Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project

Alan Wassyng[*] and Mark Lawford[**]

Dept. of Computing and Software, Faculty of Engineering, McMaster University,
Hamilton, Ontario, Canada L8S 4L7

Abstract. This paper describes the lessons we learned over a thirteen year period while helping to develop the shutdown systems for the nuclear generating station at Darlington, Ontario, Canada. We begin with a brief description of the project and then show how we modified processes and notations developed in the academic community so that they are acceptable for use in industry. We highlight some of the topics that proved to be particularly challenging and that would benefit from more in-depth study without the pressure of project deadlines.

Keywords: Industrial application, specification, verification, inspection, safety critical software, experience paper.

1 Introduction

Among the reasons researchers have cited for the slow adoption of formal methods by industry are insufficient tool support, cumbersome notation, and a lack of "real world" examples (see e.g. [1]). Referring to the work on the flight software for the U.S. Navy's A-7 aircraft, one of the first well known applications of semi-formal methods to safety critical software development [2] and related works (e.g. [3]), Parnas writes: *"Although that project is still alive (now known as SCR) more than two decades later, I still see a strong lack of good examples. Other such projects would be a worthwhile investment."* [4]

This paper describes an application of formal methods in the development of safety critical software in the nuclear industry over a thirteen year period. The work makes use of tabular specifications, building upon the ideas of [2], but whereas that earlier work dealt solely with the formal specification of requirements, this paper describes an attempt to apply formal methods "all the way down" from requirements, through design, implementation and verification. The methods have been refined over use on several projects involving the specification, implementation and verification of hundreds of functions. We discuss methods of addressing the applicability of formal methods in a production setting, provide examples of how formal methods were used, and very briefly discuss the work that remains to be done to improve the utility of formal methods.

[*] Consultant to Ontario Hydro/Ontario Power Generation Inc., May 1989–June 2002
[**] Consultant to Ontario Hydro/Ontario Power Generation Inc., Feb 1997–Dec 1998

K. Araki, S. Gnesi, and D. Mandrioli (Eds.): FME 2003, LNCS 2805, pp. 133–153, 2003.

In the remainder of the paper, Section 2 describes the application of the formal methods to the Darlington Nuclear Generating Station Shutdown Systems software. Section 3 details the lessons learned over the course of applying and refining the formal methods. Open questions for future research are discussed in Section 4. Related work is discussed in more detail in Section 5, and Section 6 draws some final conclusions.

2 The Project

2.1 Application Setting

The software application described in this paper relates to the computerised shutdown system for a nuclear powered generating station. In the Canadian Nuclear Industry there is a mandatory split between plant operation and safety systems. The shutdown application is implemented on redundant hardware and consists of two independent systems, Shutdown System One (SDS1) and Shutdown System Two (SDS2). Each of these systems, SDS1 and SDS2, consists of three "channels", each channel involving a Trip Computer and a Display/Test Computer. The Trip Computers are connected to the plant sensors and contain the software that makes the decisions as to whether the plant should be shut down or not, and actually invoke the shutdown mechanism. This arrangement enables the Trip Computers to be concerned with safety issues alone. This paper is specifically about the development of software for the Trip Computers. Following a general introduction to SDS1 and SDS2 Trip Computers, we will restrict our attention to SDS1 software.

For comparison with other projects, the code produced for SDS1 consisted of approximately 60 modules, containing a total of 280 access programs. There were about 40,000 lines of code (33,000 FORTRAN and 7,000 Assembler) including comments. SDS1 has 84 system inputs (monitored variables) and 27 system outputs (controlled variables).

2.2 A (Very) Brief History

The original version of the software was developed in the late 1980s by Ontario Hydro. The regulators were not sure how to judge whether the software would perform correctly and reliably, and would remain correct and reliable under maintenance. David Parnas, as a consultant to the regulator, suggested that a requirements/design document be constructed without reference to the existing code. After validating that document, a verification process was conducted. The entire process was documented in [5]. The verification results were presented in a guided walkthrough with the regulators. At the conclusion of the walkthrough, the regulators concluded that the software was safe for use, but that it should be redesigned to enhance its maintainability.

2.3 Preparing a Strategy for the Redesign

Following the successful but painful completion of the verification and walkthrough of the Darlington shutdown systems in 1990, a series of studies were conducted by Ontario Hydro (now Ontario Power Generation Inc., - OPG). Two major conclusions were: i) The software would be redesigned using Parnas' information hiding principle [6] as the principal design heuristic. ii) As far as possible, we would include verification activities in the "forward going process". Before embarking on the Darlington Shutdown Systems Redesign, OPG set about defining a working standard for safety critical software, as well as procedures for the major steps in the software lifecycle. The Standard for Software Engineering of Safety Critical Software [7] defines the lifecycle stages, attributes of related documents, team responsibilities and team independence requirements. Procedures describing how to perform and document the Software Requirements Specification (SRS), the Software Design Description (SDD) [8], and the Systematic Design Verification (SDV) [9] were developed at that time. The procedures were tried on actual projects and have been continually refined as we have gained experience in their application.

SDS1 and SDS2 are developed independent of each other as much as is prudent. The two systems employ different shutdown technologies and run on different kinds of computers. This helps prevent common failure modes in the two systems. The system-level requirements in both SDS1 and SDS2 are known as the "Design Input Documentation" (DID), consisting of the Trip Computer Design Requirements (TCDR) [10] and the Trip Computer Design Description (TCDD) [11]. In SDS1, the TCDR and TCDD are described mathematically, and the SRS is contained within the TCDD. The SDS1 lifecycle phases and documents are shown in Fig. 1.

