}$$

$$id_C = OUT \circ SOF_{out} \circ Abst_C \tag{4}$$

Here (3) and (4) represent verification of hardware hiding modules, M_p is the pseudo-monitored variables and C_p is the pseudo-controlled variables.

Consider the following example of a hardware hiding module corresponding to the left side triangle of the commutative diagram in Figure 4. The temperature of the primary heat transport system which belongs to *M* might have a value of 500.3 Kelvin. A temperature sensor converts this to 3.4 volts which is measure by a 12-bit A/D which maps this via (part of) *IN* in a parameter with a value of 2785 counts in *I*. A hardware hiding module might then process this input corresponding to map SOF_{in}, producing a value of 500 Kelvin in the appropriate temperature variable of the software state space $M_{p.}$

Note the “wrong way” $Abst_C$ arrow - this is used to reduce the number of required abstraction functions since the output of one block comparison may be the input to

$$\begin{tikzcd}
M \arrow[rrr, "REQ"] \arrow[d, "IN"'] \arrow[dr, "Abst_M"] & & & C \arrow[dl, "Abst_C"'] \\
I \arrow[r, "SOF_{in}"] & M_p \arrow[r, "SOF_{req}"] & C_p \arrow[r, "SOF_{out}"] & O \arrow[u, "OUT"']
\end{tikzcd}$$

Figure 4: “Vertical” decomposition using Hardware Hiding.

another. This can effectively reduce by up to 1⁄2 the number of abstraction functions that the verifier is required to supply as inputs to the tools. The proof obligation (4) precludes the possibility of trivial implementations. While the invertibility of *OUT* implied by (4) is not possible in all situations, it was applicable to the majority of the safety critical requirements on Darlington.

To make the proof obligations manageable, further “vertical” decomposition of the proof obligations can be obtained by isolating outputs. In effect, we project **C** onto a single output and then restrict *REQ* to the relevant subset of **M**. The next simplifying step we make is a “horizontal” decomposition based upon dataflow.

“*Horizontal” Decomposition of Proof Obligations*

With the aid of “supplementary functions[0]” the main block comparison proof (2) can be sequentially decomposed as shown in Figure 5 into a sequence of simpler obligations of the form:

$$Abst_{vi} \circ REQ_i = SOF_i \circ Abst_{Vi-1} \qquad (5)$$

The cost of this decomposition is that the verifier must provide a cross reference between internal quantities in the requirements and design in the form of the abstraction functions: $Abst_{Vi} : V_i \rightarrow V_{ip}$. Now we see the benefit of “wrong way” arrows. The same $Abst_{Vi}$ can be used on the output of one block and then the input of the next block. We note that we only need to check invertibility of $Abst_C$ to satisfy (4) and not these internal abstraction functions.

Some very simple design rules made the mathematical verification of the code against the design much more tractable. For example, we use a *Get, Process, Set* heuristic so that if there is a dependency upon the value of an asynchronous variable (e.g. a timer), it is read when “getting” all the other inputs and its stored value is then used throughout the module. Without this heuristic it may be impossible to formally verify blocks involving asynchronous variables.

$$\begin{tikzcd}
M \arrow[r, "REQ_1"] \arrow[d, "Abst_M"'] \arrow[rrrrr, bend left, "REQ"] & V_1 \arrow[r, "REQ_2"] \arrow[d, "Abst_{V_1}"'] & V_2 \arrow[d, "Abst_{V_2}"'] & \dots & V_{n-1} \arrow[r, "REQ_n"] \arrow[d, "Abst_{V_{n-1}}"] & C \arrow[d, "Abst_C"] \\
M_p \arrow[r, "SOF_1"] \arrow[rrrrr, bend right, "SOF_{req}"'] & V_{1p} \arrow[r, "SOF_2"] & V_{2p} & \dots & V_{(n-1)p} \arrow[r, "SOF_n"] & C_p
\end{tikzcd}$$

Figure 5: Horizontal decomposition of (2) into verification blocks.

A formal method should be tightly integrated with the software development process, i.e. it is directly applied to project documents used by all parties as part of the forward development process. This includes requiring integrated tool support for the formal methods in order to make them practical. Figure 6 shows the custom tools developed to support the SDS Redesign Project.

The input for the formal verification tools was automatically generated from the word documents for the requirements (SRS), design (SDD) and verification (DVR) that were used by all project personnel.

