This article was downloaded by: *[Lawford, Mark][Canadian Research Knowledge Network]*
On: *2 November 2010*
Access details: *Access Details: [subscription number 918588849]*
Publisher *Taylor & Francis*
Informa Ltd Registered in England and Wales Registered Number: 1072954 Registered office: Mortimer House, 37-41 Mortimer Street, London W1T 3JH, UK

Transactions of the Royal Society of South Africa
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t917447442

Integrated software methodologies - An engineering approach

Alan Wassyng[a]; Mark Lawford[a]
[a] McMaster Centre for Software Certification, Department of Computing and Software, McMaster University, Hamilton, Canada

Online publication date: 29 October 2010

To cite this Article Wassyng, Alan and Lawford, Mark(2010) 'Integrated software methodologies - An engineering approach', Transactions of the Royal Society of South Africa, 65: 2, 125 — 136
To link to this Article: DOI: 10.1080/0035919X.2010.522449
URL: http://dx.doi.org/10.1080/0035919X.2010.522449

PLEASE SCROLL DOWN FOR ARTICLE

Full terms and conditions of use: http://www.informaworld.com/terms-and-conditions-of-access.pdf

This article may be used for research, teaching and private study purposes. Any substantial or systematic reproduction, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The accuracy of any instructions, formulae and drug doses should be independently verified with primary sources. The publisher shall not be liable for any loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.

Transactions of the Royal Society of South Africa
Vol. 65(2), June 2010, 125–136

UNISA university of south africa PRESS
Taylor & Francis
Taylor & Francis Group

Integrated software methodologies – An engineering approach

Alan Wassyng & Mark Lawford

McMaster Centre for Software Certification, Department of Computing and Software, McMaster University, Hamilton, Canada
e-mail: wassyng@mcmaster.ca / lawford@mcmaster.ca

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.

Keywords: software engineering, software process, formal methods, software requirements, software design, software verification.

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

INTRODUCTION

Software engineering is gradually being recognised 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 realise 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 in deriving architectural design from the requirements (Khedri & Bourguiba, 2004), and there is 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 (Darimont *et al.*, 1997). 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 organised as follows. The first section deals with the development of software methods, starting with typical approaches used up until now, and then the proposed approach, together with an example drawn from an industrial safety-critical project. Tool support is described and some representative related work is discussed. Finally, the conclusions are presented.

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 (1972), *classes* in object-orientated designs) then a *module interface* typically comprises the exported constants and types and the *access programmes* (*methods* in object-orientated designs) of that module. Common wisdom dictates that the interface specification describes the externally visible required behaviour of an access programme 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 original trace assertion method proposed by Bartussek and Parnas (1977). (Parnas now has a revised trace-based method.) However, the thrust of this discussion applies equally well to other methods.

The trace assertion method used module interface call sequences, *traces,* as the components of the specification. In Parnas's design documentation (Parnas & Clements, 1986; Parnas & Madey, 1995) these trace assertions are then used in the *Module Interface Specification.* This specification approach can work but it has an essential 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 require-

ISSN 0035-919X Print / 2154-0098 Online
© 2010 Royal Society of South Africa
DOI: 10.1080/0035919X.2010.522449
http://www.informaworld.com

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

ments document, we may be able to 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 on the code by including cross-references to the design document in the code (see Figure 11.

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. In fact, it would be better if the developers of the software design method have input into the requirements method and its documentation. 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 (1995), using their now famous '4-variable model' (see the top section of Figure 1).

This model relates the *monitored variables* M to the *controlled variables* C through a relation REQ (requirements). M and C represent vector functions of time in the physical domain. The monitored variables are transformed by hardware devices into *input variables* I. These variables are stored in the computer (or, at least, at the boundary of the computer) and are inputs to the software application. The relation SOF represents the design of the software application and so SOF operates on I to produce the *output variables* O. These variables are also stored in the computer and are transformed by hardware into the controlled variables. The functional behaviour of the input and output hardware devices can be represented by the relations IN and OUT. It is clear from Figure 1 that 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 is likely to 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 several 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, we realised in 1993, while developing the first version of the verification method described by Moum (2000),