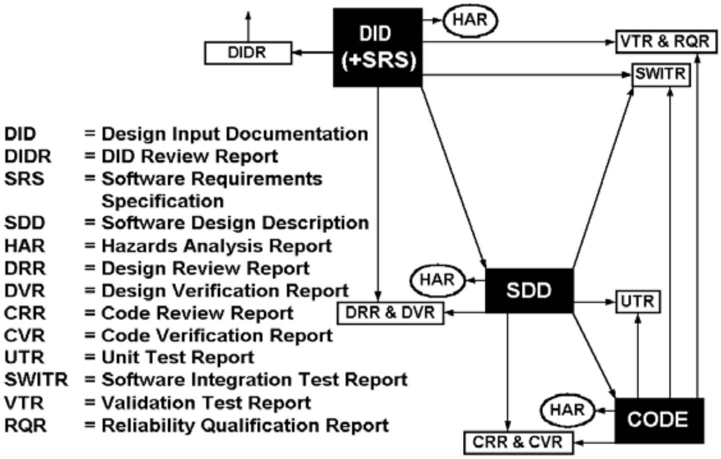

DIDR
DID (+SRS)
HAR
VTR & RQR
SWITR
HAR
SDD
UTR
DRR & DVR
HAR
CODE
CRR & CVR
DID = Design Input Documentation
DIDR = DID Review Report
SRS = Software Requirements Specification
SDD = Software Design Description
HAR = Hazards Analysis Report
DRR = Design Review Report
DVR = Design Verification Report
CRR = Code Review Report
CVR = Code Verification Report
UTR = Unit Test Report
SWITR = Software Integration Test Report
VTR = Validation Test Report
RQR = Reliability Qualification Report

Fig. 1. SDS1 lifecycle phases and documents.

2.4 System-Level Requirements

The TCDR contains the externally visible requirements for the Trip Computer, while the TCDD augments those requirements by including requirements that are specifically computer related, and by providing detailed descriptions of all fail-safe requirements.

The Trip Computer Design Requirements (TCDR). The model chosen for the TCDR was a Mills-type black-box [12]. The system, which in this case is a Trip Computer, is represented by a "black box", which relates responses generated by the system, to stimuli received by the system. The relationship is described by a mathematical function. The functional descriptions are specified in Parnas-style "function tables" [13]. If S is the set of stimuli entering the black-box, R is the set of responses exiting the black-box, and S_h is the set of stimulus history, then

$$R = f(S, S_h) \tag{1}$$

describes the behaviour of the black-box. This model was chosen for the TCDR since it's level of abstraction is close to the way in which domain experts understand relevant system behaviour.

In all our documents, stimuli are referred to as monitored variables, and responses are controlled variables. We prefix identifiers by a suitable character followed by _ so as to help identify the role of the identifier, e.g. *m_name* is a monitored variable, *c_name* is a controlled variable, *f_name* is an internal function (produced as a result of decomposing the requirements), *k_name* is a numerical constant, and *e_name* is an enumerated token. In our model, time is an implicit stimulus and every monitored and controlled variable can be time-stamped. We use the notation *m_name* to represent the current value of the monitored variable *m_name*, and m_name_{-1} to represent the previous value of *m_name*.

The functional description represented by (1) provides an idealised view of the required behaviour of the system. The TCDR recognises that this idealised behaviour can never be achieved, and so specifies a variety of tolerances within which the final implementation must operate. Apart from accuracy tolerances, the TCDR specifies timing tolerances in the form of a Timing Resolution on all monitored variables, and Performance Timing Requirements on each monitored-controlled variable pair.

It should be clear that in any real system it will not be possible to describe the behaviour represented by (1) in a single function. Instead, the requirements include a number of inter-acting functions, most of which are represented by function tables. It quickly became apparent that to have function tables widely accepted in industrial applications we needed to take into account the preferences of non-academic practitioners. The function table we used almost exclusively in SDS1 is shown below with an equivalent construct:

	Result
Condition	*name*
$Condition\_1$	res_1
⋮	⋮
$Condition\_n$	res_n

```
if $Condition\_1$ then $name = res\_1$
elsif ...
elsif $Condition\_n$ then $name = res\_n$
```

where we insist that the following two properties hold:

Disjointness: $Condition\_i \wedge Condition\_j \Leftrightarrow FALSE$, $\forall i, j = 1..n, i \neq j$, and

Completeness: $Condition\_1 \vee \ldots \vee Condition\_n \Leftrightarrow TRUE$.

We also found that we can use the table structure to emphasize the logical relationships involved. For example, we extended the table structure to include tables of the form:

		Result
Condition		*name*
$Condition\_1$	$Sub\_Condition\_1$	$res\_1.1$
	$Sub\_Condition\_2$	$res\_1.2$
$Condition\_2$		res_2
⋮		⋮
$Condition\_n$		res_n

in which adjoining cells are interpreted as being "anded".

One of the crucial challenges is to define heuristics for partitioning the system, and for finding notations that allow us to work with the partitioned system without losing intellectual control of the complete system behaviour as represented by the composition of (potentially) many function tables. One aid in this regard is the use of *natural language expressions* in the function tables. These are natural language phrases that have clear meaning to domain experts. Their use sometimes dramatically simplifies a function table. In order to retain complete mathematical rigour, all such natural language expressions are themselves defined in function tables in a separate section of the TCDR.

The following table illustrates an actual functional description in the TCDR. It evaluates the current value of the Neutron Overpower (NOP) setpoint, and clearly relies heavily on a number of natural language expressions.

	Result
Condition	*f_NOPsp*
NOP Low Power setpoint is requested	*k_NOPLPsp*
NOP Low Power setpoint is cancelled & NOP Abnormal 2 setpoint is requested	*k_NOPAbn2sp*
NOP Low Power setpoint is cancelled & NOP Abnormal 2 setpoint is cancelled & NOP Abnormal 1 setpoint is requested	*k_NOPAbn1sp*
NOP Low Power setpoint is cancelled & NOP Abnormal 2 setpoint is cancelled & NOP Abnormal 1 setpoint is cancelled	*k_NOPnormsp*

As an example "NOP Abnormal 1 setpoint is requested or cancelled" is defined by:

	Result
Condition	NOP Abnormal 1 setpoint is requested or cancelled
(m_NOPspAbn1ON = e_NotPressed) & (m_NOPspAbn1OFF = e_NotPressed)	No Change
(m_NOPspAbn1ON = e_NotPressed) & (m_NOPspAbn1OFF = e_Pressed)	cancelled
(m_NOPspAbn1ON = e_Pressed) & (m_NOPspAbn1OFF = e_NotPressed)	requested
(m_NOPspAbn1ON = e_Pressed) & (m_NOPspAbn1OFF = e_Pressed)	requested

Thus we can see that the natural language expressions effectively partition the system so that history-based requirements can be stated in much smaller tables. (Try, for example, to describe *f_NOPsp* without using the natural language expressions.) Actually, natural language expressions were developed for a different reason. They were a decisive factor in getting domain experts to buy-in to the idea of using tabular representations of requirements, since they enable those experts to read and understand the tables without undue effort, while still retaining the rigour and precision required by a formal approach. The positive effect on the decomposition of the requirements was a pleasant by-product.

