Integrated Software Methodologies - An Engineering Approach

Alan Wassyng* & Mark Lawford*

Software Quality Research Laboratory, Department of Computing and Software,
McMaster University, Hamilton, Canada,
wassyng@mcmaster.ca, lawford@mcmaster.ca

Abstract. We make the case for greater integration of the individual methods and documentation used in the software development life-cycle. We illustrate practical benefits from this approach through the use of examples drawn from a methodology that has been used to develop successful safety-critical applications. Integration limits the number of different specifications that have to be developed and also typically results in improved traceability from requirements through to code. We also show that integration may reduce the burden imposed by formal verification. Integrated software toolsets are a natural consequence. This integration is exactly what engineers do in other disciplines, and software engineers should take note of this.

SQRL Report No. 51

* Partially supported by NSERC.

1 Introduction

Software Engineering is gradually being recognized as an engineering discipline. Like other engineering disciplines, there is a mutual dependence between the engineering discipline and its basic science foundation. Engineering uses research from the basic sciences to develop engineering approaches, methods, heuristics and standards. It also feeds back suggestions for further research, both in depth and in shifts of focus. We are at one of those stages where significant progress can be achieved in Software Engineering by changing our focus. Software Engineers, like other engineers, are naturally concerned with *standards* and *process* - about creating suitable standards and processes and managing those processes. Other engineering disciplines discovered many years ago that there are significant advantages to developing processes that depend on each other for the final solution of an engineering problem. To date, Software Engineers have been slow to realize this. For many years now, we, the general software and computing community, have conducted research in aspects of requirements analysis and documentation, software design, programming, testing, verification and validation, specification, inspection, languages, compilers, and operating systems. Relevant aspects of this research and experience often find their way into the definition of software standards and development processes. We have developed tremendous knowledge and skills in all of these areas. In the early 1990s there seemed to be some attempt to develop "integrated methods", but that thrust seems to have dissipated - except for attempts at integration through *refinement* and *automated code generation*. Recently there has been progress on deriving architectural design from the requirements [8], and there are a small number of research efforts that target some form of integration, the most notable of these being the work by van Lamsweerde and others on KAOS [4]. However, in general, we seem to have neglected this aspect of research, namely how the different life-cycle phases do and should interact, and how to directly use documentation from one phase in another, related, phase. Hopefully, this paper will encourage Computer Scientists and Software Engineers to concentrate on research and development of methods and software tools that are much more integrated over the software development life-cycle than is currently the norm.

The paper is organized as follows. The first section deals with the development of software methods, starting with typical approaches used up until now (Section 2), and then the proposed approach (Section 3) together with an example drawn from an industrial safety-critical project (Section 4). Finally, the conclusions are presented in Section 5.

2 A Typical Approach

Understandably, researchers who are involved in the development of software methods, typically concentrate their efforts on a specific phase of the software life-cycle. Their emphasis is on the development of the *best* method for that phase, dependent on the evaluation criteria they deem important. For example, there are numerous papers on specification of software interfaces. If we

assume that the software design is decomposed into *modules* (as proposed by Parnas [12]), then a *module interface* typically comprises the exported constants and types and the *access programs* (*methods* in object-oriented designs) of that module. Common wisdom dictates that the interface specification describes the externally visible required behaviour of an access program without divulging how this will be achieved. This has led to a number of proposed specification methods for module interfaces. An example of this approach is the *trace assertion* method proposed by Bartussek and Parnas [2]. However, the thrust of this discussion applies equally well to other methods.

The trace assertion method uses module interface call sequences, *traces*, as the components of the specification. In Parnas's design documentation ([13, 14]) these trace assertions are then used in the *Module Interface Specification*. This specification approach works, but it has a flaw. The method has no dependence on, and does not even overtly acknowledge, the possible existence of a mathematically precise requirements specification. If we assume the existence of a formal requirements document, we can use the requirements specifications of individual functions in the design interface specification. Similar situations exist for all phases of the software development life-cycle. For example, if we know that the methodology includes a detailed design document, then that document can be used to "comment" the code by including cross-references to the design document in the code (see Fig. 11).