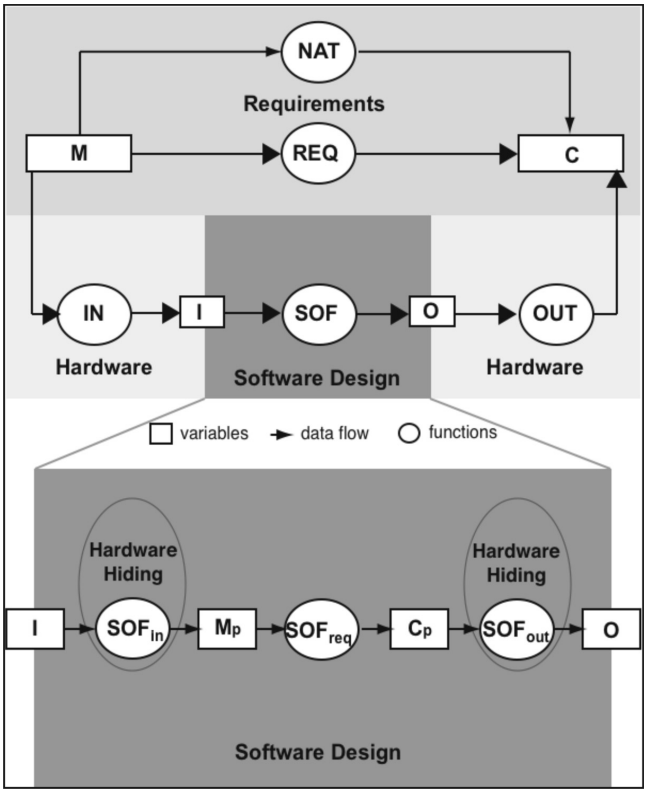


Figure 1. Modified four-variable model.

that we should deal with the software design in a slightly modified way as shown in the bottom expansion in Figure 1. Others have published similar modifications of the 4-variable model (Bharadwaj & Heitmeyer, 2000; Thomson *et al.*, 1999).

Instead of using I to produce O in the software design we can include a little extra software to transform I to a close approximation of M. We call this approximation *pseudo-M* and denote it by M_p. Actually, some software is necessary just to trigger the hardware to produce I in the first place. The simple statements to convert I to M_p add very little complexity, and the trigger software plus the conversion software is packaged in a *hardware hiding* module (Parnas, 1972) represented by SOF_{in}. Similarly, SOF_{out} converts *pseudo-C* (C_p) to O. Since we now have representations of M and C for the software to work with, the software behaviour, represented by SOF_{req}, should be constructed to match the behaviour described by REQ. This means that we can solve the first of our problems, namely that the requirements (REQ) can be used to define the behaviour required in the software design (SOF_{req}). See the discussion on 'References' related to Figure 7, below.

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 programme 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 programme, the requirements functions will not have the necessary granularity.

Let us start by assuming that the requirements specification data-flow topology can be made to match 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

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

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).

SFTs are our solution to the second problem. We develop the SFTs during the software design process and they are then available to document the module interfaces and to aid in the mathematical verification of the software design. Examples of SFTs are shown below. The SFTs serve three important purposes in our integrated methodology.

i) Together with the modification to the 4-variable model, they enable us to document the module interfaces using the requirements specification.
ii) They document how the designers decided to decompose the requirements, so they serve as design notes.
iii) The major step in the design verification can be performed piece-wise. See below. We call the version of the requirements specification that contains SFTs, the *pseudo-requirements specification.*

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 Wassyng and Lawford (2003).

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 (Janicki *et al.,* 1997; Wassyng & Lawford, 2003). Comments in the tables are italicised and are enclosed within braces. As an example, the definitions of the "Neutron OverPower Parameter Trip and Sensor Trips" are shown in Figure 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.

Software design

The software design reorganises 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 (Parnas, 1972) 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 programmes, specifies their invocation syntax and specifies the semantic behaviour of each access programme by referencing the appropriate requirements or pseudo-requirements functions. Figure 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 programme EPTNP is thus documented by listing c_NOPparmtrip in the references section. The internal declarations for module NPParTrip are shown in Figure 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.