The natural language expressions were carefully constructed so as to read as though they are simply text statements in a natural language, but are still reasonably easy to parse in associated software tools. Rather than "=", we use words like "is" and "are" to assign appropriate values. Clearly, in natural language expressions, the enumerated tokens representing the result values are not prefixed by "e_". The set of possible enumerated tokens is included in the natural language expression, elements being separated by "or".

The Trip Computer Design Description (TCDD). The model used in the TCDD is a Finite State Machine (FSM) with an arbitrarily small clock-tick. So, if $C(t)$ is the vector of values of all controlled variables at time t, $M(t)$ is the vector of values of all monitored variables at time t, $S(t)$ is the vector of values of all state variables at time t, and the time of initialisation is t_0, we have:

$$\begin{aligned} C(t_k) &= \mathit{REQ}(M(t_k), S(t_k)) \\ S(t_{k+1}) &= \mathit{NST}(M(t_k), S(t_k)), \text{ for } k = 0, 1, 2, 3, \ldots \end{aligned} \tag{2}$$

and the time between t_k and t_{k+1} is an arbitrarily small time, δt. Typically, state data in the TCDD has a very simple form, namely the previous values of functions and variables. We indicate elements of state data by f_name_{-1}, which is the value of *f_name* at the previous clock-tick, and similarly, m_name_{-1} and c_name_{-1}. We actually allow x_name_{-k}, x=c,f,m and k=1,2,3,..., but seldom use $k > 1$.

The description of required behaviour in the TCDD builds on the behaviour specified in the TCDR by converting all black-box representations into the FSM model, and by adding design specific behaviour that now recognises that the system will be implemented on a digital computer. This includes the introduction of fail-safe protection and self-checks.

As an example of how behaviour in the TCDD augments the behaviour in the TCDR, consider the case of momentary pushbuttons. In the TCDR, as we have already seen, the behaviour depends solely on the ON/OFF status of the pushbuttons. In the TCDD, that same behaviour takes into account that the pushbuttons have to be debounced. So the natural language expression "NOP Abnormal 1 setpoint is requested or cancelled" would be defined by:

	Result
Condition	NOP Abnormal 1 setpoint is requested or cancelled

f_NOPspAbn1ON = e_pbStuck OR
f_NOPspAbn1OFF = e_pbStuck
	requested

f_NOPspAbn1ON = e_pbNotDebounced &
f_NOPspAbn1OFF = e_pbNotDebounced
	No Change

f_NOPspAbn1ON = e_pbNotDebounced &
f_NOPspAbn1OFF = e_pbDebounced
	cancelled

f_NOPspAbn1ON = e_pbDebounced &
f_NOPspAbn1OFF = e_pbNotDebounced
	requested

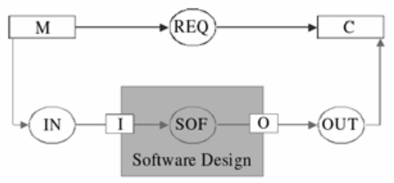
f_NOPspAbn1ON = e_pbDebounced &
f_NOPspAbn1OFF = e_pbDebounced
	requested

and *f_NOPspAbn1ON* (for example) defined by

	Results
Condition	*f_NOPspAbn1ON*
m_NOPspAbn1ON = e_NotPressed	e_pbNotDebounced
[m_NOPspAbn1ON = e_Pressed] & NOT [(m_NOPspAbn1ON = e_Pressed) Held for k_Debounce]	e_pbNotDebounced
[(m_NOPspAbn1ON = e_Pressed) Held for k_Debounce] & NOT [(m_NOPspAbn1ON = e_Pressed) Held for k_pbStuck]	e_pbDebounced
(m_NOPspAbn1ON = e_Pressed) Held for k_pbStuck	e_pbStuck

The above table illustrates the use of a generic function defined for use throughout the TCDD, namely "(condition) Held for duration", which evaluates to True when "condition" has been True for at least "duration" time. Such functions are defined precisely in the TCDD itself.

Not only does the TCDD define/redefine the behaviour specified in the TCDR, it also describes how the software will interface with the hardware. To achieve this we use Parnas' four-variable model [14]. This model relates the variables in the requirements domain to the variables in the software domain. Specifically, I and O represent the input and output variables in the software. SOF is the function that describes the software's behaviour as follows:



$$O = SOF(I^*) \tag{3}$$

$$I = IN(M) \tag{4}$$

$$C = OUT(O) \tag{5}$$

where I^* indicates the variables in I as well as the state variables ultimately dependent on I. (We already saw in (2) that $C = REQ(M^*)$, where M^* indicates the variables in M as well as the state variables.)

All the required information relating to (4) and (5) is included in the TCDD. Another important element of the TCDD is the list of Anticipated Changes.

2.5 Software Design

The software design re-organises the way in which the behaviour in the TCDD is partitioned. This is done to achieve specific goals, two of which are: i) The software design should be robust under change; and ii) On the target platform, all timing requirements will be met.

Like all other stages in the lifecycle, the SDD process and documentation are described in detail in a Procedure. The quality of the design is tied closely to a number of quality attributes defined in the Procedure. The Procedure uses these attributes to drive the design process. It also describes what documentation is required.

Information hiding principles form the basis of the design philosophy. The list of anticipated changes in the TCDD is augmented by the software developers and is used to create a Module Guide that defines a tree-structure of modules, each module having a secret and responsibility. Leaf modules represent the eventual code, and the entries for those also list the TCDD functions to be implemented in that module.