Roughly 70% of the over 200 functional blocks from the software designs of the Redesign Project were formally verified using the SDV Tool together with the automated theorem prover PVS. The remainder of the verification blocks that did not involve straight forward block comparisons, requiring additional reasoning about the program's main execution thread and timing constraints, were handled by rigorous manual arguments.

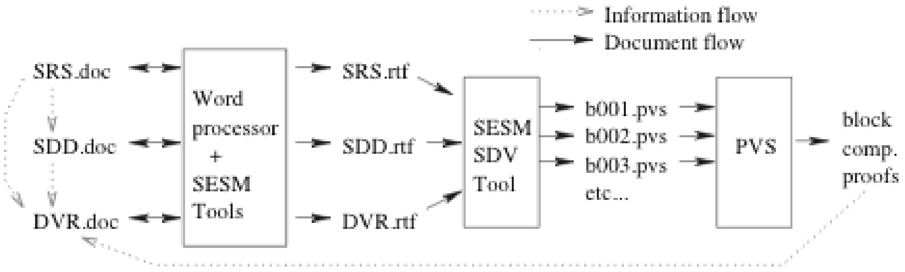

Figure 6: Tool Support for Verification of Darlington SDS Redesign.

Future of Formal Verification

Many times formal methods have been "bolted onto the side" of an existing S/W development process or applied to a project *after* the product is developed by having a "Formal Methods Guru" come and create another (formal) version of the requirements and/or design. The short term result is that errors are found and papers are written showing how good the formal methods were. Long term, the formal methods guru moves on to another source of publications, the "formal" version of documents is not understood by anyone and rots away into oblivion. It is hard enough to affordably maintain and keep one set of documents in sync with the code, let alone two.

The solution to this problem is to have one set of documents that are formal *and* readable by domain experts, easily maintained and have tool support that integrates with the company's existing software process. In the Darlington Redesign Project we avoided the “Two Model Trap” by successfully integrating tabular methods with tool support into the forward going software development process.

While the Darlington Redesign Project was a success, we only scratched the surface of what could be done. The really difficult stuff such as verification of real time properties, tolerances, sequential behavior, and numerical analysis results for fixed point arithmetic, were done manually. It was not in the budget to develop tools for all these verification aspects and the regulator did not require it. At the time that was probably the right decision but times have changed and there are now options for automatically verifying many of these properties.

The tool qualification problem

Everything that was done using the formal methods tools on the Darlington Redesign Project was also done manually too! Tools are great, but they do not buy you much credit with the regulator if they can be a single point of failure that can cause an error to go undetected. If this is the case, standards often will require the tool to be qualified to the level of the system they are being used on. This has implications for the current Model Driven Development methodologies being pushed for critical control systems since it is highly unlikely that we will be seeing a formally verified Model Driven Development framework like Matlab/Simulink any time soon.

The bad news is that you will, in all likelihood, need two different tools in order to avoid having to do verification manually, because “demonstrating soundness of the tools” will likely be difficult or impossible. The good news is that it is not as hard as you might think to knock the tool qualification requirements down a level by doing the same thing with two different tools. There is often more than one way to get a formal verification result. Domain Specific Languages (DSLs) can be used to generate code for combinations of theorem provers, SMT solvers, and model checkers. This has the added benefit of helping avoid vendor lock-in in verification tools. In order to develop a successful formal verification process integrated into the forward going software development process, consideration must be given to tool qualification requirements and how verification tasks might be performed in more than one way when you are selecting your tools and designing your development process.

Questions and Some Answers from the Darlington Experience

If tools perform automated verification in the forward process, do we really need an independent Verification team?

While the results of formal verification can be re-run and the verification tools are free from expectations of a human, the input created for the tools and the proofs themselves may require human input. Any software process that attempts to eliminate an independent verification team will have to eliminate any potential sources of developer expectations affecting the results.

Do I still need to test if I am doing formal verification?

Yes. Do not sell formal verification as a way to reduce testing, it should not. Formal verification is done on *models of the system*. Testing is done *on the real system*. However, formal verification tools can help generate test cases, e.g., an SMT solver can be used to generate tests for all cells of a table, model checkers can be used to generate longer test sequences with specific properties and the formal models can be used as oracles since many verification tools have the ability to execute subsets of their specification language. Thus formal methods can certainly help reduce the cost of testing, but they should not supplant it.

How do Formal Verification & Certification relate?