Figure 5 shows the detailed design for access programme EPTNP. It shows the call (shown as an external value) to another access programme responsible for maintaining the digital output status (SDONP in DigitalOutput), and the explicit call to a "Get" programme (GSTNP in NPSnrTrip) that provides current values of the 18 NOP sensor trips. In this example there is little difference between detailed design and 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 Figure 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.

The initial version of the module interface specification of the Watchdog module is shown in Figure 7. Note the references for the access programs EWDOG and SWDOG. Different portions of "Watchdog test active" are used in each of those access programmes, so "Watchdog test active" is listed for both of those programmes. 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 programme. However, this is clearly not adequate since we do not know exactly what portions of the function apply to each programme.

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. Figure 8 shows the supplementary function tables that "replace" "Watchdog test active". Note that only two of the three SFTs are used in the Watchdog 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 Figure 9 that the references for the access programmes in the Watchdog module now define the semantic behaviour of the programmes 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.

Software design verification

Motivation: In 1989 some of us were involved in the verification of the first version of the Darlington Shutdown System (Archinoff *et al.,* 1990). 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

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

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) ($f\_NOPsntrip_i$ = e_NotTrip	e_NotTrip

2.3.9.2 NEUTRON OVERPOWER SENSOR TRIPS

Determines whether there is an NOP sensor trip, which is used to determine if 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 i^{th} NOP signal, i=1,..,18	2.4.1.2

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 i^{th} NOP signal *{Calibrated NOP $signal_i$ is now in the trip region}*	e_Trip
f_NOPsp – k_NOPhys < Calibrated i^{th} NOP signal < f_NOPsp *{Calibrated NOP $signal_i$ is now in the deadband region}*	$(f\_NOPsentrip_i)_{-1}$
Calibrated i^{th} NOP signal ≤ f_NOPsp – k_NOPhys *{Calibrated NOP $signal_i$ is now in the non-trip region}*	e_NotTrip

Figure 2. Requirements specification of the NOP parameter trip and sensor trips.

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 Figure 1) is extremely onerous. For deterministic systems this proof obligation is:

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

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

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

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

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

can be performed piece-wise, and after that we can prove REQ_p = REQ. The success of this approach depends on being

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

.23 MODULE NPParTrip

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 DigitalOutput module.

References: *c_NOPparmtrip*

GPTSNP

return: t_boolean

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

References: *c_NOPparmtrip*

IPTNP

Initializes all the NPParTrip module internal states.

References: *Initial Value – c_NOPparmtrip*

Figure 3. Example module interface specification.

4.23.1 MODULE NPParTrip INTERNAL DECLARATIONS

	Name	Value/Origin	Type
Constants:	KNUMNP	Global	t_integer

	Name	Definition/Origin
Types:	t_boolean	

	Name	Type
State Data:	PTSNP	t_boolean

Figure 4. Example module internal declarations.

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 pseudo-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 Figure 10.

The example shows that we "associate" outputs of functions

4.23.1.1 ACCESS PROGRAM EPTNP

	Name	Ext_value	Type	Origin
Inputs:	l_ST	GSTNP(l_ST)	ARRAY 1 TO KNUM NP 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

Figure 5. Example access programme detailed design.

Condition			*Result* Watchdog test active	*Result* $t_{wdstart}$
NOT ['Calibrate enable requested']			False	$t_{now} - k\_WDtestdelay - 1$
'Calibrate enable requested'	'M_RxFnType exists' & [M_RxFnType = e_WD_test]		True	t_{now}
	NOT ['M_RxFnType exists'] OR 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

Figure 6. Requirements description of "Watchdog test active".

between requirements and design using abstraction functions A_i, i =1, 2. 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. Figure 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 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.

Code

The programming languages available on the hardware platform used for the example shutdown system were a dialect of FORTRAN 66 and Assembler. Figure 11 shows an extract from the access programme 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 Figure 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.