The *module cover page* describes the responsibility of the module, and lists all exported constants and types as well as the access programs for the module. The role of each access program is described in natural language, and the black-box behaviour of the program is defined by referencing the TCDD functions implemented in the access program. (This will be explained in more detail when we discuss "supplementary function tables" later in this section.)

The *module internal declarations* describes all items that are private to the module, but not confined to a single program. The *detailed design* of each program is documented using either function tables or pseudo-code (sometimes both). Pseudo code is used when a sequence of operations is mandatory and cannot easily be described in tabular format, or when specific language constructs have to be used, for example when specific assembler instructions have to be used in transfer events. The function tables used in the software design are very similar to those used in the TCDD, but are arranged vertically rather than horizontally. Variables and constants in the SDD are restricted to 6 characters because the software design had to be implemented in FORTRAN 66, the only compiler available for the hardware platform.

As an example, we provide an extract from a typical module design. It consists of the module cover page shown in Fig. 2, and the module's internal declarations, and the specification of one of its programs, shown in Fig. 3. It is likely that just a portion of a TCDD function may be implemented in an access program, or that a composition of TCDD functions may be implemented in an access program. This poses a couple of important problems. i) We reference TCDD functions to specify the black-box behaviour of an access program, and so if the access program does not implement a single, complete TCDD function, this black-box behaviour is difficult to specify; and ii) It is difficult to verify the SDD behaviour against the TCDD behaviour when the data-flow topologies of the two are different.

The way we overcome these difficulties is by use of "supplementary function tables". Imagine a pseudo requirements specification in which the data-flow topology exactly matches that in the SDD. If such a pseudo requirements specification were to exist, then verifying the SDD against the TCDD could be performed in two steps: i) Verify the SDD against the pseudo requirements specification; and ii) Verify the pseudo requirement specification against the TCDD (we need to verify only those blocks that are different from the original TCDD). The way we create the pseudo requirements specification is by piece-wise "replacing" some composition of TCDD functions by a new set of functions that have the same behaviour as the TCDD functions, but the topology of the SDD.

n.m MODULE Watchdog (1.10)

Determines the watchdog system output.

	Name	Value	Type
Constants:	(None)		

	Name	Definition
Types:	(None)	

Access Programs:

EWDOG

Updates the state of the watchdog timer Digital Output.

References: *c_Watchdog*, 'Watchdog test active'.

IWDOG

Initializes all the Watchdog module internal states and sets the initial watchdog output.

References: Initial Value, Initialization Requirements.

SWDOG

NCPARM: `t_boolean` - in

Signals to Watchdog module that a valid watchdog test request is received if NCPARM = $TRUE. Note that NCPARM is a "Conditional Output Call Argument"; calling the program with NCPARM = $FALSE has no effects on the module.

References: 'Watchdog test active'.

Fig. 2. Example module cover page

These replacement functions are represented by what we called "supplementary function tables" (SFTs).

Thus, the SFTs are developed during the forward going process, by the software designers themselves, but are not considered "proved". They are then available to aid in the mathematical verification of the software design. Rather than show a series of function tables that demonstrates the use of SFTs, we present some simple data flow examples in Fig. 4 to illustrate these points.

The top left diagram in the figure shows an extract from the TCDD. If we assume that the software design includes programs that implement the behaviour starting with an input "a" and resulting in an output "e", but partitions the behaviour differently from the TCDD, we may have a situation as pictured in the top right diagram of Fig 4.

If this is the design, the designers must have had good reasons for partitioning the behaviour this way, and must also have good reason to believe it implements the original requirements. For instance, they may have split some of the functions in the TCDD, so that the requirements can be viewed as shown in the bottom left diagram.

Finally, we regroup the functions so that they match the topology of the design as shown in the bottom right portion of Fig. 4. We can now describe f_x, f_y, f_c′, f_d′, f_z and f_e′ in tabular format, and these function tables "replace" the original f_c, f_d and f_e. The "replacement" tables are the SFTs, and they, as

n.m.1 MODULE Watchdog Internal Declaration

	Name	Value/Origin	Type
Constants:	KWDDLY	1000	`t_integer`

	Name	Definition/Origin
Types:	`t_boolean`	Global
	`t_integer`	Global
	`t_MsecTimerID`	Timer
	`t_PBId`	DigitalInput
	`t_PBStat`	DigitalInput
	`t_PosInt`	Global
	`t_TimerOp`	Timer
	`t_WDogStat`	DigitalOutput

	Name	Type
State Data:	WDGST	`t_boolean`
	WDGTST	`t_boolean`

n.m.1.1 ACCESS PROGRAM EWDOG

	Name	Ext_value	Type	Origin
Inputs:	l_CalEn	`GPBKS($PBCAL)`	t_PBStat	DigitalInput
	l_TREQD	`GCMSEC($CWDG)`	t_PosInt	Timer

	Name	Ext_value	Type	Origin
Updates:	WDGST	-	`t_boolean`	State
	WDGTST	-	`t_boolean`	State

	Name	Ext_value	Type	Origin
Output:	l_WdgClk	`SCMSEC`($CWDG, l_WdgClk)	t_TimerOp	Timer
	l_WdgDO	`SDOWDG`(l_WdgDO)	t_WDogStat	DigitalOutput

Range check assertion 1: (l_CalEn = $DBNC) OR (l_CalEn = $NDBNC)

Modes:

l_InTest	0 < l_TREQD < KWDDLY
l_NoTest	l_TREQD = 0
l_TstEnd	l_TREQD >= KWDDLY

VCT:EWDOG

	WDGTST = $FALSE		NOT(WDGTST = $FALSE)			
	WDGST= $FALSE	NOT(WDGST = $FALSE)	l_CalEn = $NDBNC	NOT(l_CalEn = $NDBNC)		
				l_NoTest	l_InTest	l_TstEnd
l_WdgClk	$CRSET	$CRSET	$CRSET	$CSTRT	$CNC	$CRSET
l_WdgDO	$WDON	$WDOFF	$WDNC	$WDNC	$WDNC	$WDNC
WDGST	$TRUE	$FALSE	NC	NC	NC	NC
WDGTST	NC	NC	$FALSE	NC	NC	$FALSE

Fig. 3. Example module internal declarations and program specification.

