International Journal on Software Tools for Technology (2006) 8(4/5): 337–354
DOI 10.1007/s10009-005-0209-6

SPECIAL SECTION ON THE INDUSTRIALIZATION OF FORMAL METHODS: A VIEW FROM FORMAL METHODS 2003

Alan Wassyng · Mark Lawford

Software tools for safety-critical software development

Published online: 23 September 2005

Abstract We briefly present a software methodology for safety-critical software, developed over many years to cope with industrial safety-critical applications in the Canadian nuclear industry. Following this we present discussion on software tools that have been used to support this methodology, and software tools that could be used, but have not been used for a variety of reasons. Based on our experience, we also present and motivate a list of high-level requirements for tools that would facilitate the development of safety-critical software using the presented methods, together with a small number of tools that we believe are worth developing in the future.

1 Introduction

Software development is maturing. With that maturity has come the realisation that any particular development methodology will not succeed unless it is well-supported by software tools. There are many diverse software tools available to software developers. Most of these tools are targeted at particular tasks. Not many of them provide comprehensive support for a particular methodology. When they do, they can be extraordinarily successful. For example, although UML [28] had no semantic basis, it proved to be extremely successful in industry. The success of UML, to a large extent, can be attributed to the comprehensive tool support that was available for it.

Software development for safety-critical systems is generally viewed as costly and time consuming. Software tools are always touted as a means to combat the labour intensive nature of software development, and safety-critical software development in particular. We are convinced that appropriate tool support can, indeed, help us produce safety-critical software at reduced costs. What is sometimes lost in the "better, cheaper, faster" mantra of tool proponents is that for safety critical systems, the most important thing is that tool support should also make it easier to see and demonstrate the quality of the software–for the producer, customer and regulator alike.

This paper concentrates on software tools for safety-critical software and is based on many years of experience of developing safety-critical software applications, with and without tool support. The emphasis is on software tools that support our methods.

The remainder of the paper is organized as follows. Section 2 provides an overview of the software engineering methodology we have used, in enough detail to understand the role and application of tools. Section 3 describes the tools that we believe are essential for making the method practical. Following this, Sect. 4 comments on additional tools that would be useful, but currently have not been implemented or have not been integrated with our methods. A discussion of regulatory requirements on supporting tools for safety-critical software in Sect. 5 helps to illustrate why developing tools for safety-critical software presents some unique challenges. Section 6 provides a list of high-level requirements for tools, motivated by our experience, and the last two sections discuss future tools and draw conclusions.

2 A safety-critical software methodology

The software methodology we have used on safety-critical projects has been described previously in [20, 33]. It is based primarily on Parnas' descriptions of a "Rational Design Process" [23, 26] and makes extensive use of tabular expressions [13, 32]. The methodology was developed at Ontario Hydro, now Ontario Power Generation (OPG) Inc. The primary applications were the two independent Shutdown Systems for the Darlington Nuclear Generating Station in Ontario, Canada.

A. Wassyng (✉) · M. Lawford
The Software Quality Research Laboratory
Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1
E-mail: wassyng@mcmaster.ca

The Software Cost Reduction (SCR) Method developed at the U.S. Naval Research Laboratory (NRL) [7–9], is also based on Parnas' Rational Design Process, and also makes extensive use of tabular expressions. NRL focused quite early on the production of software tools. OPG focused primarily on the production of the software applications and documentation that were essential to the nuclear regulator granting operating licences for the Darlington Nuclear Generating Station. Tool support was developed where and when necessary. This is due to the nature of a research versus a production environment. NRL researchers have focused on extending capabilities of different tools rather than applying them end to end on the critical path of production for a major project. OPG focused on what was needed for regulator approval and what was immediately beneficial to the project. However, those involved in the OPG projects now have the experience to apply to the development of software tools to aid that methodology. We also have the luxury of being able to learn from the extensive tool development conducted by NRL.

2.1 Requirements

The software requirements for both of the Darlington shutdown system are described mathematically. For one of the shutdown systems, the system level requirements are described mathematically, and contains the Software Requirements Specification. This is the system we will describe in this paper. The life-cycle phases and documents are shown in Fig. 1. This figure does not include the requirements development process, its starting point is the requirements document.

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 clarify 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.

The requirements model we use is a finite state machine with an arbitrarily small clock-tick. The finite state machine is assumed to describe idealised behaviour, i.e. results are produced instantaneously. 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, we can define functions R (requirements) and NST (next state) as follows:

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

where the time of initialisation is t_0, and the time between t_k and t_{k+1} is an arbitrarily small time, δt. Typically, state data at the requirements level 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 also describe how the software will interface with the hardware. To achieve this we use Parnas' 4 variable model [24].

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. *REQ* is the function that describes the requirement behaviour. We already saw in (1) how R relates C to M and S. The state data, S, is an encapsulation of the history of M, so Fig. 2 and Eqs. (2)–(5) should be interpreted as including relevant history.

The vast majority of the requirements are specified using tabular expressions. As an example, the definitions of the "Neutron OverPower Parameter Trip and Sensor Trips" are shown in Fig. 3. Roughly speaking, a sensor trip occurs when a function that depends on a sensor value goes out of safe range. A parameter trip depends on a function of related sensor trips. To give a better idea of the "horizontal function tables" we use to define the required behaviour, some of the other functions required for the evaluation of this trip are shown in Fig. 4 (without their related lists of inputs and other such information). These examples also demonstrate

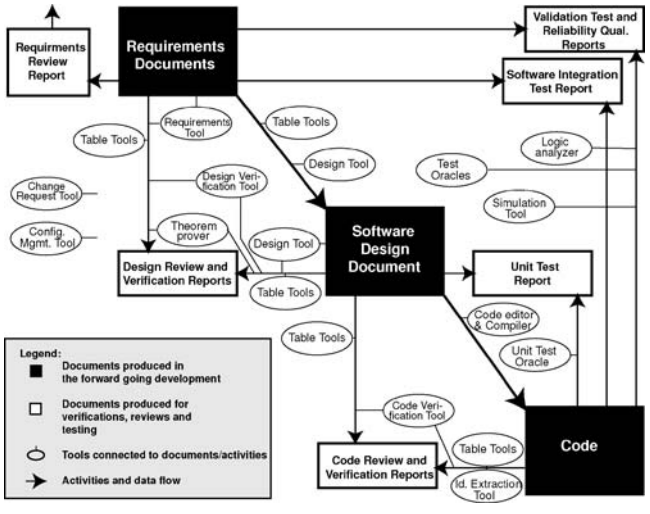
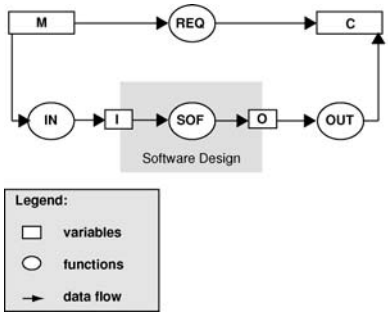


Fig. 1 Life-cycle phases, documents and tools



$$C = REQ(M) \tag{2}$$
$$O = SOF(I) \tag{3}$$
$$I = IN(M) \tag{4}$$
$$C = OUT(O) \tag{5}$$

Fig. 2 The 4 variable model

2.3.9.1 Neutron Overpower Parameter Trip

2.3.9.1.1 Inputs/Natural Language Expressions

Input	NL Expression	Reference
f_NOPsentrip$_i$, i=1,...,18	-	2.3.9.2.4

2.3.9.1.2 c_NOPparmtrip

Condition	*Result* **c_NOPparmtrip**
Any (i ∈ 1,...,18)(f_NOPsentrip$_i$ = e_Trip) {*Any NOP sensor is tripped*}	e_Trip
All (i = 1,...,18)(f_NOPsentrip$_i$ = e_NotTrip) {*All NOP sensors are not tripped*}	e_NotTrip

2.3.9.2 Neutron Overpower Sensor Trips

Determines whether there is an NOP sensor trip, which is used to determine whether there is an associated parameter trip.