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 tables in the design, the code is extremely well-structured. It is relatively easy to reverse-engineer a function table representation of the

4.44 MODULE Watchdog (1.10)
Determines the watchdog system output.

	Name	Value	Type
Constants:	(None)		

	Name	Definition
Types:	(None)	

Access Programs:

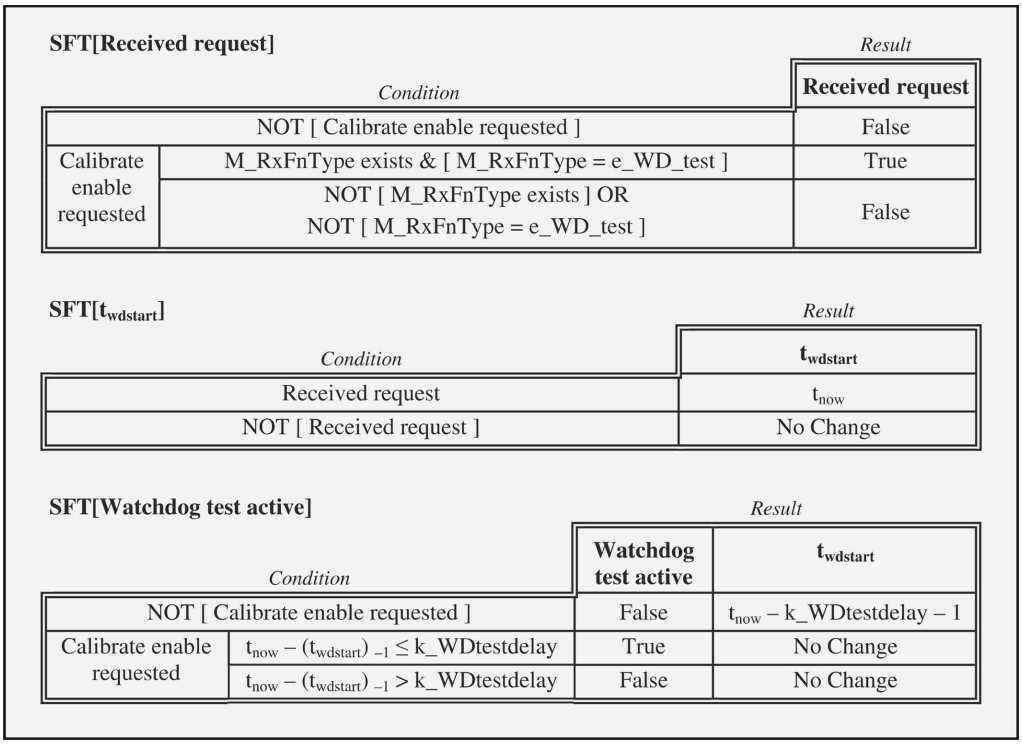
EWDOG
Updates the state of the watchdog timer DO.
References: *c_Watchdog, 'Watchdog test active'*.*

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

Figure 7. Initial module interface specification of watchdog module.

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010



SFT[Received request]

Condition		Result: Received request
NOT [Calibrate enable requested]		False
Calibrate enable requested	M_RxFnType exists & [M_RxFnType = e_WD_test]	True
	NOT [M_RxFnType exists] OR NOT [M_RxFnType = e_WD_test]	False

SFT[$t_{wdstart}$]

Condition	Result: $t_{wdstart}$
Received request	t_{now}
NOT [Received request]	No Change

SFT[Watchdog test active]