3 A Better Approach

Let us suppose that we are developing a software design method, and are considering how to document the module interface specifications. However, we know that there is (or will be) a method to develop a mathematically precise requirements document. What is to prevent us from using this formal specification of requirements to document the module interfaces? After all, the abstraction level of the requirements specification should be appropriate for the module interface specification as well. It would certainly be an advantage to be able to use the requirements specification to document the module interfaces, since we would not have to develop an "extra" specification. We also would not have to verify that the behaviour described by the interface specification is the same as the behaviour described in the requirements.

There are two main reasons why it may be difficult to use the requirements specification to document the module interfaces.

The first reason is that the requirements specification deals with variables in the physical application domain, while the software design deals with variables inside the computer. This situation was described by Parnas and Madey [14] using their now famous "4 variable model" (see the top section of Fig. 1).

This model relates the *monitored variables* M, to the *controlled variables* C, through a relation REQ (requirements). M and C represent vectors of time-functions in the physical domain. The monitored variables are transformed by hardware devices into *input variables* I. These variables are stored in the com-

puter (or, at least, at the boundary of the computer), and are inputs to the software application. The software application is represented by the relation SOF, and so SOF operates on I to produce O, the *output variables*. These variables are also stored in the computer, and are transformed by hardware into the controlled variables. The input and output hardware devices can be represented by the relations IN and OUT. It is clear from Fig. 1 that, if the 4 variable model is applied, the requirements document will specify REQ in terms of M and C, while the software design document will specify SOF in terms of I and O.

The second reason that makes it difficult to use the requirements specification to document the module interfaces, is that the requirements and software design typically will be decomposed in totally different ways. The requirements decomposition may be chosen to make the requirements document more readable and understandable. The software design may be decomposed to enhance the development and maintainability of the software.

It turns out that with some effort both of these problems can be overcome. There are probably a number of ways to achieve this. This paper describes the approach we used.

Mismatch between requirements and design variable: Although the 4 variable model is a useful representation of the situation, in 1993, while developing the first version of [10], we realized that we should deal with the software design in a slightly modified way as shown in the bottom expansion in Fig. 1. Others have published similar modifications of the 4 variable model, [16, 3].

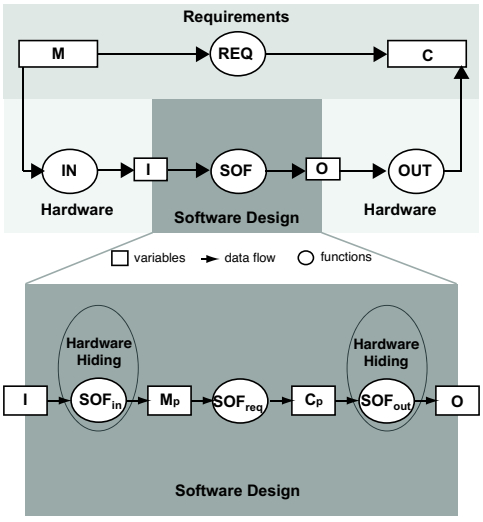

Fig. 1. Modified 4 Variable Model

With the addition of very simple *hardware hiding modules*, we can arrange that SOF_{req} is defined in terms of close approximations of the requirements variables M and C. Thus, the behavioural modules in the software design, contained within SOF_{req}, are defined in terms of M_p and C_p. This means that REQ can be used to define the behaviour required in SOF_{req}. See the discussion on "References" in Section 4.2.

Different data-flow decompositions in requirements and design: The second problem is not as simple to deal with. The difference in data flow topology between requirements and design is obvious. It is a problem for us because it is likely that the behaviour of a single access program will be determined by portions of one requirements function composed with portions of another requirements function. If we need to describe the behaviour of that access program, the requirements functions will not have the necessary granularity.

Imagine if the requirements specification data-flow topology exactly matches that in the software design. One way in which we can create such a requirements specification is by piece-wise "replacing" some composition of existing 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 *supplementary functions*. Since most functions in the requirements document are described by tabular expressions, we typically refer to these supplementary functions as *supplementary function tables* (SFTs).

We develop the SFTs during the software design process. They are then available to document the module interfaces and also to aid in the mathematical verification of the software design. Examples of SFTs will be shown in Section 4. A simple data flow example to illustrate SFTs was included in [17].