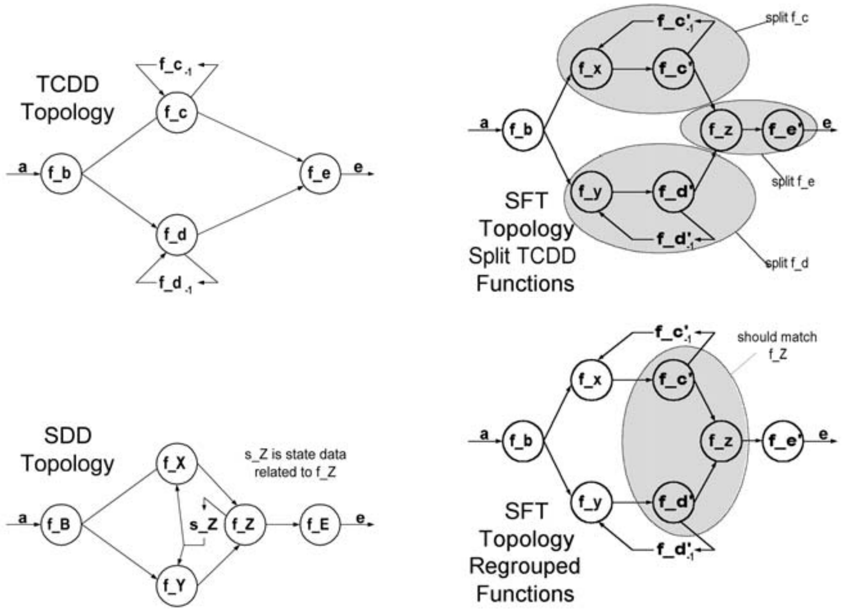
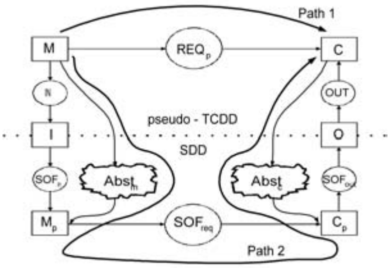


Fig. 4. Example use of supplementary function tables.

well as relevant TCDD functions, are used on module cover pages as references to the TCDD behaviour.

One final point regarding the SDD is that it is easy to "extend" the input and output mappings so that instead of I, and O, the transfer events work with M_p and C_p known as "pseudo M" and "pseudo C", constructed to be as close to M and C as possible. Then, instead of constructing SOF, the software design simply has to implement REQ, as described in the TCDD. This situation is shown graphically in Fig. 5. More details on this decomposition technique can be found in [15].



$$REQ_p(M) = Abst_c^{-1}(SOF_{req}(Abst_m(M)) \quad (6)$$

$$Abst_m(M) = SOF_{in}(IN(M)) \quad (7)$$

$$C = OUT(SOF_{out}(Abst_c(C))) \quad (8)$$

Fig. 5. Modified 4 variable model.

2.6 Software Design Verification (SDV)

There are two primary goals of the software design verification. i) Prove that the behaviour described in the SDD matches the behaviour described in the TCDD, within tolerance; and ii) Identify behaviour in the SDD that is outside the behaviour specified in the TCDD, and show that the behaviour is justified and that it cannot negatively affect TCDD specified behaviour. To accomplish the first goal, we conduct a mathematical comparison of behaviour in the SDD against the behaviour in the TCDD. This is by far the more time consuming of the two activities, since the SDD adds very little behaviour not already defined in the TCDD. To understand the verification process we need to start with the overall proof obligation. Fig. 5 shows the input and output mappings replaced by abstraction functions. Comparing Path 1 with Path 2 we see that our proof obligation is given by (6). The abstraction functions have to be verified through Equations (7) and (8).

We prove this in two steps. Firstly we prove the SDD complies with the pseudo-TCDD. Since the data-flow topology in the SDD is the same as in the pseudo-TCDD, we can show that verification can be performed block-by-block, as long as each block has equivalent inputs and outputs in the pseudo-TCDD and SDD. This piece-wise approach results in a feasible method. The second, smaller proof, pseudo-TCDD versus TCDD is dealt with on a case-by-case basis.

2.7 Automated SDD Verification

A report on manually applying tabular methods to the block-by-block SDD verification in [16], highlights the excessive effort required to perform the verification by hand. As a result, software tools were developed to extract the relevant behaviour specifications from the word processor documents that were used to describe the TCDD and SDD, into a format suitable for input to SRI International's PVS automated reasoning system [17]. In this way, a subset of all the verification blocks was proved by an automated system. Additional details on the reasons for choosing PVS, experience with the verification procedure, and the tooled tabular methods employed on the SDS Redesign project can be found in [18], [15].

2.8 Coding and Code Verification

One of the challenges we faced was to come up with a way of implementing information hiding modules in FORTRAN 66. Using labeled common statements we managed to define conceptual modules. The Coding Procedure defined quite specific rules for converting function tables into FORTRAN code. One of the decisions we made was that comments in the code usually just refer back to items in the SDD. This reinforced the concept that the SDD is a live document, and forced good traceability from code constructs to design constructs.

One of the really pleasant surprises we got, was how remarkably straightforward code verification is once the majority of the code is developed from function tables. We include comments in the code to indicate whether the code segment is based on a function table or “algorithm”, i.e. pseudo-code. The basic strategy in the code verification is to create a function table or algorithm from the code, without reference to the SDD, and then to compare that function table or algorithm with its counterpart in the SDD.