2.3.9.2.1 Inputs/Natural Language Expressions

Input	NL Expression	Reference
f_NOPsp	-	2.3.9.3.3
f_NOPGain$_i$, i=1,...,18, k_CalNOPHiLimit, k_CalNOPLoLimit, k_NOPOffset, m_NOPai$_i$, i=1,...,18	Calibrated ith NOP signal, i=1,...,18	2.4.12

2.3.9.2.2 Initial Value

Name	Initial Value	References
(f_NOPsentrip$_i$)$_{-1}$, i=1,...,18	e_Trip	TCDR

2.3.9.2.3 Output Units/Type

Output	Type
f_NOPsentrip$_i$, i=1,...,18	y_trip

2.3.9.2.4 f_NOPsentrip$_i$, i=1,...,18

{For each i=1,...,18}

Condition	*Result* **f_NOPsentrip$_i$**
f_NOPsp ≤ Calibrated ith NOP signal {*NOP signal$_i$ is now in the trip region*}	e_Trip
f_NOPsp - k_NOPhys < Calibrated ith NOP signal < f_NOPsp {*NOP signal$_i$ is now in the deadband region*}	(f_NOPsentrip$_i$)$_{-1}$
Calibrated ith NOP signal ≤ f_NOPsp - k_NOPhys {*NOP signal$_i$ is now in the non-trip region*}	e_NotTrip

Explanatory notes:
The reference to "TCDR" is to a system level requirements document.
"y_trip" is a type defined to be (e_NotTrip, e_trip).

Fig. 3 Specifications of the neutron overpower parameter trip and sensor trips in the requirements

Condition	*Result* **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

To help the reader understand this table: The natural language expressions "NOP ... setpoint is requested or cancelled" are dependent on operator inputs via momentary pushbuttons. The "safest" setpoint (the lowest numeric value) is k_NOPLPsp, the one for Low Power. Clearly, the behaviour reflects a prioritised ordering, from Low Power to Normal.

{For each i=1,...,18}

Condition	*Result* **Calibrated i^{th} NOP signal**
x_i < k_CalNOPLoLimit	k_CalNOPLoLimit
k_CalNOPLoLimit ≤ x_i ≤ k_CalNOPHiLimit	x_i
k_CalNOPHiLimit < x_i	k_CalNOPHiLimit

where x_i = [(m_NOPai$_i$ - k_NOPOffset) * f_Gain$_i$] + k_NOPOffset

This function simply clamps the value of "Calibrated i^{th} NOP signal" between the constants k_CalNOPLoLimit and k_CalNOPHiLimit.

Fig. 4 Examples of tabular expressions related to the NOP trip in the requirements

the use of *natural language expressions*, that are used to make the tables more intuitive for domain experts who have to validate the requirements. The natural language expressions also help to partition the requirements, and all natural language expressions are defined mathematically. One of these is shown in the example in Fig. 4. We also use natural language comments in the tables to aid readers. These comments are italicised, and are enclosed within braces.

The advantages of tabular expressions have been widely documented (e.g. [11, 25]), and two of the primary reasons tables are so useful in the specification of requirements are the completeness and disjointness criteria that can be enforced easily. Table tools make this enforcement even easier, especially in the case of more complex conditions.

2.2 Software design

The software design process is one of the most labour intensive stages in the life-cycle. The resulting document, the Software Design Document, is arguably the most crucial document in the project. Common wisdom has it that the requirements documents determine the success of the project. It is true that if the requirements documents are not "correct", complete, understandable and unambiguous, the project is likely to fail. However, the design must exhibit all of these attributes, but also must be decomposed in such a way as to enhance the future maintenance of the software, including subsequent changes to the hardware platform. Especially in safety-critical projects, the design must also be verifiable (this typically means that the design must be mathematically verifiable against the requirements), and must be sufficiently detailed as to provide a complete specification of behaviour from which code can be developed and against which it subsequently can be verified and tested. Thus, while the requirements document is crucial for initial success of the project, the software design is critical to the long term success of the project.

Surprisingly, other than the table tools (and, occasionally, documentation layout tools), we have not used any design specific software tools in this phase of the life-cycle.

The software design re-organises the way in which the behaviour in the requirements 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 life-cycle, the software design process and documentation are described in detail in a Procedure (a project standard that specifies the process and mandatory documentation format). 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.

4.23 MODULE NPParTrip (1.2.1)

Provides the current NOP parameter trip status to drive the NOP parameter trip output.

	Name	Value	Type
Constants:	(None)		
	Name	**Definition**	
Types:	(None)		

Access Programs:

EPTNP

Determines the current NOP parameter trip status and posts the parameter trip output state to the DigitalOutput module.

References: *c_NOPparmtrip.*

GPTSNP

return: t_boolean

Returns the current NOP parameter trip status of the NOP trip parameter. A return value of $TRUE or$FALSE indicates that the parameter is tripped or not tripped respectively.

References: *c_NOPparmtrip.*

IPTNP

Initialises the NPParTrip module internal states.

References: *Initial Value - c_NOPparmtrip.*

4.23.1 MODULE NPParTrip Internal Declaration

	Name	Value/Origin	Type
Constants:	KNUMNP	Global	t_integer
	Name	**Definition/Origin**	
Types:	t_boolean	Global	
	Name	**Type**	
State Data:	PTSNP	t_boolean	

Fig. 5 Example module cover page and internal declarations

Information hiding principles form the basis of the design philosophy. A list of anticipated changes documented in the requirements is augmented by the software developers and is used to create a Module Guide that defines a tree-structure of modules. Each module hides a secret (a requirements or design decision that is likely to change in the future), and fulfils a responsibility. Leaf modules represent the eventual code, and the entries for those also list the specific requirements functions to be implemented in that module. The interfaces to the modules are defined by the externally visible entities, primarily exported types, constants and programs. Externally accessible programs are known as "access programs". The *module cover page* (Fig. 5) 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 requirements functions encapsulated 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, e.g. 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 requirements. Over the years we found that horizontal condition tables worked well for requirements documents, since they read naturally from left to right, and requirements typically involve functions that have single outputs, but many predicates. However, the functions in the software design often have more than one output, and fewer predicates, so we use vertical condition tables in the design document.

As examples, we provide extracts related to the requirements functions that were displayed in Fig. 3. The module cover page for module NPParTrip is shown in Fig. 5. The module's internal declarations, and the specification of its programs, are shown in Fig. 6. The complete sensor trip module is not shown in order to save space. However, Fig. 7 shows the access program that evaluates f_NOPsentrip. If we compare Fig. 7 with the required behaviour of f_NOPsentrip in Fig. 3, it is obvious that the design computes an "additional" quantity. This is a common occurrence, since the designers have to be concerned with efficient implementation of the behaviour, which is not a consideration at the requirements level. In this case, STSNP maps to f_NOPsentrip, and STINP maps to latched values of the sensor trips, that are specified in a different function in

4.23.1.1 ACCESS PROGRAM EPTNP

	Name	Ext_value	Type	Origin
Inputs:	l_ST	GSTNP(l_ST)	ARRAY 1 TO KNUMNP of t_boolean	NPSnrTrip
	Name	Ext_value	Type	Origin
Updates:	(None)	-	-	-
	Name	Ext_value	Type	Origin
Outputs:	l_TrpDO	SDONP(l_TrpDO)	t_boolean	DigitalOutput
	PTSNP	-	t_boolean	State

Local Terms:

l_NoSTrp	(ALL i=1..KNUMNP)(l_ST[i] = $FALSE)

VCT:EPTNP

	l_NoSTrp	NOT(l_NoSTrp)
l_TrpDO	$FALSE	$TRUE
PTSNP	$FALSE	$TRUE

4.23.1.2 ACCESS PROGRAM GPTSNP

	Name	Ext_value	Type	Origin
Inputs:	PTSNP	-	t_boolean	State
	Name	Ext_value	Type	Origin
Updates:	(None)	-	-	-
	Name	Ext_value	Type	Origin
Outputs:	lGPTSNP	-	t_boolean	Func

VCT:GPTSNP

	TRUE
GPTSNP	PTSNP

4.23.1.3 ACCESS PROGRAM IPTNP

	Name	Ext_value	Type	Origin
Inputs:	(None)	-	-	-
	Name	Ext_value	Type	Origin
Updates:	(None)	-	-	-
	Name	Ext_value	Type	Origin
Outputs:	PTSNP	-	t_boolean	State

VCT:IPTNP

	TRUE
PTSNP	$TRUE

Fig. 6 Example module program specifications. (VCT means “vertical condition table”)

the requirements document. Variables and constants in the design are restricted to six characters because the software design had to be implemented in FORTRAN 66, the only compiler available for the hardware platform.