The SFTs serve two important purposes in our integrated methodology. i) Together with the modification to the 4 variable model, they enable us to use the requirements specification to document the module interfaces. ii) The major step in the design verification can be performed piece-wise. See Section 4.3. We call the version of the requirements specification that contains SFTs, the *pseudo-requirements specification*.

4 A Detailed Example

Most of the extracts in the example presented below, have been taken from a successful industrial safety-critical application, namely a shutdown system for a nuclear power generating station that was the subject of [17].

4.1 Requirements

The requirements model is a time-discrete finite state machine with an arbitrarily small time step. The names of the *monitored variables* are prefixed by m_. The names of the *controlled variables* are prefixed by c_. Similarly, constant names

are prefixed by k_, internal functions caused by functional decomposition of the requirements have names prefixed by f_, type names are prefixed by y_ and enumerated token names are prefixed by e_. The value of a function or variable at the previous clock step is denoted by the name subscripted by -1, such as f_name$_{-1}$ or c_name$_{-1}$. The current time is denoted by t_{now}.

The vast majority of the requirements are specified using tabular expressions [6, 7]. Comments in the tables are italicized, and are enclosed within braces. As an example, the definitions of the "Neutron OverPower Parameter Trip and Sensor Trips" are shown in Fig. 2. Roughly speaking, a sensor trip occurs when a function that depends on a sensor value goes out of its safe range. A parameter trip depends on a function of related sensor trips.

4.2 Software Design

The software design re-organizes the way in which the behaviour in the requirements is partitioned. This is done to achieve specific goals, the major 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.

Information hiding principles [12] form the basis of the design philosophy, and the module interface specification for each module lists exported constants and types. It also lists all access programs, specifies their invocation syntax, and specifies the semantic behaviour of each access program by referencing the appropriate pseudo-requirements functions.

Fig. 3 shows the module interface specification for module NPParTrip, the module responsible for evaluating and supplying the value of the controlled variable c_NOPparmtrip. The semantic behaviour of access program EPTNP is thus documented by listing c_NOPparmtrip in the *references* section.

The internal declarations for module NPParTrip are shown in Fig. 4. The state data that stores the current value of the NOP Parameter Trip, PTSNP is the only state variable in this module. The constant KNUMNP (= 18) is the number of NOP sensors. The mapping of $TRUE to e_Trip and $FALSE to e_NotTrip is documented in a different section of the software design document.

Fig. 5 shows the detailed design for access program EPTNP. The design detail shows the call (shown as an *external value*) to another access program responsible for maintaining the digital output status (SDONP from module DigitalOutput), and the explicit call to a "Get" program (GSTNP in module NPSnrTrip) that provides the current values of the 18 NOP sensor trips. In this simple example there is no significant difference between the detailed design and its interface specification.

To demonstrate the use of supplementary functions and supplementary function tables consider the following requirements and module interface specification extracts.

The requirements extract in Fig. 6 shows the formal definition of the *natural language expression* 'Watchdog test active', which is used in c_watchdog (not shown in this example), the function that describes the status of the watchdog.

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*}	No Change
'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. 2. Requirements Specification of the NOP Parameter Trip and Sensor Trips

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

Initializes the NPParTrip module internal states.
References: *Initial Value [c_NOPparmtrip]*.

Fig. 3. Example Module Interface Specification

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. 4. Example Module Internal Declaration

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

Fig. 5. Example Access Program Detailed Design

The initial version of the module interface specification of the Watchdog Module is shown in Fig. 7. Note the references for the access programs EWDOG and SWDOG. Different portions of 'Watchdog test active' are used in each of those access programs, so 'Watchdog test active' is listed for both of those programs. This is because the software designer decided that it would be better to split triggering the watchdog test timer from the rest of the watchdog logic. The asterisk is used to indicate that portions of the function rather than the complete function define the semantics of the access program. However, this is clearly not adequate since we do not know exactly what portions of the function apply to each program.