3 Primary Lessons Learned

3.1 Pure Technology Lessons

Mathematical System-Level Requirements. The use of mathematical requirements at the TCDR and TCDD level in SDS1 was controversial. A full investigation of the benefits and drawbacks still has to be performed. On the plus side, some of us thought that the precision of the requirements prompted discussion with the domain experts at a very early stage, and in most situations the required behaviour was unambiguously interpreted by specifiers, software designers and verifiers. However, there were some cases in which details of the model had not been sufficiently understood or described, and just because the requirements seemed precise it did not mean that they were unambiguous. They turned out to be ambiguous because of differences in the interpretation of the model. For example, the Timing Resolution on a monitored-controlled variable pair effectively allows a tolerance on the frequency of evaluation of an internal function (say f_a) that depends on that monitored variable, and lies on the relevant data-flow path. What about an internal function (say f_b) that depends on f_a and previous values of f_a, but not directly on the current value of the specific monitored variable? In our project, one group decided that since no specific Timing Resolution seemed to apply, infinite tolerance was applicable. Another group argued that in the absence of a Timing Resolution, no tolerance was applicable. Since this is impossible to achieve, it was argued that the Timing Resolution for the relevant monitored-controlled variable pair applied to all internal functions on that data-flow path. This apparent ambiguity led to a design that was significantly different from that expected by the verifiers. Each group used the precision of the mathematical description to support its interpretation of the requirement.

A significant practical advantage of a mathematical TCDD, was that a separate SRS was not required, thus eliminating a source of potential errors as well as the review/verification step between an SRS and (a natural language) TCDD.

Coping with Accepted Discrepancies. It sometimes appears to be worthwhile to accept seemingly innocuous discrepancies between the SDD and TCDD in so-called “non-critical” functions (a historical report, for instance), when fixing them would strain resources or severely impact the schedule. However, although the discrepancies can be identified as mathematical mismatches between the two descriptions of behaviour, it can be difficult to identify all black-box visible effects of the discrepancy. Thus, domain experts may not be able to judge the full impact of the discrepancy. The time spent on justifying these accepted discrepancies, may be better spent in trying to "correct" them if at all possible.

The Effect of Older/Slower Hardware. The best scenario for the software scheduler is typically a single, infinite loop. However, one immediate effect of slower hardware is that specific functions may need to be scheduled more frequently within that infinite loop in order to meet timing requirements. This complicates the software scheduler. Since the systematic verification process is driven by the software schedule, a more complicated scheduler has a negative effect on the design verification. In general, difficulties related to meeting timing requirements often force us to make compromises we would not have to consider in other circumstances. This situation can stretch the design and verification teams to the limits of their capabilities - and has the potential to raise tensions between the two teams. Older hardware is sometimes not just slower than newer hardware. It is quite often less reliable as it ages. Thus, to develop a fail-safe solution may necessitate the inclusion of quite sophisticated self-checks that otherwise would not be necessary.

3.2 Soft Technology Lessons

Process. The shutdown systems projects we were involved in were successful, primarily due to two factors: i) The people working on the projects were highly motivated to succeed and were technically competent; and ii) The processes used were well researched, practical, and well documented. One of the major achievements of the developers of the processes was that the procedures were designed to work together. For example, the software design process was defined to use the mathematically precise requirements to advantage, and to produce mathematically precise specifications from which the code could be developed. In addition, the design process took into account the mathematical verification that would have to be performed, as well as design reviews and eventual unit testing. It is important to note that the mathematical aspects introduced into these processes were never an end unto themselves. They always served a practical purpose that would enable the teams to produce a software product that was demonstrably safe to use.

Technology/Research Transfer. The use of function tables throughout the development process was arguably the greatest single benefit we experienced. However, it was only through perseverance that we managed to get function tables accepted to the extent they are now. The convergence to a simplified notation allowed us to represent behaviour in a way that was readily understood by all readers/developers of the major system documents. An example of an original function table used in 1989 is shown in Fig. 6.

Special symbols were used to bracket identifiers to indicate the type of identifier, for example ##enumeration tokens##, #constants#, /inputs/, etc. In

<table>
<tr><td></td><td>APPLY(OR, ((’|ai_tbl.pmp[d]| < 2745) OR ((2745 <= ′ |ai_tbl.pmp[d]|&lt;= 2795) &amp; (′ |lotrip.sgf.i[d]|<>##u##))), d=1..4) OR ((′ |ai_tbl.pmp[5]|&lt; ′ |vsps.sgf|) OR ((′ |vsps.sgf|&lt;= ′ |ai_tbl.pmp[5]|&lt;= ′ |vsps.sgf|+50) &amp; (′ |lotrip.sgf.i[5]|&lt;&gt;##u##)))</td><td>APPLY(&amp;, ((′ |ai_tbl.pmp[d]| > 2795) OR ((2745 <= ′ |ai_tbl.pmp[d]| &lt;= 2795) &amp; (′ |lotrip.sgf.i[d]|=##u##))), d=1..4) &amp; ((′ |ai_tbl.pmp[5]|&gt;′|vsps.sgf|+50) OR ((′ |vsps.sgf|&lt;= ′ |ai_tbl.pmp[5]|&lt;= ′ |vsps.sgf|+50) &amp; (′ |lotrip.sgf.i[5]|=##u##)))</td></tr>
<tr><td>|lotrip.sgf.n|′</td><td>1..5</td><td>0</td></tr>
</table>

Fig. 6. Example of original tabular notation

addition, $'x$ was used to indicate the value of x immediately prior to the evaluation of the table (pre-value of x), and x' indicated the post-value of x. Such tables were used successfully in the verification project in 1989, but when we tried to use these tables to present requirements behaviour, readers were quick to tell us that they probably would not use them. Domain experts especially found them close to unreadable.

So what did we do to function tables that made them suddenly acceptable to professional software developers and engineers responsible for determining system requirements?

One quite trivial change that worked wonders was - we simply turned the tables sideways. It was suggested by someone new to function tables who said something like “I would relate to the tables better if we read them left-to-right, just as we do normal text”. We tried that and it worked. Not only did domain experts prefer them written that way, but it turned out that it was easier to format them on standard $8\frac{1}{2}'' \times 11''$ pages. At least this was true for the requirements documents, which tend to have single results, and seem to need more rows (disjoint primitives) than columns (nested conditions).