It is likely that just a portion of a requirements function may be implemented in an access program, or that a composition of requirements functions may be implemented in an access program. This poses a couple of important problems. (i) We reference requirements functions to specify the black-box behaviour of an access program, and so if the access program does not implement a single, complete requirements function, this black-box behaviour is difficult to specify; and (ii) it is difficult to verify the design behaviour against the requirements behaviour when the data-flow topologies of the two are different.

The way we overcome these difficulties is through the use of *Supplementary Function Tables* (SFTs). Imagine a pseudo-requirements specification in which the data-flow topology exactly matches that in the software design. If such a pseudo-requirements specification were to exist, then verifying the design against the requirements could be performed in two steps: (i) verify the design against the pseudo-requirements specification; and (ii) verify the pseudo-requirement specification against the original requirements (we need to verify only those blocks that are different from the original). The way we create the pseudo-requirements

4.27.1.1 ACCESS PROGRAM ESTNP

	Name	Ext_value	Type	Origin
Inputs:	l_STSP	GSPNP	t_SnrValue	SPSlcNP
	l_SVal	GNPCAL(l_SVal)	ARRAY 1 TO KNUMNP of t_SnrValue	NPSnrCal
	Name	Ext_value	Type	Origin
Updates:	STINP	-	ARRAY 1 TO KNUMNP of t_boolean	State
	STSNP	-	ARRAY 1 TO KNUMNP of t_boolean	State
	Name	Ext_value	Type	Origin
Outputs:	(None)	-	-	-

Assertion 1: l_STSP > KTHYNP > 0

Range check assertion 1: (KSPLNP ≤ l_STSP ≤ KSPHNP)

Range check assertion 2: ((ALL i=1..KNUMNP)(KAIMIN ≤ l_SVal[i] ≤ KAIMAX))

VCT:ESTNP

FOR i=1 TO KNUMNP	l_SVal[i] ≤ (l_STSP - KTHYNP)	l_SVal[i] < l_STSP AND l_SVal[i] > (l_STSP - KTHYNP)	l_SVal[i] ≥ l_STSP
STSNP	$FALSE	NC	$TRUE
STINP	NC	NC	$TRUE

Fig. 7 Extract of the sensor trip access program. (NC means No Change)

specification is by piece-wise "replacing" some composition of requirements functions by a new set of functions that have the same behaviour as the original requirements functions, but the topology of the design. These replacement functions are represented by what we called 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. 8 to illustrate these points.

The top left diagram in the figure shows an extract from the requirements. 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 requirements, we may have a situation as pictured in the top right diagram of Fig 8.

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 requirements, 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. 8. In this example, we have assumed that s_Z is a compound data structure, and mutually exclusive elements are used in f_X and f_Y. We can now describe $\mathsf{f\_x}$, $\mathsf{f\_y}$, $\mathsf{f\_c'}$, $\mathsf{f\_d'}$, $\mathsf{f\_z}$ and $\mathsf{f\_e'}$ in tabular format, and these function tables "replace" the original $\mathsf{f\_c}$, $\mathsf{f\_d}$ and $\mathsf{f\_e}$. The "replacement" tables are the SFTs, and they, as well as relevant requirements functions, are used on module cover pages as references to the required behaviour.

One final point regarding the software design 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 requirements. This situation is shown graphically in Fig. 9. More details on this decomposition technique can be found in [20, 33].

2.3 Software design verification

The main activity in the software design verification is the comparison of tabular expressions in the design with corresponding expressions in the requirements. The results of the verification are documented in the Design Verification Report. This is complicated by the fact that the requirements describe idealised behaviour, and then specify tolerances on that behaviour so that an implementation becomes feasible. The tolerances can take the form of accuracies on monitored and controlled variables, and timing tolerances that allow for finite processing time and the fact that analogue signals

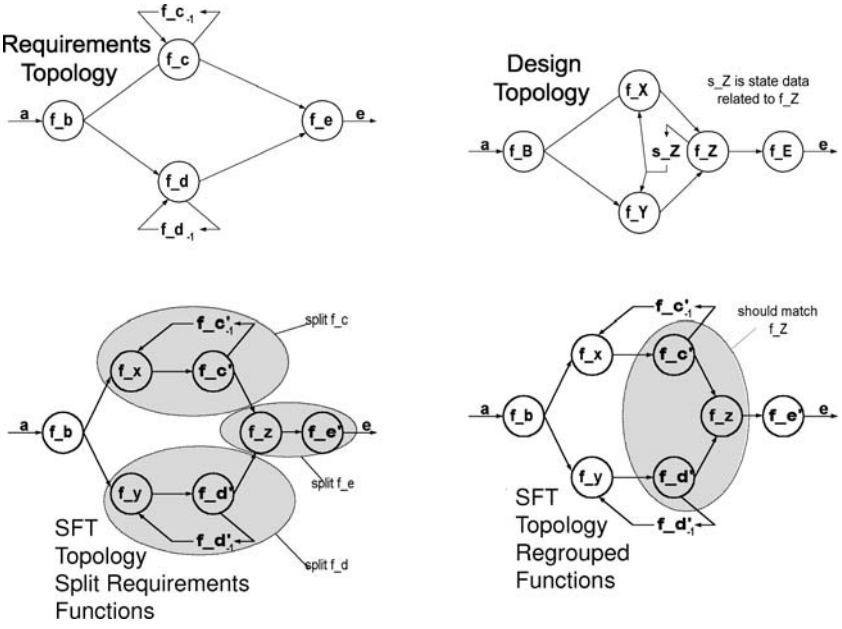
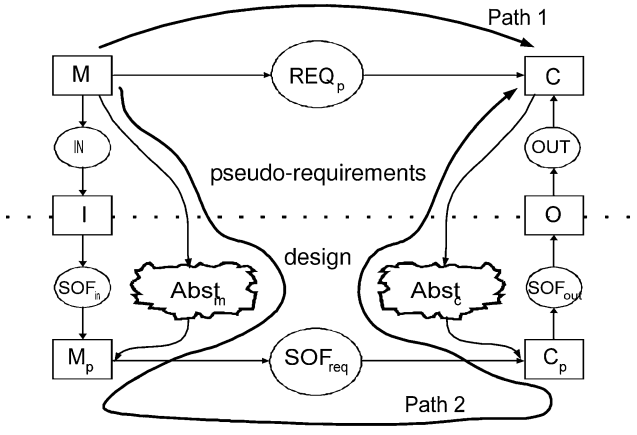


Fig. 8 Example use of supplementary function tables



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

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

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

Fig. 9 Modified 4 variable model

must be sampled. As we will see in Sect. 3, the design verification process invites the extensive application of software tools.

There are two primary goals of the software design verification. (i) Prove that the behaviour described in the design matches the behaviour described in the requirements, within tolerances; and (ii) identify behaviour in the design that is outside the behaviour specified in the requirements, and show that the behaviour is justified and that it cannot negatively affect the required behaviour. To accomplish the first goal, we conduct a mathematical comparison of behaviour in the design against the behaviour in the requirements. This is by far the more time consuming of the two activities, since the design adds very little behaviour not already defined (at a higher level of abstraction) in the requirements. To understand the verification process we need to start with the overall proof obligation. Figure 9 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 Eqs. (7) and (8).

We prove (6) in two steps. Firstly we prove the design complies with the pseudo-requirements. Since the data-flow topology in the design is the same as in the pseudo-requirements, we can show that verification can be performed piece-wise. This approach results in a feasible method and is now quite well supported by software tools (see Sect. 3).

The SFTs developed during the design phase help to reduce the work-load on the verifiers. Although there is no obligation on the designers to verify that their SFTs are correct, their intimate knowledge of the design and requirements should lead them to construct accurate SFTs. Thus, the second proof, pseudo-requirements versus the original requirements, is "smaller" than the proof of (6). These proofs typically have to be tailored to the particular design decisions that were made, and are dealt with on a case-by-case basis. For example, with reference to Fig. 8, we would need to prove that the composition of f_x and f_c' is equivalent to f_c (including the relevant state data), that the composition of f_y and f_d' is equivalent to f_d (including the relevant state data), and that the composition of f_z and f_e' is equivalent to f_e.

2.4 Coding, code verification and testing

Conceptually, coding is not much different from coding in non safety-critical projects, except that it is likely that the coding guidelines are more precise and more restrictive. Almost all the behavioural details are already described in the software design.