The software designer then makes supplementary functions to describe precisely how the function 'Watchdog test active' is split into three functions, the composition of which describes behaviour equivalent to that of the original function. Fig. 8 shows the supplementary function tables that "replace" 'Watchdog test active'. Note that only two of the three SFTs are used in the Watchdog

Condition			*Result* **Watchdog test active**	*Result* **$t_{wdstart}$**
NOT ['Calibrate enable requested']			False	$t_{now}-$ k_WDtestdelay -1
'Calibrate enable requested'	M_RxFnType = e_WD_test		True	t_{now}
	NOT[M_RxFnType = e_WD_test]	t_{now} - $(t_{wdstart})_{-1} \leq$ k_WDtestdelay	True	No Change
		t_{now} - $(t_{wdstart})_{-1} >$ k_WDtestdelay	False	No Change

Fig. 6. Requirements Description of 'Watchdog test active'

4.44 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 [c_watchdog, $t_{wdstart}$].*

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 effect on the module.
References: *'Watchdog test active'**.

Fig. 7. Initial Module Interface Specification of Watchdog Module

module. The other SFT, Received request, is implemented in the communication module in which the call to SWDOG takes place. (These SFTs were produced for this paper. The project used an equivalent but different description of the SFTs.)

Finally, we can see in Fig. 9 that the references for the access programs in the Watchdog module now define the semantic behaviour of the programs precisely, since the ambiguous requirements functions denoted with asterisks have been replaced by relevant supplementary functions. All supplementary function tables are defined in a section of the software design document.

4.3 Software Design Verification

Motivation: In 1989 some of us were involved in the verification of the first version of the Darlington Shutdown System [1]. This involved the verification of code against a detailed specification at an abstraction level somewhere between requirements and software design. The verification was performed "after the fact", in that the code was developed without any formal verification in mind. It proved extremely difficult and time-consuming to complete. Our conclusion was that the verification step was too large. This was a valuable lesson in the development of a safety-critical software development methodology. This experience led us to develop a methodology in which we would perform two formal verifications instead of one. The first verification is to prove the software design compliant with the requirements specification, and the second is to prove the code compliant with the software design. The two smaller steps proved to be far easier than the single large step. This decision reflects our end-to-end view of the development process, and clearly influenced the overall methodology.

Design Verification: One crucial problem that we had to overcome was that the obvious proof obligation to prove that SOF is compliant with REQ (see Fig. 1) is extremely onerous. This proof obligation is:

$$\begin{aligned} OUT(SOF(IN(M))) &= REQ \\ I &= IN(M) \\ C &= OUT(O) \end{aligned} \tag{1}$$

It is easy to see that even for small problems this becomes very complex and prohibitively time-consuming to perform.

In Fig. 1, we can represent the function chain resulting in M_p by an *abstraction function* Abst_m. Similarly, we can represent the C_p chain by Abst_c. Then the proof obligation becomes:

$$\begin{aligned} Abst_c(REQ) &= SOF_{req}(Abst_m(M)) \\ M_p &= Abst_m(M) \\ C_p &= Abst_c(C) \end{aligned} \tag{2}$$

2.6.14 SFT[Received request]

Condition		*Result* **'Received request'**
NOT ['Calibrate enable requested']		False
'Calibrate enable requested'	M_RxFnType = e_WD_test	True
	NOT [M_RxFnType = e_WD_test]	False

2.6.21 SFT[$\mathbf{t}_{wdstart}$]

Condition		*Result* $\mathbf{t}_{wdstart}$
'Received request'		t_{now}
NOT ['Received request']	NOT ['Calibrate enable requested']	x
	'Calibrate Enable Requested'	No Change

where x = t_{now} - k_WDtestdelay - 1

2.6.25 SFT[Watchdog test active]

Condition		*Result* **'Watchdog test active'**	$\mathbf{t}_{wdstart}$
NOT ['Calibrate enable requested']		False	x
'Calibrate enable requested'	t_{now} - $(t_{wdstart})_{-1} \leq$ k_WDtestdelay	True	No Change
	t_{now} - $(t_{wdstart})_{-1} >$ k_WDtestdelay	False	No Change

where x = t_{now} - k_WDtestdelay - 1

Fig. 8. Supplementary Function Tables for 'Watchdog test active'

4.44 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, SFT[Watchdog test active].*

IWDOG

Initializes all the Watchdog module internal states and sets the initial watchdog output.
References: *Initial Value [c_watchdog,* $t_{wdstart}$*].*

SWDOG

NCPARM: t_boolean - in
Signals to Watchdog module that a valid watchdog test request is received if NCPARM = $TRUE.
References: *SFT[*$t_{wdstart}$*].*