Certifying (licensing, regulatory) authorities typically audit - be it process or product based, by looking at samples or checking parts of the work. For example, the regulator on the Darlington Redesign Project (the Atomic Energy Control Board - now the Canadian Nuclear Safety Commission) audited the verification results by checking samples of the verification work – after agreeing, in principle, to the software development process rigorously documented by Ontario Hydro. Interestingly though, automated tools let you "audit everything" relatively easily just by rerunning all the tools. Further, certification of software involves much more (and sometimes less) than formal verification.

Lessons Learned

There are a number of lessons learned from the Darlington SDS Redesign Project that we need to consider.

1. Mathematically based requirements were a crucial first step. If we do not formalize the requirements, we cannot perform mathematical verifications. In the Darlington case we were not forced to do this by the regulator, but it certainly helps with certification. Clearly it is better if the formalization is done as part of the main line forward going process.
2. In selecting a formal method, making sure that the formal specifications are understandable by domain experts should be the first priority. The domain experts have to be able to read and understand all the details of the requirements. Standards prescribing formal methods typically do not require readability by domain experts, but it certainly helps with certification and increases the likelihood that the documents will be used.

3. Just as automated testing makes regression testing less time consuming and much more beneficial, formal methods tools that can be run automatically can make "regression verification" possible. On the Darlington Redesign Project, custom tools were developed to automate rerunning all of the block verification proofs in the Systematic Design Verification. These proofs could be re-run over-night any time the Requirements, Design or Verification documents changed - no matter how small or large the change. A summary of where there were broken (failed) proofs quickly highlighted the significant changes in the system.

Future research needs to consider guaranteeing semantic consistency between formal models for different provers/analysis tools so that multiple verification tools can be used for each proof obligation to eliminate the need for manual repetition of tool supported work to satisfy regulatory requirements. With the increasing use of model driven development, we also need to be concerned about the semantics of formal models used for V&V and how they compare to the semantics of the engineering modeling tools (e.g. Matlab/Simulink, MapleSim, etc).

Notes:

[1] David Lorge Parnas and Jan Madey, "Functional Documents for Computer Systems," *Science of Computer Programming* 25:1 (1995): 41–61.

[2] Alan Wassyng and Mark Lawford, "Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project," *FME 2003: International Symposium of Formal Methods Europe Proceedings* (Pisa, Italy), Lecture Notes in Computer Science 2805 (Springer, August 2003), pp. 133–153.

[3] Parnas and Madey, "Functional Documents for Computer Systems."

[4] Paul Joannou, et al., *Standard for Software Engineering of Safety Critical Software*, CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1, January 1995.

[5] Mark Lawford, Jeff McDougall, Peter Froebel, and Greg Moum, "Practical Application of Functional and Relational Methods for the Specification and Verification of Safety Critical Software," *Proceedings Algebraic Methodology and Software Technology*, 8th International

Conference, AMAST 2000, Iowa City, Iowa, USA, May 2000, Lecture Notes in Computer Science 1816 (Springer, 2000), 73–88.

6 Karen L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," *IEEE Transactions on Software Engineering* 6:1 (January 1980): 2–13.

7 Wassyng and Lawford, "Lessons learned."

8 Lawford, et al., "Practical application."

9 Greg Moum, "Procedure for the Systematic Design Verification of Safety Critical Software," CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1003-PROC Rev. 1, December 1997.

MARK LAFWORD is an Associate Professor in the Department of Computing and Software at McMaster University in Hamilton, Ontario, Canada where he helped develop the Software Engineering and Mechatronics Engineering programs. He received his Ph.D. degree in electrical engineering at the University of Toronto in 1997. From 1997 to 1998, he was with Ontario Hydro as a consultant on the Darlington Nuclear Generating Station Shutdown Systems Redesign Project, where he was a co-recipient of an Ontario Hydro New Technology Award. His research interests include software certification, formal methods and real-time systems.

ALAN WASSYNG is the Director of the McMaster Centre for Software Certification. For thirteen years he consulted for Ontario Hydro (OH). After leading one of the teams that developed "program function tables" during the original licensing of the Darlington Nuclear Generating Station, he was fortunate to continue his involvement as one of the primary people involved in the creation of the methodology used by OH for the development of safety-critical software, and helped develop the redesigned software for the Darlington Shutdown Systems. In 1995 he was a co-recipient of an Ontario Hydro New Technology Award for Development of Safety-Critical Software Engineering Technology. His research is in the development and certification of safety-critical software systems.