Another change we made was to remove all the bracketing symbols as well as the pre and post indicators. It turns out that in our simple tables it is quite obvious whether a variable is used as a pre-value or a post-value. The result/output of a table is a post-value, and we allow only pre-values of variables in any cells of the tables. Note that by pre- and post-values, we mean with respect to invocation of that particular table, not with respect to the clock-tick of the underlying model. We are always explicit with respect to the clock-tick.

Rather than bracket identifiers with special symbols, we decided to prefix identifiers with a character (occasionally two) followed by an underscore as outlined in Sec. 2.4. This arrangement is far easier to read, easier to remember, and still retains the effect that alphabetic sorting of identifiers also sorts identifiers into classification groups. After years of working with professional software engineers on these projects, we feel qualified to say that this aspect of formal methods, i.e. arriving at notation and presentation methods that industrial participants will accept, is about as important as the technology in the formal methods themselves. Without this effort, even the best of formal methods will probably not be used in practice.

A side benefit of using function tables was that we could easily identify conditions that lead to a safe-state. This is important in the software design, where we want to specify the behaviour in such a way that it can be coded into structures that produce safe behaviour even in the case of hardware malfunctions. By always having the safe-state in the right-most column, it was clear to designers and coders alike.

As we indicated earlier, one of the guiding principles for the redesign of the Darlington Shutdown Software, was Parnas' "Information Hiding" [6]. As usual, there is a world of difference between understanding and appreciating the principles versus putting those principles to work and implementing them in a software design process - it took us about two years through the first iteration. An important difference between what we put into practice compared with the original description was that most of the top level "secrets" that were to be encapsulated in a module, came from requirements that were likely to change, not from design decisions as described in [6]. It is important to note the enormous effort that had to be expended in order to use the ideas that came from academia, even though the ideas were well accepted (indeed, quite famous) within the academic community, twenty years after they were first introduced. Even today, it is not universally appreciated that information hiding is significantly more than data abstraction. The principles of information hiding can be used to develop a software design process that eventually results in the decomposition of the design into data abstractions. The importance of this process is that the resulting data abstractions are constructed in such a way that, if in the future we need to change the design to incorporate a "likely change", the changes will be confined to a single module, or, at most, a very small number of modules. This is a crucial selling point of the technology to senior management. Most people, quite rightly, would be skeptical of any claim to be able to cope "easily" with all future changes. What information hiding promises and delivers, is that future changes that are considered likely and are used to drive the decomposition of the software design, will be relatively easy to implement and verify.

Rationale. Most engineers already appreciate the importance of documenting rationale for design decisions. Even before the start of the redesign of the Darlington shutdown system software, the responsible group at Ontario Power Generation conducted a separate project to document the rationale for the existing software. Both the TCDR and TCDD contain appendices that record changes made from previous "in use" systems. In addition, rationale is documented for each function and natural language expression, whenever possible. However, rationale for some decisions that were made fifteen to twenty years ago has been lost. In at least one case, that lack of rationale led us to include a requirement that was later shown to be problematic.

Software Tools Limitations and Suggested Improvements. While the use of standard word processors created documents that were easily readable by a human, producing machine readable output from the documents was much more difficult. While particular projects may standardize on a given word processor (and version), maintenance and revision of software obviously becomes an issue when projects are tied to a specific vendor and revision of a product.

The utility and adoption of mathematically sound tools has been hampered by a lack of well defined (consistent) open standards and associated support tools. Significant resources are sometimes required to develop tools that are not directly related to a company's core competencies.

One lesson we learned was that developing tools to cope with mundane, but necessary and time-consuming tasks, can significantly reduce the effort required to perform those tasks, and thus reduce the schedule time that has to be allocated to those tasks. These tools may not present the same excitement and challenge to their developers as more sophisticated tools might, but their importance to the project can be even more substantial.

3.3 Political Lessons

People. Experienced project managers know that the success of their project is tied directly to the ability, knowledge and attitude of the people working on the project. Building reliable software applications of a non-trivial size requires team members to communicate with each other and with other teams. Often two teams are in an intentionally adversarial role - but both teams should be clear that they share the responsibility for achieving a high quality software application. "Scoring points" at the expense of an individual or team is counterproductive to the project. Since formal methods are not yet commonplace in industrial software projects, it is important that all project personnel buy-in to the specific methodology being used. It is healthy to argue over different approaches and processes, but compromises are often necessary, and continual second-guessing of such decisions, after the decision has been finalised, is usually destructive.

Don't Compromise Normal Best Practice. Ontario Power Generation wisely adopted a "defence-in-depth" policy for their safety critical software projects. Over the years, software professionals and researchers have developed heuristics for reducing defects at different stages of the software lifecycle. For example, at the coding stage, there are reasonably well accepted coding guidelines that take into account the strengths and weaknesses of the language, that help us to avoid situations that have been shown to be more error prone. It is sometimes tempting to argue that since we are applying so much more rigour to the entire lifecycle, we can relax these "best practice" heuristics. Our experience is that it is a mistake to follow this path. It is a good idea to remind ourselves periodically that mathematicians have been known to make mistakes.

4 Open Questions

Nothing is ever perfect. Software development is far from that ideal. Also, there are tremendous financial and schedule pressures on projects such as those described in this paper. It is always useful to examine what we did and try to determine what areas may be strengthened in future projects. We do not have the space to discuss this in detail, but wanted to mention them briefly, almost as a final lesson-learned.

Probably the most challenging issue throughout the projects has been to find ways to specify and verify functional and performance timing requirements. So far, our methods have dealt much better with the logic aspects of behaviour than with anything that involves timing. Nothing we have seen in the literature has stood out as clearly better than what we did. Another challenge is the way in which we should deal with selfchecks. In the case of hardware malfunction, it is important that the application be left in a safe-state if at all possible. The current software design achieves this at significant cost. More research in this area is warranted. Much progress has been made already on automating verification activities [3]. Extending these techniques to deal with more cases, especially those involving time-dependent behaviour, should be a high priority. In general, we should now be able to build significantly improved software tools. Finally, we are getting to the stage where we have sufficient experience with these processes to see how we can extend them to help with non safety critical software.