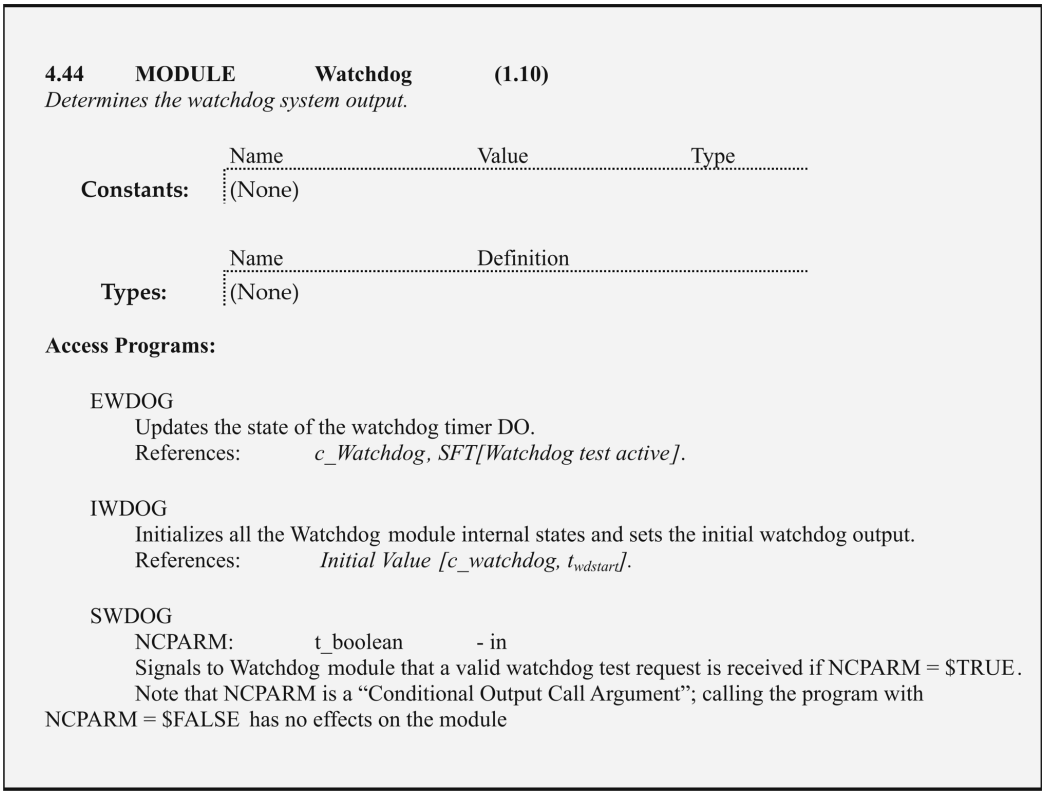
Condition		Result: Watchdog test active	Result: $t_{wdstart}$
NOT [Calibrate enable requested]		False	$t_{now} - k\_WDtestdelay - 1$
Calibrate enable requested	$t_{now} - (t_{wdstart})_{-1} \leq k\_WDtestdelay$	True	No Change
	$t_{now} - (t_{wdstart})_{-1} > k\_WDtestdelay$	False	No Change

Figure 8. Supplementary function tables for “Watchdog test active”.

code. In the example in Figure 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.

SOFTWARE TOOLS

The number of different kinds of software tools is growing rapidly. 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



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 DO.
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.
Note that NCPARM is a “Conditional Output Call Argument”; calling the program with NCPARM = $FALSE has no effects on the module

Figure 9. Revised module interface specification of the watchdog module.

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

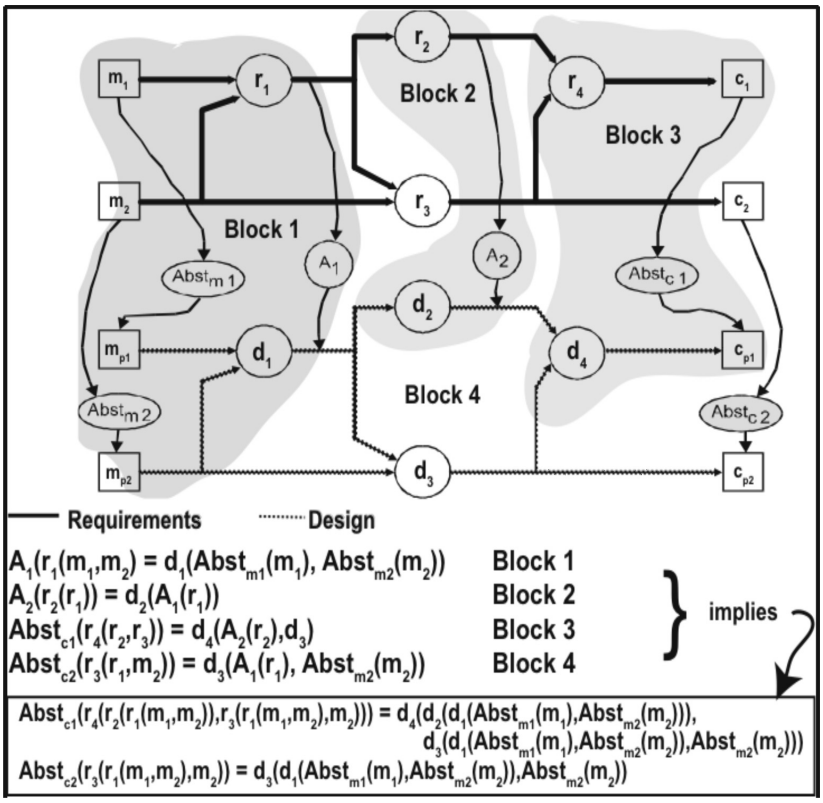


Figure 10. Piece-wise proof example.