Fig. 9. Revised Module Interface Specification of the Watchdog Module

This may not appear to be more tractable than (2), but if we verify SOF_{req} against the pseudo-requirements, REQ_p, rather than against REQ, then we can show that

$$Abst_c(REQ_p) = SOF_{req}(Abst_m(M)) \tag{3}$$

can be performed piece-wise, and after that we can prove $REQ_p = REQ$. The success of this approach depends on being able to choose the individual blocks to verify and to show that verification of those blocks implies verification of the whole. If we assume that the data-flow topologies of the requirements and design are equivalent, we can show that we can identify blocks in which the inputs and outputs of the block in the requirements, are "associated with" the inputs and outputs of the block in the design. The verification task then becomes to prove compliance of the design block with the requirements block. We can also show that the total proof obligation will be satisfied if each block's proof obligation is satisfied. A small example is shown in Fig. 10.

The example shows that we "associate" outputs of functions between requirements and design using abstraction functions A_i. In this example we can identify the four individual blocks as shown, and if we prove compliance of each of the blocks individually, then simple substitution shows that (3) is satisfied. Fig. 10 shows the block proof obligations for that specific example. The corresponding instantiation of (3) is shown in the boxed equations in the figure.

So, the modified 4 variable model and SFTs are vital links between the requirements, software design, and design verification methods, helping us to

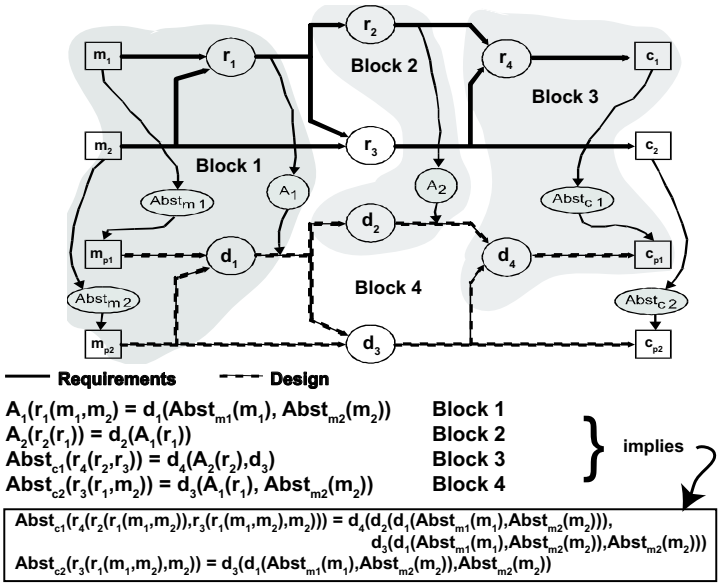

Fig. 10. Piece-Wise Proof Example

integrate those phases of the software life-cycle. What is important here is that we probably would not have invented SFTs if we had not been intent on using the requirements functions to document the module interface specifications.

4.4 Code

The programming languages available on the hardware platform used for the example shutdown system were a dialect of FORTRAN 66 and Assembler. Fig. 11 shows an extract from the access program EPTNP written in FORTRAN. Note that some comments have been removed from the extract so that the figure can fit on the page. Those comments are not relevant to the discussion presented in this paper. As can be seen, the code is a very direct, mechanical translation from the tabular description in the software design shown in Fig. 5. The comments in the code do not try to provide the reader with the coder's intent, or an overview of the semantic behaviour implemented in the code. Rather, the comments in the code are typically references to applicable sections in the software design document. For example, note the comments to "VCT EPTNP" (Vertical Condition Table EPTNP). This comment serves to block off the code that implements the behaviour in the detailed design for EPTNP.