5 Related Work

Software practitioners have clearly indicated the need to automate routine tasks in order to effectively and reliably develop software. The application of formal methods similarly needs to become a largely automated process with tools of even better quality than those used for building and testing software. Knight *et al.* hypothesize that by incorporating formal methods tools into existing software packages such as off the shelf office suites and other software engineering tools, formal methods might be able to overcome their lack of "superstructure" and become more widely used in industry [19]. The experience described here and in [18] supports this conjecture.

While applications of tool supported formal methods to industrial examples have been previously described in e.g., [20], [21], such case studies typically focus on a specific aspect, such as requirements analysis. Our work differs from other successful industrial integrations of formal methods into the software engineering process such as [22] since in our case i) The formal methods were applied to the entire system, and ii) the formal methods documentation and the main project documentation are one and the same. It should be pointed out though that the application described here resulted in roughly one fifth the number of lines of code as the application in [22] and it was not distributed.

6 Conclusion

The use of formal methods in these projects was successful, primarily because of the quality of the personnel involved, and the fact that, by the nature of the problem, OPG was prepared to put sufficient resources into making it successful. It was successful also because it was practical. The level of rigour was commensurate with the task. The formal methods approach was pervasive, but was never allowed to develop beyond its usefulness. Tremendous progress was made in taking ideas from the formal methods research community, and making them practical. However, it is our opinion that the formal methods research community should be trying much harder to make their methods more practical in the first place. Even after thirteen years though, there are significant aspects that have not been completely solved - timing issues for example. We coped with them adequately, but to make these techniques more cost effective, we have to have much more general solutions. Finally, the methods are just too costly without reliable, comprehensive tool support.

Acknowledgements

The work presented in this paper represents the combined efforts of many current and former employees of Ontario Power Generation and AECL, including: Glenn Archinoff, Dominic Chan, Rick Hohendorf, Paul Joannou, Peter Froebel, David Lau, Elder Matias, Jeff McDougall, Greg Moum, Mike Viola, and Alanna Wong. The authors would like to thank Rick Hohendorf and Mike Viola for helping to obtain permission from OPG to publish this work. The FM 2003 reviewers provided invaluable feedback. Finally we would like to acknowledge David Parnas. This work represents the successful application of many of his ideas regarding software engineering.

References

[1] Saiedian H. (ed.): An invitation to formal methods. IEEE Computer **Apr** (1996) 16–30

[2] Heninger, K.L.: Specifying software requirements for complex systems: New techniques and their applications. IEEE Transactions on Software Engineering **6** (1980) 2–13

[3] Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR*: A toolset for specifying and analyzing software requirements. In: Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998. Volume 1427 of LNCS, Springer (1998) 526–531

[4] Parnas, D.L.: Software design. In Hoffman, D., Weiss, D., eds.: Software Fundamentals: Collected Papers by David L. Parnas. Addison-Wesley (2001) 137–142

[5] Archinoff, G.H., Hohendorf, R.J., Wassyng, A., Quigley, B., Borsch, M.R.: Verification of the shutdown system software at the Darlington nuclear generating station. In: International Conference on Control and Instrumentation in Nuclear Installations, Glasgow, UK, The Institution of Nuclear Engineers (1990)

[6] Parnas, D.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15** (1972) 1053–1058

[7] Joannou, P., et al.: Standard for Software Engineering of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1 (1995)

[8] McDougall, J., Lee, J.: Procedure for the Software Design Description for Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1002-PROC Rev. 1 (1995)

[9] Moum, G.: Procedure for the Systematic Design Verification of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1003-PROC Rev. 1 (1997)

[10] Wassyng, A.: Darlington NGD Shutdown System Trip Computer Software Redesign Project, SDS1, Trip Computer Design Requirements Procedure. Technical Report NK38-MAN-68200-003, Rev. 04, Ontario Hydro (2001)

[11] Wassyng, A.: Darlington NGD Shutdown System Trip Computer Software Redesign Project, SDS1, Trip Computer Design Description Procedure. Technical Report NK38-MAN-68200-001, Rev. 03, Ontario Hydro (2001)

[12] Mills, H.D.: Stepwise refinement and verification in box-structured systems. Computer **21** (1988) 23–36

[13] Janicki, R., Parnas, D.L., Zucker, J.: Tabular representations in relational documents. In Brink, C., Kahl, W., Schmidt, G., eds.: Relational Methods in Computer Science. Advances in Computing Science. Springer Wien New York (1997) 184–196

[14] Parnas, D.L., Madey, J.: Functional documents for computer systems. Science of Computer Programming **25** (1995) 41–61

[15] Lawford, M., McDougall, J., Froebel, P., Moum, G.: Practical application of functional and relational methods for the specification and verification of safety critical software. In Rus, T., ed.: Proceedings Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 2000. Volume 1816 of LNCS, Springer (2000) 73–88

[16] Viola, M.: Ontario Hydro's experience with new methods for engineering safety critical software. In: SAFECOMP'95: The 14th International Conference on Computer Safety, Reliability and Security, Belgirate, Italy, Springer (1995) 283–298

[17] Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Transactions on Software Engineering **21** (1995) 107–125

[18] Lawford, M., Froebel, P., Moum, G.: Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Accepted for publication in May 2002. `http://www.cas.mcmaster.ca/~lawford/papers/` (To appear)

[19] Knight, J.C., Hanks, K.S., Travis, S.R.: Tool support for production use of formal techniques. In: 12th International Symposium on Software Reliability Engineering (ISSRE 2001), Hong Kong, China, IEEE Computer Society (2001)

[20] Heitmeyer, C., Kirby, Jr., J., Labaw, B., Archer, M., Bharadwaj, R.: Using abstraction and model checking to detect safety violations in requirements specifications. IEEE Transactions on Software Engineering **24** (1998) 927–948

[21] Crow, J., Di Vito, B.L.: Formalizing Space Shuttle software requirements: Four case studies. ACM Transactions on Software Engineering and Methodology **7** (1998) 296–332

[22] Hall, A., Chapman, R.: Correctness by construction: Developing a commercial secure system. IEEE Software **Jan/Feb** (2002) 18–25