Most code functions are produced directly from tabular expressions in the design. Relatively few functions are coded from pseudo-code. The coding procedure imposes relatively strict guidelines on how to implement code from tabular expressions and from pseudo-code. Comments in the code are used to document whether the code was developed from tables or from pseudo-code. For all those functions described by tabular expressions in the design, the code verifiers manually develop function tables from the code and then compare them with the tables in the design. Since the pseudo-code is already very close, in abstraction level, to the code, the pseudo-code implementations are verified by direct comparison and logical argument. An example of code is presented in Fig. 10. This is an extract from the FORTRAN implementation of the access program EPTNP in module NPParTrip that was shown in Fig. 6.

Finally, there are all the typical testing procedures: unit testing, software integration testing, validation testing, and statistically driven random testing. Space does not permit an in-depth discussion of the testing procedures and documents.

3 Essential tools for our methodology

Figure 1 shows most of the tools we have used with our methods and project documents. Not surprisingly there is a good match between these tools and those we consider essential for the cost effective application of the methodology. This section will examine each of the tools in a little more detail.

To understand the current suite of tools used at OPG it is necessary to know that all project documentation has to be available in Microsoft Office Word format. MS Word is the corporate approved word processing platform.

3.1 Generic tools

It is crucial that *all* project documents and tools be maintained under configuration control. This means that there must be an easily accessible/usable configuration manager. In some early projects, configuration management errors caused costly delays. This prompted OPG management to mandate comprehensive configuration management support. An e-mail front-end was developed to support PVCS, a commercially supported configuration management system. With the e-mail interface, all project personnel had access to project documents under configuration management. We

```
      SUBROUTINE EPTNP
      IMPLICIT COMPLEX (A-Z, $)
C-----------------------------------------------------------------------
C PARAMETERS
C     (NONE)
C-----------------------------------------------------------------------
C GLOBAL CONSTANTS OR "EXPORT" CONSTANTS FROM OTHER MODULES
      INSERT ERRHDX
      INSERT GLOBAX
C-----------------------------------------------------------------------
C INTERNAL VARIABLES/CONSTANTS AND EXPORT CONSTANTS OF THE MODULE
      INSERT NPPARV
C-----------------------------------------------------------------------
C ACCESS/LOCAL PROGRAM DATA DECLARATIONS
C     Declaration for SDD Inputs.
C     < l_ST : ARRAY 1 TO KNUMNP OF t_boolean >
      INTEGER LLST(18)
C     Declaration for SDD Outputs.
C     < l_TrpDO: t_boolean >
      INTEGER LLTRPD
C-----------------------------------------------------------------------
C EXPLICIT FUNCTION DECLARATIONS FOR STANDARD LIBRARY USED
C     (NONE)
C-----------------------------------------------------------------------
C EXPLICIT FUNCTION DECLARATIONS FOR ACCESS PROGRAMS USED
C     (NONE)
C-----------------------------------------------------------------------
C ACCESS/LOCAL PROGRAM CONSTANT DEFINITIONS
C     PLN number
      INTEGER  KPLN
      DATA  KPLN/701/
C     < SDD Input: l_ST = NPSnrTrip.GSTNP(l_ST) >
      CALL GSTNP(LLST)
C*****RANGE-CHECK NOTE:  ****************************************
C     Removed range-check note to save space to fit this extract
C*****END RANGE-CHECK NOTE  ************************************
C     Variable initialization.
C     < SDD output: l_TrpDO >
      LLTRPD = $TRUE
      PTSNP = $TRUE
C --- < VCT EPTNP (A1.2) > ---------------------------------------
      IF (.NOT. ((LLST(1) .EQ. $FALSE) .AND.
     +   (LLST(2)  .EQ. $FALSE) .AND. (LLST(3)  .EQ. $FALSE) .AND.
     +   (LLST(4)  .EQ. $FALSE) .AND. (LLST(5)  .EQ. $FALSE) .AND.
     +   (LLST(6)  .EQ. $FALSE) .AND. (LLST(7)  .EQ. $FALSE) .AND.
     +   (LLST(8)  .EQ. $FALSE) .AND. (LLST(9)  .EQ. $FALSE) .AND.
     +   (LLST(10) .EQ. $FALSE) .AND. (LLST(11) .EQ. $FALSE) .AND.
     +   (LLST(12) .EQ. $FALSE) .AND. (LLST(13) .EQ. $FALSE) .AND.
     +   (LLST(14) .EQ. $FALSE) .AND. (LLST(15) .EQ. $FALSE) .AND.
     +   (LLST(16) .EQ. $FALSE) .AND. (LLST(17) .EQ. $FALSE) .AND.
     +   (LLST(18) .EQ. $FALSE))) GO TO 20000
C     < l_NoSTrp >
C     See RANGE-CHECK NOTE (1.a)
C        < SDD output: l_TrpDO >
         LLTRPD = $FALSE
         PTSNP  = $FALSE
         GO TO 29999
20000 CONTINUE
C     Range check on <l_ST>
C     See RANGE-CHECK NOTE (2.a)
      IF (.NOT.((LLST(1) .EQ. $TRUE) .OR.
     +    (LLST(2)  .EQ. $TRUE) .OR. (LLST(3)  .EQ. $TRUE) .OR.
     +    (LLST(4)  .EQ. $TRUE) .OR. (LLST(5)  .EQ. $TRUE) .OR.
     +    (LLST(6)  .EQ. $TRUE) .OR. (LLST(7)  .EQ. $TRUE) .OR.
     +    (LLST(8)  .EQ. $TRUE) .OR. (LLST(9)  .EQ. $TRUE) .OR.
     +    (LLST(10) .EQ. $TRUE) .OR. (LLST(11) .EQ. $TRUE) .OR.
     +    (LLST(12) .EQ. $TRUE) .OR. (LLST(13) .EQ. $TRUE) .OR.
     +    (LLST(14) .EQ. $TRUE) .OR. (LLST(15) .EQ. $TRUE) .OR.
     +    (LLST(16) .EQ. $TRUE) .OR. (LLST(17) .EQ. $TRUE) .OR.
     +    (LLST(18) .EQ. $TRUE))) GO TO 29998
C     < NOT(l_NoSTrp) >
         GO TO 29999
29998 CONTINUE
C     See RANGE-CHECK NOTE (2.b)
C        ErrorHdler.SFAT($FERNG, KPLN)
         CALL SFAT($FERNG, KPLN)
29999 CONTINUE
C --- < END VCT EPTNP (A1.2) > -----------------------------------
C     < SDD output call: DigitalOutput.SDONP(l_TrpDO) >
      CALL SDONP(LLTRPD)
C
      INSERT STHRD
      RETURN
      END
```

Fig. 10 Code extract from EPTNP (NOP sensor trip)

refer to the complete system as the *Configuration Management Tool*.

Another essential generic tool is a database for recording and tracking potential changes. These may be changes in process or in an actual life-cycle deliverable. They may arise as a result of discovering a defect of any type, but may also be simple suggestions for a potentially better way of doing something. At OPG these were called *Document Change Requests*. We like the way OPG removed the "defect" or "bug" connotation in Document Change Requests. In order to emphasise the fact that the tool deals with all change requests, we will refer to the tool simply as the *Change Request Tool*.

The *Table Tools* enable users to generate function tables and check tabular expressions for completeness and disjointness. They are also capable of translating between a limited number of table formats, namely: structured decision tables, horizontal condition tables, and vertical condition tables [1]. These tools are not as extensive as those produced at McMaster [15] or at NRL [8, 9]. Our experience has shown that some tables are more readable than others [33], and OPG standardised on just a few types of tabular expressions. In addition, the *Table Tools* were specifically designed to interface to MS Word.

3.2 Life-cycle phase-oriented tools

These tools were developed to support the major development life-cycle phases and their resulting documents as described in Sect. 2. Section 2 as well as the brief descriptions below of the existing tools can therefore be seen as an overview of the technical requirements for these tools.

The *Requirements Tool* is capable of directing the author by generating the required documentation section titles. The tool dynamically constructs a database of identifiers and uses the database in a number of consistency checks. It also links to the *Table Tools* so that tabular expressions can be checked for completeness and disjointness. Finally, the tool is capable of extracting tabular expressions so that they can be translated for use in a theorem prover (PVS).

The *Design Tool*, like the *Requirements Tool*, guides the author of the software design by generating templates. It also dynamically generates a database of identifiers, links to the *Table Tools*, and extracts and translates tabular expressions for use in PVS.

The *Design Verification Tool* has proved to be the most used tool in the suite.

An experience report on manually applying tabular methods to the block-by-block software design verification appears in [31]. The report cites the excessive amount of time required to perform the verification by hand as a major short fall of the method. As a result, OPG and its partners undertook efforts to automate the verification procedure.