4.5 Code Verification

The code verification became one of the easier steps in our development life-cycle. First of all, as a consequence of the mechanical production of code from function

```
      SUBROUTINE EPTNP
      IMPLICIT COMPLEX (A-Z, $)
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 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. 11. Code Example of Access Program EPTNP

tables in the design, the code is extremely well-structured. It is relatively easy to reverse-engineer a function table representation of the code. In the example in Fig. 11, the comments in the code inform the code verifier that the software design was described by a function table (VCT), not by an algorithm (pseudo-code). This helps the verifier who then knows, without reference to the software design, that a function table must be produced from the code so that it can be compared with the function table in the design. The mechanical nature of these operations certainly raises the chance of developing software tools to automate many aspects of the code verification. These tools have not yet been completed.

5 Conclusion

This paper has a very simple message, namely that we need to build methodologies comprised of methods that are designed to work together. This is an approach that is common in most engineering disciplines. Over the past few years, software engineers have targeted *reuse* as a means to achieve reliability and a reduction in the cost of critical software applications. Integrated methods will allow us to benefit from reuse of specifications, documentation and tools in a much more comprehensive way. The methods that we presented in this paper are introductory examples of what can be achieved. They were developed in an industrial setting and this had a strong influence on both the methods and the support tools. The driving force behind developing integrated methods was, in fact, that this is a normal (good) engineering approach in industry. Industrial pressure also dictated that a good process and a good tool were "good enough," and there was little time to explore the impact of what we had achieved in a broader sense. One conclusion we did draw is that there is little hope of achieving significant improvements in software quality without adequate tool support, and our experience has been that these comprehensive and integrated methods also drive ideas for new kinds of tools.

Acknowledgements

The work presented in this paper is based on the efforts of many current and former employees and consultants of Ontario Power Generation Inc., and AECL, including: Glenn Archinoff, Dominic Chan, Rick Hohendorf, Paul Joannou, Peter Froebel, David Lau, Elder Matias, Jeff McDougall, Greg Moum, Brian Quigley, Mike Viola, and Alanna Wong. Finally we would like to acknowledge and thank David Parnas. This work reflects the successful application of many of his pioneering and fundamental ideas regarding software engineering.

References

1. 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)
2. Bartussek, W., Parnas, D.L.: Using Assertions About Traces to Write Abstract Specifications for Software Modules. UNC Report No. TR77-012 (1977)
3. Bharadwaj, R., Heitmeyer, C.L.: Developing High Assurance Avionics Systems with the SCR Requirements Method. In: Proceedings of the 19th Digital Avionics Systems Conference, Philadelphia, P.A., (2000)
4. Darimont, R., Delor, E., Massonet, P., van Lamsweerde, A.: GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. In: *Proc. of (19th) International Conference on Software Engineering*, 17-23 May (1997), 612–613
5. Heitmeyer, C.: Software Cost Reduction. In: Marciniak, J.J. (ed), Encyclopedia of Software Engineering, John Wiley & Sons, Inc., 2nd edition, (2002)
6. 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, **12** (1997) 184–196
7. Janicki, R., Wassyng, A.: Tabular Expressions and Their Relational Semantics. Fundamenta Informaticae, **68** (2005) 1–28
8. Khedri, R., Bourguiba, I.: Formal Derivation of Functional Architecture Design. In: Cuellar, J.R., Liu, Z. (eds) Proceedings of Second International Conference on Software Engineering and Formal Methods, Beijing (2004) 356–365
9. 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: Proc. of AMAST 2000, Iowa City, Iowa, USA. Lecture Notes in Computer Science, **1816** (2000) 73–88
10. Moum, G.: Systematic Design Verification Procedure, Revision 02, NK38-MAN-68000-001. Ontario Power Generation, Darlington NGD Shutdown System Trip Computer Software. (2000) Proprietary.
11. Owre, S., Rushby, J., Shankar, N.: Integration in PVS: Tables, Types, and Model Checking. In: Brinksma, H. (ed), Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, **1217** (1997) 366–383
12. Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, **15**(12), (1972) 1053–1058
13. Parnas, D.L., Clements, P.: A Rational Design Process: How and Why to Fake it. IEEE Trans. Software Engineering, **12**(2) (1986) 251–257
14. Parnas, D.L., Madey, J.: Functional Documents for Computer Systems. Science of Computer Programming, **25** (1995) 41–61
15. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, (1998)
16. Thompson, J.M., Heimdahl, M.P.E., Miller, S.P: Specification-Based Prototyping for Embedded Systems. In: Proceedings of the 7th European Software Engineering Conference/ACM SIGSOFT Foundations of Software Engineering. Lecture Notes in Computer Science, **1687** (1999) 163–179
17. Wassyng, A., Lawford, M.: Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project. In: Araki, K., Gnesi, S. Mandrioli, D. (eds), International Symposium of Formal Methods Europe Proceedings. Lecture Notes in Computer Science, **2805** (2003) 133–153