successful. For example, although UML (Rumbaugh *et al.*, 2004) 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. The Software Cost Reduction (SCR) Method developed at the US Naval Research Laboratory (NRL) (Heitmeyer, 2002; Heitmeyer *et al.*, 1995, 1998) is also based on Parnas's Rational Design Process, and also makes extensive use of tabular expressions. The NRL focused quite early on the production of software tools but these tools are usually single purpose.

Most software development tools have nothing to do with the kind of tool we are discussing in this paper. In fact, tool integration typically means integration of tools on a specific platform so that users can select an appropriate tool without leaving that platform. The tools may be related in their function, and there may even be multiple tools that perform the same tasks. This is quite different from the type of integration we are proposing. This integration is the use of tools in which the output of one is likely to be used as the input of another, and in which the tools can be used in a cooperative way. More than anything however, integration in our context is the integration of tools in a methodology. The tool suite is designed specifically to support various aspects of the integrated methods.

Tools to support methodologies

Figure 12 shows the tools that were used to support the development of the industrial safety-critical project described by Wassyng and Lawford (2003). Some of the tools were developed specifically for the project. Other tools, such as compilers, theorem provers and configuration management tools, are generic commercial tools that were purchased for use on the project. In retrospect, what is interesting is that it took a number of years before we realised that, although we had tools for every phase of the life-cycle and although many of those tools were built specifically to support a particular phase, most of the tools did not take enough advantage of how thoroughly the life-cycle methods were integrated, as described above.

For instance, we developed tools that supported the production of the software design document. A tool facilitated the actual construction of the physical (Microsoft Word™) document as well as the creation and checking of tabular expressions. What it did not do was use information from the requirements phase to help in the conceptual development of the design. On the other hand, for the software design verification phase the tool extracted information from both the requirements and design phases to help perform automated design verifications using PVS, a commercial theorem prover (Lawford *et al.*, 2000; Owre *et al.*, 1997).

Integrated tools

As a result of our experience, we are now in a position to describe tools that are more integrated with the life-cycle phases and more integrated in terms of combining information from multiple phases. In this section we will discuss three tools. The Design Verification Tool makes use of the formal requirements and design documents together with the verification report. The Identifier Extraction Tool used the source code in the later stages of the project. The final tool we describe was conceived but not ready for use in the project. However, it is a good example of a tool that would never have been considered if we had not already integrated the methods so comprehensively.

The workflow for the Design Verification Tool is shown in Figure 13. This tool automatically generated the proof obligations for the verification blocks shown in Figure 10. It made use of the formal tabular function definitions in the requirements document and design document together with a cross-reference between these two documents that was provided in the design verification report. The cross-reference was manually constructed, providing the break down of the blocks and the mappings between functionality of the requirements and the design. The proof obligations for each block were then fed into

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

```
. . . . .
C     Variable initialization.
C     < SDD output: l_TrpDO >
      LLTRPD = $TRUE
      PTSNP = $TRUE
C --- < VCT EPTNP > ------------------------------------
      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 > ----------------------------------
C     < SDD output call: DigitalOutput.SDONP(l_TrpDO) >
      CALL SDONP(LLTRPD)
. .
```

Figure 11. Code example of access programme EPTNP.

the automated theorem prover PVS where they were often proven without any user interaction. Adhering to strict formatting guidelines in developing the requirements and design documents facilitated the automatic extraction of the proof obligations from Rich Text Format (RTF) versions of the documents. While the version of the tools used in the shutdown systems redesign in 1994–2002 required that the significant parts of the cross-reference be manually entered there is no reason why the cross-reference could not be automated entirely, using the References in the access programmes of the design document such as those shown in Figure 9.