Figure 11 provides a graphic overview of the relationship between the documents and tools employed in the verification process. The word processor, augmented with the tools

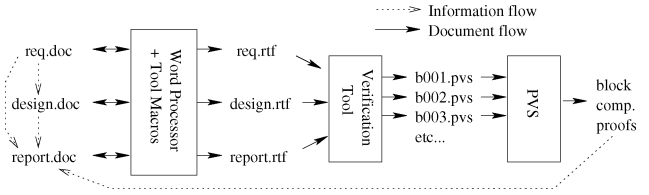


Fig. 11 Tools and documents of the design verification process

described above, was used to create and check, first the software requirements, then the software design and finally the design verification report. The tools provide basic completeness and consistency checks of the requirements, design and design verification report documents that can be run offline on an entire document or invoked interactively via a macro from within the word processor to check individual tabular function definitions as they are created.

The design verification report provided the cross reference between the requirements and design inputs, outputs and functions, and defined the abstraction functions that are part of the block decomposition of the proof obligations. These parts of the design verification report were manually generated by the verifier with guidance provided by the requirements and design documents. The word processor was then used to create Rich Text Format (RTF) versions of these three documents that become input for the *Design Verification Tool*. At the time the tools were developed RTF provided the closest thing to a "standard" input format for the tools independent of the word processor used to create the documents. For each verification block the *Design Verification Tool* produced an input file for SRI International's PVS automated reasoning system [22]. Each file contained the theorems, and associated function and type definitions that need to be proved for the block as required by the verification procedure. Due to space limitations, we refer the reader to [18, 20] for examples of the types of proof obligations passed to PVS in the design verification of the Darlington shutdown systems.

Although the PVS specification language and interactive proof environment have their own steep learning curve, the verifiers require only a small subset of PVS' capabilities to perform the verification. Additionally, by designing the tools to employ standard word processors for document creation, we have ensured that no other team members require knowledge of the underlying proof system. While the verification procedure currently makes use of only a fraction of PVS' capabilities, integrating the tools with PVS provides the opportunity to increase the scope of the computer assisted verification to include real-time properties [3, 6, 19] and functional properties involving tolerances [20]. Additional reasons for choosing PVS were its existing support for tabular methods [21] integrated with theorem proving and model-checking, and the abilities of its extensive type-checking capabilities to be used to detect software errors [29].

Additional details on experience with the verification procedure and the tooled tabular methods employed in it can be found in [18, 20].

The original *Code Verification Tool* was hardly used. Its primary purpose was to again guide the author by generating templates for the code verification documentation. However, we created a tool that examined the code, extracted identifier information, and classified identifiers in the code in much the same way that module specifications are documented. This actually accounts for a huge percentage of the work we normally have to do in verifying code against RDP (Rational Design Process) modules. "All" that is left after that is to describe the behaviour of a code access program as a tabular expression (e.g.), so that the behaviour can be compared with the corresponding behaviour in the SDD. Right now we have not yet managed to automate the extraction of behaviour into tabular expressions. However, our experience is that compiling the identifier information takes roughly 10 to 15 times the amount of time it takes to analyse the code behaviour. This means that the tool we constructed performs the mundane but time-consuming tasks, and leaves placeholders for the tabular expression (or pseudo-code if appropriate) and the subsequent comparison with the design behaviour.

Finally, a *Simulation Tool* was built to provide a platform for testing. Automated oracles and test suites were used with the simulator. In cases where precise timing was essential, individual tests were conducted with the aid of a logic analyser.

One important point to note about the tool suite that was developed – the tools were designed to work together. For example, the *Requirements Tool* was used not only in the requirements phase, but also in the software design verification phase. The *Table Tools* were designed so that they interfaced with all the other tools in which function table manipulation and checking may be required.

4 Nice-to-have tools for our methodology

The tools suggested in this section may exist in other environments, or may simply be tools that we are confident can be built. Top of the list in tools that would be useful to us is *Model Checking*. Model checking has proven itself in many circumstances, and has become a staple component in the SCR toolset. The underlying model in both versions of the software requirements documents at OPG is a finite state machine. It should therefore be reasonably straightforward to link model checkers to the current tool suite. This would give the requirements teams capabilities to check the requirements behaviour over multiple clock ticks of the finite state machine, whereas our current capability is limited to one-step transitions.

Another tool that would be useful is a *Module Decomposition Tool* that would be used during software design. One of the crucial tasks facing the software designers is the decomposition of the behaviour presented in the requirements, into a modular design based on information hiding principles. During this decomposition phase, designers successively decompose modules into smaller and smaller modules, using lists of likely and unlikely changes to the specified behaviour to define secrets [23, 33] , as well as a set of software quality attributes. To understand the effect of the decomposition, the designer has to trace exactly which requirement(s) (function tables or parts of a function table, usually) each module will be responsible for, and also what dependencies there will be between modules (the *Uses Hierarchy*, for example). Since the requirements are completely and formally documented, a tool could be used to show the designer the effect of a change in the module structure, i.e. what requirements functions are encapsulated in each module; what constants are used in each module; and what other modules are used by each module. This would clearly be of benefit to the designer at this stage of the process. It could also perform completeness and consistency checks, e.g. all requirements are encapsulated in one, and only one, module. If the requirement function is only partially "implemented" in a module, this should be noted. (This is what SFTs were developed to cope with, see Sect. 2.2.)

Many of the traditional tools would prove useful in our software development. For example, *Profilers*, if available for the languages and platforms we use, would facilitate both coding and testing. *Code Analysis Tools*, such as *CodeSurfer* [2] could facilitate code inspections and reviews.

There are other tools that we would like, but right now do not have the necessary theory developed to build the tools. One of these, a tool to construct tabular expressions, is discussed in Sect. 7.

5 Regulatory aspects

Regulatory considerations may have a tremendous impact on the tools that are, and can be used for safety-critical software development. Clearly, depending on the country and application domain, regulators may play a definitive role in the success or failure of any particular safety-critical project.

In most cases, we believe that regulators will prove to be in favour of the extensive use of software tools. Their use can make feasible approaches that are otherwise too costly. For those who have been involved in such projects, it is reasonably obvious that software tools have the capacity to improve the quality of the developed application. There are many aspects that are more reliably analysed/implemented by automated tools than by error prone humans. However, it is likely that regulators will require that software tools used in safety-critical applications should be *qualified* before being used. In the Canadian nuclear industry, there are usually just two ways of qualifying such tools. (i) Prove that the tool was developed with the same rigour as is necessary for the applications to which it is to be applied. (ii) Make a case that through extensive use of the product, its reliability has been "proven".

This introduces a rather serious "catch-22" situation for any specially developed software tools. These tools will probably have relatively low usage profiles, so building a case on extensive usage is not possible. Building tools to the

same standards as the actual products to which it will be applied is often too costly if the number of such products is likely to be small. There is one way out of this dilemma, which is the method we employed in most cases. We can build the tools, qualify them as best we can, but not use them to the exclusion of manual techniques. This negates much of the cost benefit of using the tools, but still realises the quality benefit discussed earlier.

Another approach may be to develop more than one version of the tool and compare their outputs. This approach is often prohibitively expensive, and does not guarantee that common-mode errors in the tool have been avoided.

In other industries, the required reliability of each tool is tied to the extent to which the output of that tool can be checked – by manual or automated means.

The development of in house and/or use of third party CASE tools to facilitate specification, formal verification, and testing of safety-critical software has become an increasingly common phenomena. Unfortunately 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. Faced with a lack of third party software in the area of high performance computing software, a US President's Information Technology Advisory Committee has recommended an open source software approach [27]. In the case of safety-critical software, this approach has the added benefit of making the entire verification and testing process transparent to regulators, allowing them to investigate the internal workings of any tools employed on a project. While the regulators may not posses the skills to perform such an investigation, it does allow the regulator to employ independent experts to evaluate the tools on their behalf. Any reported findings could be made public and used as a basis for further evaluation and use of the tools in a regulated environment.

In the long term, we believe that the developers of methodologies will have to also deliver qualified software tools that facilitate the implementation of that methodology. Those developers will need to convince regulators in many countries and many application domains, that those tools have been developed, verified and tested to the degree commensurate with their usage. Using open source tools and standards, the regulators (and the public) can convince themselves of the quality of the tools by "looking under the hood" and seeing how they work. Sharing the source of the safety-critical tools they have developed, companies can distribute development costs and increase tool usage profiles. Taking the open source concept one step further, organisations could also share formal analyses of source code.

6 Requirements for tools

This section presents primarily high-level requirements for tools to support the safety-critical methods we use. The emphasis is on requirements that deal with those aspects we did not already deal with in Sects. 3 and 4.

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 [16]. The experiences described in [18, 33] support this conjecture.

While applications of tool supported formal methods to industrial examples have been previously described in e.g., [10], such case studies typically focus on a specific aspect, such as requirements analysis, and typically involved some reverse engineering of previously developed requirements documents. As a result these methods usually were not part of the production software engineering process. To be truly successful in industry, formal methods tools cannot just be "bolted on to the side" of an existing software development process.

This section describes what we believe to be important high-level requirements for tools that would support our methodology. These requirements do not necessarily reflect any actual requirements of the tool suites currently in use. Rather, they are the requirements that we believe should apply, now that we have experience in both developing and using generic software tools as well as tools specifically oriented towards safety-critical application development.

The requirements are documented by listing the requirement itself, followed by rationale for that requirement in italics.

6.1 General requirements

The following requirements apply to all phases of the development life-cycle.

- The tools should form a comprehensive suite, designed to interface with each other, with complementary functions. The integrated suite shall provide automated support for all phases of the software development life-cycle.

  We gain real advantages if we can make assumptions about the inputs to a tool. One of the lessons we have learned is that generalising tools so that a wide variety of inputs is allowed, typically reduces rather than enhances the scope of the tool. An example of how to take advantage of our knowledge in one phase to help a subsequent phase is the Module Decomposition Tool discussed in Sect. 4.
- The tools shall generate documentation in as universal a document format as possible, while ensuring that:
  - The documentation format has adequate support for tables.

  - The documentation format has adequate support for graphics.
  - The documentation format has adequate support for mathematical notation.

  Many safety-critical projects have a long life-cycle. It is likely that being tied to a specific word processor or proprietary document format will prove to be an obstacle at some stage of the development or maintenance. We experienced a change from WordPerfect to Word that was time-consuming and thus costly.
- It shall be possible to construct parts of the documentation manually in such a way that the reader cannot tell that the section has been created manually. There should be a mechanism for automatically including such manually created sections in future versions of the document.

  Project deadlines should not be missed simply because a tool fails to operate adequately. We worked on one project where a tool could print only the complete document, not page-ranges of the document. This caused a problem when a few pages had to be corrected immediately before a deadline. If manual edits cannot be incorporated automatically in future versions of the document, then manual edits will prove to be of very little use.
- The tools shall maintain a database of identifiers common to all phases of the development life-cycle. This includes identifiers defined in manually created sections of documents.

  Integrated tools require an integrated database of identifiers.
- The tools shall ensure that entities are not defined in more than one place. If a definition is required in more than one place, the definition shall be stored once only and copied to all applicable locations.

  This requirement makes sense for manually implemented methods as well. Anyone who works in software development should be sensitive to this issue.
- Tools that aid in the construction of a project document shall guide the author so that all mandated sections of the document are completed, and must be flexible enough to allow additional sections if not prohibited by governing procedures.

  A friendly nudge from the tool that a mandatory section has not been completed is useful, as long as it is not too intrusive. Flexibility is important, since we have found that users of the tools are rightly frustrated if the tool insists on completion of specific section in order if the completion order is not important. Try as we may, it has proved to be impossible to predict the exact structure of many of the project documents. For example, the specification of the serial communication to and from the shutdown computer had to take into account a non-standard multiplexor that required detailed descriptions that were not foreseen when the requirements procedures were developed. If additional sections are not allowed, document preparation grinds to a halt.
- Tools that aid in the construction of project documents shall be capable of spell-checking the document. The database of identifiers shall be used to augment the spell-check dictionary.

  We have found that an effective check for misused identifiers is to use a spell-checker.
- The suite of tools shall facilitate the seamless integration of the configuration management tools described below.

  Version control is of utmost importance in any software project. It is far more reliable to have the tools control the configuration management than it is to rely on people to manually invoke the configuration management tools. We worked on a project in which someone spent significant time verifying an old version of a document. There was a configuration management system in place, but humans are error prone - especially in time pressured situations.

6.2 Configuration management tool

- The tool shall facilitate management of "baselines".

  A baseline represents a milestone and specifies a collection of documents and their particular revisions as representative of that milestone. Many configuration management systems control revisions of individual documents but do not provide a baseline capability. Without baselines it is becomes difficult to track that requirements 1.5 was used to produce Software Design 3.2 and Code 2.5, let alone all the test suites, verification and review documents that are associated with those specific documents.
- The tool shall timestamp documents entered into the database.

  This is an obvious requirement, but has to be stated.
- The tool should have at least the capabilities of commercially available configuration management tools such as PVCS. It is conceivable that a front-end for an existing commercial tool may be sufficient. *Many commercially available configuration management tools have evolved to the stage where they do an excellent job of configuration management. A front-end such as the e-mail interface developed by OPG may supplement the features of the commercial system, and may hide some of the complexity of the system by making just a necessary subset of the features available to general users. In addition, including mathematical theories associated with the formal verification would be beneficial. Existing systems such as MAYA, a tool to support formal verification and incremental development of software [4, 12], and related work on collaborative content management [30] and version control for structured mathematical knowledge [17], extend revision control systems such as CVS [5], to help guarantee global correctness of the results of formal reasoning about systems.*

6.3 Change request tool

- The tool shall facilitate the entry and subsequent tracking of the analysis and disposition of change requests.

This is the obvious role of the tool.

- The tool shall link related change requests. This can be achieved both through algorithms that detect commonalities and dependencies in change requests, and also by allowing users to insert links manually.

  This is useful from two points of view: (i) A similar change request may already have been entered. (ii) An earlier change request that has been analysed may affect the analysis of the current change request.
- The tool shall timestamp the individual components of each change request. (It is anticipated that change requests will consist of a number of sections, e.g. problem, analysis, implementation, follow-up.)

  Post analysis of the development/maintenance of the application would be enhanced.
- The tool shall record the name of the person responsible for each component of the change request.

  This is important for any follow-up activities.
- The tool shall enforce the recording of the life-cycle phase in which the author of the change request was involved when the potential change was "discovered".

  This is important both for quality control and for future improvements to the processes.
- The tool shall maintain separation between the statement and analysis of the problem.

  It is a mistake to try to analyse the problem at the time that the potential problem is first documented. We know of one instance in which a change request was not filed because the person who uncovered the potential problem was (erroneously) persuaded that there was no problem.
- The tool shall facilitate the attachment of any kind of project document (or extract of that document) to any component of a change request.

  The best explanation of a problem, presentation of analysis and sometimes even the disposition of the change request, in many cases, is an extract from a relevant document. The tool must facilitate storing such attachments so that the link to the change request is obvious.

6.4 Requirements tool

- The tool shall tag every requirement (a function table may be tagged as a single requirement), and the tag shall be visible in printed versions of the document. Section numbers may be used as tags.

  It is essential that the software design, design review and verification, testing and change requests be able to refer to relevant requirements.
- The tool shall guide the author to include rationale and/or references for each requirement.

  Most engineers already appreciate the importance of documenting rationale for design decisions. Before work started on the current version of the Darlington shutdown system software, the responsible group at OPG conducted a separate project to document the rationale for the existing software. The requirements documents now 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 one case, that lack of rationale led us to include a requirement that added pushbutton debouncing to the communications link-enable pushbutton, with the same debounce delay as other pushbuttons. This decision complicated the design of the communications software, since it adversely affected how soon after enabling communications the buffer could be flushed.
- The tool shall maintain a database of requirements. The database shall contain required behaviour components including initialisation, rationale and references, descriptions of stimuli (monitored variables), responses (controlled variables), constants, and types.

  It is often essential that we find all requirements that involve a particular function, constant or some other identifier. A static database that reflects the current relationships would be indispensable. Searches in a flat text environment are possible, but the time taken dealing with false matches is wasteful. Such a database would clearly enhance our capability to trace requirements.
- The tool shall maintain lists of likely and unlikely changes that may be made in the future.

  These lists are essential inputs to the information hiding design principle.
- The tool shall maintain a database of required timing behaviour for each controlled variable/monitored variable pair.