The Identifier Extraction Tool that was actually used in the project was a rather mundane but very effective aid in the code verification process. The software design document, the coding procedure and the code itself were developed taking into account what would be required during the code verification phase. The code verification involves extracting, typing and classifying code variables exactly as they would have been typed and classified in the software design document. The verifier also has to develop a function table representation of the code (for those programmes developed from function tables) and then compare it with the function table in the software design. The code verification document has sections for each code module, and each section is structured to mimic the software design documentation of a module, with some extra information to show the result of the checks that are performed. The tool leaves a placeholder for the extracted function table and for the check items, but creates the remaining documentation directly from the code. Since it turns out that creating the documentation showing the typed and classified variables takes five to eight times longer than it takes to construct a function table representation of the code, the tool dramatically cut the cost of the code verification phase. Future versions of the tool should be able to fill in the checks for most of the items, but we have not yet managed to find a way of generating the function table directly from the code.

The other tool that demonstrates the potential for integrated tools is one that aids in the conceptual development of the software design. During the design phase, we use information hiding and other design quality attributes to successively decompose modules into smaller and smaller modules. Each draft set of modules can be analysed in terms of the "secrets" hidden in the modules (Parnas, 1972), and the responsibility of (service performed by) the module. The secrets come from a list

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

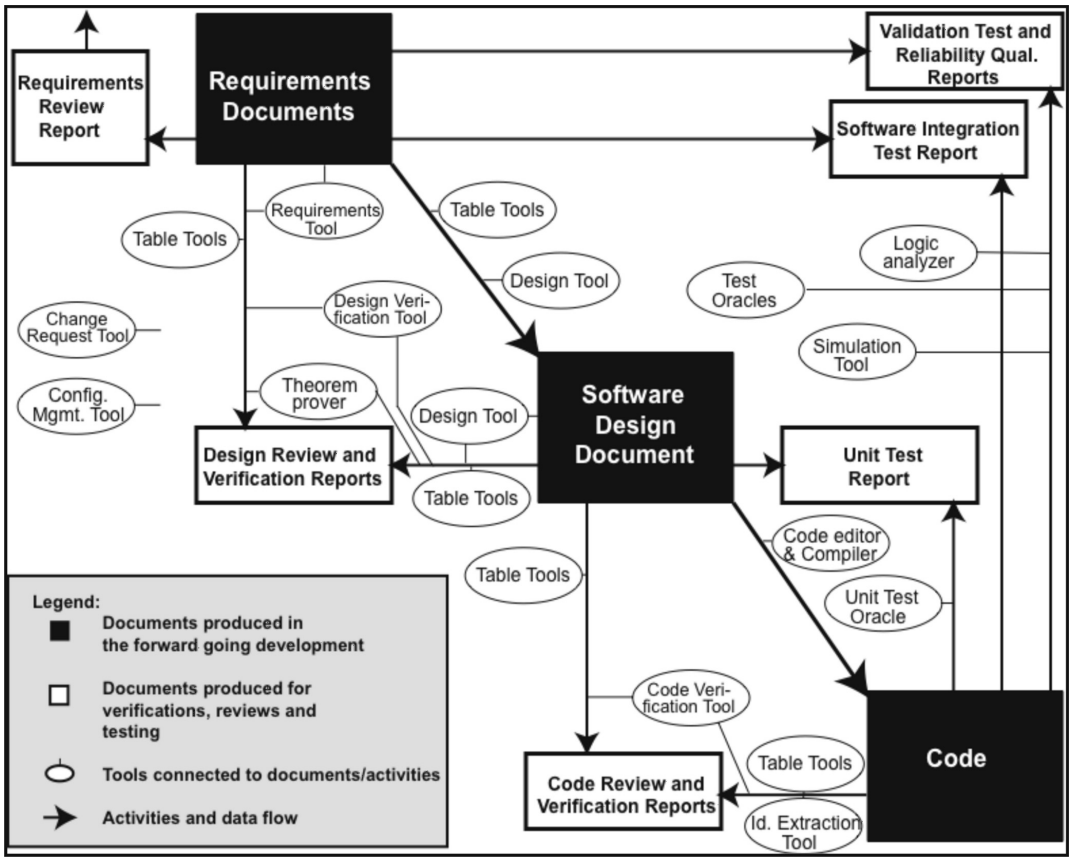


Figure 12. Tools used in a safety-critical industrial project (Wassyng & Lawford, 2006).