  Timing behaviour is the most complex issue we deal with at the requirements level. It also is a crucial element in the design of the scheduler in the software design. Time required by the hardware, especially I/O hardware, is vital. The software designer and verifiers need to have easy access to this information so that they can calculate the timing tolerances that apply to the software alone.
- The tool shall provide different views of the requirements, i.e. the requirements organised/partitioned in different ways, some textual and some graphical.

  No single view provides the best view of the requirements for all situations. Sometimes tabular expressions are required for a detailed view, while at other times, an overview via data-flow diagrams may be better.
- The tool shall be able to generate lists of differences in behaviour between two selected revisions of the requirements. It would be useful to be able to select sets of behaviour to be included. For example, select initialisation only, or select timing requirements only, or select all behaviour. The lists should include related rationale if requested.

  A document "diff" is an indispensable tool. However, current comparison tools are severely lacking when the document contains function tables and graphics.

6.5 Software design tool

- The tool shall maintain lists of likely and unlikely changes that may be made in the future. These lists should be of items not already included in the requirements lists of likely and unlikely changes. The tool shall be able to combine the requirements and design lists so that a complete set of likely and unlikely changes is documented.

  These lists are essential to the information hiding design principle. They drive the decomposition of requirements into modules as described in Sect. 2.2.
- The tool shall maintain a database of design behaviour. The database shall contain the behaviour of each access program for each module, organised by interface and implementation.

  Similar to the requirements database, the design database is an essential component of the design tool. It is far more useful as a persistent database, accessible by other tools, rather than as a non-persistent database as in the current tool suite.
- The database maintained by the tool should include mappings between requirements identifiers and software design identifiers.

  The software designer is always aware of these relationships while creating the design, and it has to be documented anyway. The information is immensely helpful in later life-cycle phases, especially during software design verification.
- The database of design behaviour shall include provision for the storage of design notes linked to any relevant aspect of the design.

  Design notes are the equivalent of rationale at the requirements level. They record all relevant design decisions, from top level decisions such as decomposition into modules, to lower level decisions such as the choice of data structures, Maintenance of the design will be hampered if relevant design notes are not included in the documentation. A tool that eases the burden of compiling and storing these notes will make it more likely that designers actually record the notes.
- The tool shall enable project managers to assign the internal detailed design of specific modules to specific designers. The tool shall then make available to each designer only what that designer is supposed to be able to access.

  This is another essential aspect of information hiding. For example, a designer implementing the internal design of module A should be allowed to see all details related to module A, but only the interface specifications for each of the other modules.
- The tool shall provide different views of the design, i.e. the design organised/partitioned in different ways, some textual and some graphical.

  This is equivalent to a similar item for the Requirements Tool. Viewing the design by data-flow, uses hierarchy, or scheduling requirements satisfies different objectives. No single view is universally useful.

6.6 Software design verification tool

- The tool shall take as input the main documentation used by all developers, reviewers, testers, and verifiers.

  Since it is less readable and typically used by fewer people, the formal documentation can easily become out of date. By taking the main documentation as input, the project receives the benefits of the associated formal methods without an increases burden in documentation, though this does require the documentation to adhere to a more rigid document format in order to facilitate parsing by the tools.
- The tool shall interface to general verification tools that have built in proof strategies or decisions that automatically discharge a significant portion of these conditions.

  Controls engineers would never dream of rolling their own mathematical design and simulation tools when programs such as Matlab with its controls toolbox are available. Similarly good theorem provers, modelcheckers and other tools are difficult to build and debug. Significant development effort has been expended on these tools and we should make use of it. The domain specific toolboxes of Matlab provide us with a successful model for specialization of a powerful, general tool to improve applicability to a particular domain.
- When the tool suite fails to automatically prove a verification obligation, it shall, when possible, generate a counter example that can be simulated/executed by the verifier. If it fails in the counter example generation it should allow the verifier to attempt the proof interactively.

  As much as possible of the verification should be automated but having access to an interactive mode for verification allows human guidance based upon knowledge of the problem to guide the verification if *necessary.*
- The tool shall have batch processing and audit trail generation facilities. After any change to any of the documents, it should be possible to automatically rerun all of the previous verification steps (consistency checks, block comparison proofs, etc.) and see which, if any, have broken.

  Generation of complete detailed proof output as an audit trail for the verification was required by the regulator. Before batch processing facilities were added to PVS, rerunning the verification proofs and producing the proper output was time consuming and tedious manual work requiring a couple of days effort. Emacs LISP scripts to automate the process were eventually written that allowed the re-verification and output generation process to be automatically run over night after each revision of the documents, automatically highlighting the failed proofs. Over night, the design verification tool went from an additional burden to a system that improved productivity.

6.7 Coding tools

Coding from formal, detailed designs, is straightforward in most programming languages. Generic programming tools are likely to be more than adequate. (See Sect. 7 for discussion on a Coding Tool we would like to see in the future.)

6.8 Code review tools

As with the coding tools, many generic code review tools could prove useful. The deciding factor is support for the specific programming language(s) used on the project.

6.9 Testing tools

Testing usually requires tools that are quite project specific and as such are really beyond the scope of this paper. Testers do, indeed need access to the requirements, software design and code, and will make heavy use of the relevant tools.

For hard real-time systems, testing timing is always a major concern. Without precise *logic analysers* this task would usually prove to be intractable.

6.10 Code verification tool

- The tool shall generate the *Code Verification Report*, organised by modules, directly from the code.

  A restatement of the purpose of the tool.
- For each code module generated by the tool, the layout of the document should be as close as feasible to the structure of the relevant sections of the software design itself.

  This facilitates manual comparison with the software design. In the future, much of the comparison may be handled automatically – see Sect. 7.
- The tool shall automatically generate design extracts in appropriate sections of the *Code Verification Report* as directed by the analyst.

  Again, for manual comparison it is necessary to show the corresponding section of the software design. It is time consuming for the analyst to copy and paste from the software design into the verification document.
- The tool shall be capable of merging the analysis and comparisons so that if the report has to be regenerated existing analyses and comparisons will be automatically inserted in the appropriate sections.

  This is special case of the general requirement (Sect. 6.1) to be able to automatically include manually created sections in future versions of the document.

6.11 Table tool

- The tool shall include the capability to check tabular expressions for completeness and disjointness, in the context within which the tabular expression is placed. (The tool will need information about the identifiers in the table cells. This information will be extracted from the relevant databases.)

  As already indicated (Sect. 2.1), one of the main benefits of representing functions by tabular expressions is that the tables present the functions so that the input domain is demonstrably complete, and the predicates are disjoint. This enforces specifications in which every input condition has an assigned output value, and the specifications are unambiguous. In some complex tables, these conditions are not obvious and automated support is particularly time-effective.
- The tool shall include the capability to transform a tabular expression of a particular "kind" (*vertical condition table*, for example) into an equivalent tabular expression of a different kind (*structured decision table*, for example).

  We often have to compare behaviour specified in one tabular format with the behaviour specified in a different format. It is essential to be able to make these comparisons. At the very least, the tool must be able to convert all tabular expressions to one tabular format. This is the option chosen by OPG.
- The tool shall be capable of comparing two tabular expressions to determine if they describe equivalent behaviour.

  This is an obvious requirement. It may be implemented within the Table Tool itself, or the tool may use theorem provers such as PVS.
- The tool shall be capable of functionally composing tabular expressions.

  One of the most common tasks in dealing with tabular expressions, is to compose two or more tables. For example, it is quite common for the software design to implement the composition of tables in a single program. Functional composition of tables presents significant technical problems, since it is necessary to be able to simplify resulting expressions if the composition is to be useful. Some progress in this regard has been made recently by Kahl [14].
- The tool shall be capable of interpreting a tabular expression that has been manually created (as long as the manually created table adheres to agreed on specifications).

  This is again a special case of the general requirement (Sect. 6.1) to be able to automatically include manually created sections in future versions of the document.

6.12 Other specification tools – pseudo-code tool

The only other specification tool we require (immediately), is one that will handle the pseudo-code used to specify algorithms when the sequence of statements is important (since tabular expressions are independent of the sequence of statements).

7 Future tools

A tool that would be of obvious benefit, as discussed in Sect. 6.10, is a tool that could extract tabular expressions from source code. If we could construct such a tool, we could go a long way towards completely automating the code verification. Although we have over thirteen years experience in performing this task manually, we still do not know how to perform this task automatically. As far as we know, no-one has succeeded in doing this – yet. Such a tool would be extremely useful in reverse engineering projects as well. We are actively investigating methods for automating this task, and are examining a few promising alternatives. The case in which the code is transformed from tabular expressions through the application of proscriptive coding guidelines is clearly easier to automate than the general case.

As described in Sect. 6.7, a tool that could transform tabular expressions into target code, would also be beneficial. Our experience is that the module internal designs in the software design are detailed enough to make this practical. The coding guidelines in use are already extremely prescriptive, and in this domain specific context, it should be possible to construct efficient code from the tabular expressions.

Another candidate for a tool is mentioned in Sect. 6.10. This tool would perform the detailed comparisons of the tabular expressions extracted from the code (together with the extracted variable classifications), against the corresponding information in the software design. It is likely that the tool would have to link to a theorem prover such as PVS.

Automatic generation of test cases is of obvious benefit. Tabular expressions would seem to lend themselves to this activity and there has been some preliminary work in this regard. However it is of practical importance that the number of generated test cases is manageable, which prohibits the application of some of the more obvious "brute force" approaches.

8 Conclusion

We believe that most formal methods will not be viable in practical software development, without extensive and comprehensive tool support. It is vital that the suite of tools should be integrated and work co-operatively over the complete software development life-cycle.

Developers of the methodologies need to produce tool support of a quality commensurate with the anticipated usage of the methodology. One way of achieving this in the long term, is to develop them as open source products, in cooperation with the growing community of software developers whose applications are (or will be) classified as safety-critical.

One lesson we have learned, is that developing tools to cope with mundane, but necessary and time-consuming tasks, can have an immense impact on schedule. The code verification tool described in Sect. 3.2, e.g., reduced the document preparation time from approximately three months to less than one month. The 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.

Another vital lesson we learned is that the tools must be flexible. The tools should never prevent the use of manual construction of sections of project documents. If they do prevent these manual efforts, it is possible that critical deadlines/milestones may be missed simply because of a tooling problem.

Acknowledgements We would like to thank the reviewers of this paper. Their thoughtful and constructive criticism resulted in many substantial improvements.

The work presented in this paper is based on the efforts of many current and former employees of OPG 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. Finally we would like to acknowledge David Parnas. This work represents the successful application of many of his ideas regarding software engineering.

References

1. Abraham, R.: Evaluating generalized tabular expressions in software documentation. Technical report CRL No. 346, McMaster University, Hamilton, ON, Canada (1997)
2. Anderson, P., Reps, T., Teitelbaum, T.: Design and implementation of a fine-grained software inspection tool. IEEE Trans. Softw. Eng. **29**(8), 721–733 (2003)
3. Archer, M., Heitmeyer, C., Riccobene, E.: Proving invariants of i/o automata with tame. Automated Softw. Eng. **9**(3), 201–232 (2002)
4. Autexier, S., Hutter, D., Mossakowski, T., Schairer, A.: The development graph manager MAYA. In: Kirchner, C.R.H. (ed.) Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology, AMAST 2002, LNCS, vol. 2422, pp. 495–501. Saint-Gilles-les- Bains, Reunion Island, France, (2002). Springer, Berlin Heidelberg New York
5. Concurrent versions system: the open standard for version control, web site at http://www.cvshome.org
6. Dutertre, B., Stavridou, V.: Formal requirements analysis of an avionics control system. IEEE Trans. Softw. Eng. **23**(5), 267–278 (1997)
7. Heitmeyer, C.: Software cost reduction. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering, 2nd edn., Wiley, New York (2002)
8. Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR*: A toolset for specifying and analyzing software requirements. In: Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98), Vancouver, BC, Canada, (1998) Lecture Notes in Computer Science, vol. 1427, pp. 526–531. Springer, Berlin Heidelberg New York (1998)
9. Heitmeyer, C., Bull, A., Gasarch, C., Labaw, B.: SCR*: A toolset for specifying and analyzing requirements. In: Proceedings of the 10th Annual Conference on Computer Assurance, Compass '95, pp. 109–122, Gaithersburg, Maryland. National Institute of Standards and Technology (1995)
10. Heitmeyer, C., Kirby, J., Jr. Labaw, B., Archer, M., Bharadwaj, R.: Using abstraction and model checking to detect safety violations in requirements specifications. IEEE Trans. Softw. Eng. **24**(11), 927–948 (1998)

11. Heninger, K.L.: Specifying software requirements for complex systems: New techniques and their applications. IEEE Trans. Softw. Eng. **6**(1), 2–13 (1980)
12. Hutter, D.: Management of change in structured verification. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE-2000), pp. 23–34. IEEE Computer Society (2000)
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, chapter 12, pp. 184–196. Springer Wien New York (1997)
14. Kahl, W.: Compositional syntax and semantics of tables. Technical report 15, Software Quality Research Lab, McMaster University, Hamilton, ON, Canada (2003)
15. Khedri, R., Wu, R., San, B.: SCENATOR: a prototype tool for requirements inconsistency detection. In: Wang, F., Lee, I. (eds.) Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis, pp. 75–86, Taiwan, Republic of China. National Taiwan University, National Taiwan University (2003)
16. Knight, J.C., Hanks, K.S., Travis, S.R.: Tool support for production use of formal techniques. In: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001), Hong Kong, China. IEEE Computer Society (2001)
17. Kohlhase, M., Anghelache, R.: Towards collaborative content management and version control for structured mathematical knowledge. In: Asperti, A., Buchberger, B., Davenport, J.H. (eds.) Proceedings of the 2nd International Conference on Mathematical Knowledge Management, MKM 2003, LNCS, vol. 2594, pp. 147–161, Bertinoro, Italy. Springer, Berlin Heidelberg New York (2003)
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 Formal Methods in System Design, (2004). Draft available at http://www.cas.mcmaster.ca/ lawford/papers/
19. Lawford, M., Hu, X.: Right on time: Pre-verified software components for constructuion of real-time systems. Technical report 8, Software Quality Research Lab, McMaster University, Hamilton, ON, Canada (2002)
20. 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 of the 8th International Conference on Algebraic Methodology and Software Technology, AMAST 2000, Iowa City, Iowa, USA, (2000). Lecture Notes in Computer Science, vol. 1816, pp. 73–88. Springer, Berlin Heidelberg New York (2000)
21. Owre, S., Rushby, J., Shankar, N.: Integration in PVS: Tables, types, and model checking. In: Brinksma, E. (ed.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97), Lecture Notes in Computer Science, vol. 1217, pp. 366–383, Enschede, The Netherlands. Springer, Berlin Heidelberg New York (1997)
22. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Trans. Softw. Eng. **21**(2), 107–125 (1995)
23. Parnas, D.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
24. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Prog. **25**(1), 41–61 (1995)
25. Parnas, D.L.: Using mathematical models in the inspection of critical software. In: Hinchey, M.G., Bowen, J.P. (eds.) Applications of Formal Methods, International Series in Computer Science, chapter 2, pp. 17–31. Prentice Hall, Englewood Cliffs, NJ (1995)
26. Parnas, D.L., Clements, P.: A rational design process: How and why to fake it. IEEE Trans. Softw. Eng. **12**(2), 251–257 (1986)
27. Paulson, L.: Better software with open source? IEEE Comput. Mag., pp. 20–21 (2000)
28. Rumbaugh, J., Jacobson, I., Booch, G.: The unified modeling language reference manual. Addison-Wesley, Reading, MA (1998)
29. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in PVS. IEEE Trans. Softw. Eng. **24**(9), 709–720 (1998)
30. Scheffczyk, J., Borghoff, U.M., Rödig, P., Schmitz, L.: Consistent document engineering: Formalizing type-safe consistency rules for heterogeneous repositories. In: Proceedings of the 2003 ACM Symposium on Document Engineering, pp. 140–149. ACM, New York (2003)
31. Viola, M.: Ontario Hydro's experience with new methods for engineering safety critical software. In: Proceedings of the 14th International Conference on Computer Safety, Reliability and Security, SAFECOMP'95, pp. 283–298, Belgirate, Italy. Springer, Berlin Heidelberg New York (1995)
32. Wassyng, A., Janicki, R.: Using tabular expressions. In: Proceedings of International Conference on Software and Systems Engineering and their Applications, vol. 4, pp. 1–17, Paris (2003)
33. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandriioli, D. (eds.) Proceedings of the International Symposium of Formal Methods Europe Proceedings, FME 2003, Lecture Notes in Computer Science, vol. 2805, pp. 133–153. Springer, Berlin Heidelberg New York (2003)