of likely changes in the requirements document, augmented by the software designers. The responsibilities come directly from the requirements functions. When we performed this phase during the project, the designers had to keep track of all this information manually. Evaluating different decompositions is extremely time consuming and prone to error. A tool that lists the constants required, the history encapsulated and the requirements functions implemented in each module, as well as the uses relationship, is a tremendous aid to the designers. Although we do not develop supplementary functions at this stage, simply tagging requirements functions that would be split in the design is sufficient. This tool would not automate the design process. It simply serves to collect information from the requirements and partition it according to the trial decomposition. We are still looking for ways of linking secrets to functions or parts of functions that would help to generate potential trial decompositions. We would also need functional decomposition and composition capability in our function table tools that is not yet attainable.

RELATED WORK

Recent works, such as Ainsworth (2008), Post *et al.* (2009) and Whalen *et al.* (2007), advocate integrating formal verification with the software process. Ainsworth (2008) advocates defining a system architecture and then prototyping the development process at the same time as the product. The idea is to quickly implement key system functionality and apply the verification tools to it to de-risk both meeting system requirements and system (re-) certification – two of the most expensive parts of safety critical software development. This approach is merited particularly when we integrate new tools into the process.

Post *et al.* (2009) concluded that there is a need to link require-

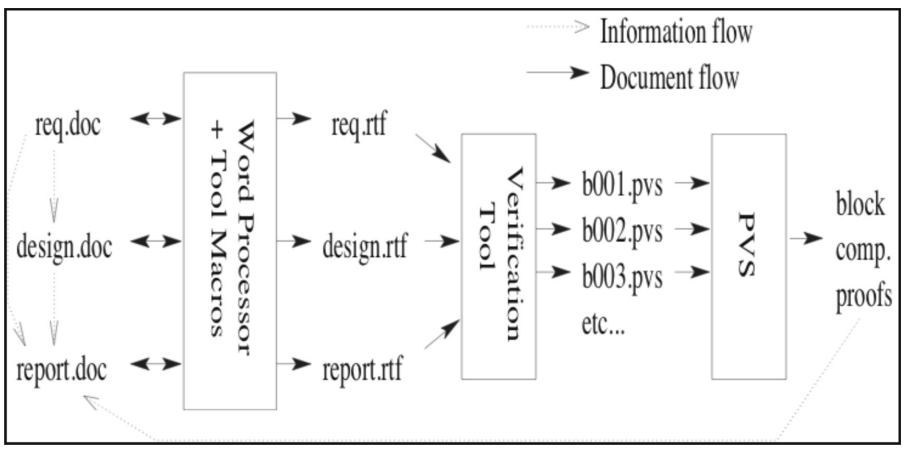


Figure 13. Design verification tool workflow (Wassyng & Lawford, 2006).

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010

ments and formal verification in the same way that most industrial processes link requirements and testing. Their conclusions are motivated by a case study involving some 50 functional requirements for four components of a driver-assistance system being developed by Bosch. A graduate student formalised functional requirements into pre/post conditions in the C code that were then model-checked. While the formal verification found errors that were not detected by testing the effort suffered from the "two versions of the requirements" problem – the informal English language requirements used by the developers and the manually translated pre/post conditions used by the verifier. While bolting a formal method onto an existing development process has minimal disturbance to the company's developers, it increases verification effort and can lead to inconsistencies between formal and informal requirements. As an example of this, one of the errors missed by formal verification but detected by testing was due to an incorrect formalisation of a requirement (Post *et al.*, 2009).

Whalen *et al.* (2007) describe the integration of formal analysis into a model-based software development process for avionics. Informal English language requirements are translated into temporal logic properties of Matlab Simulink/Stateflow subsystem models. The models are translated into input for various implicit state model-checkers by, for example, using custom tools developed at Rockwell-Collins together with the Reactis tool to first translate the models into the synchronous programming language Lustre and then using more tools to translate the Lustre models to input for the different model-checkers. The method was applied to the analysis of a major subsystem of an existing Lockheed Martin operation flight plan system over several revisions. In breaking down the effort in applying the method it was found that "A significant fraction of verification time went towards model preparation because the models were not initially constructed for analysis. Although we were successful, we believe that formal verification can have an even greater impact if its use is anticipated from the outset of the design process" (Whalen *et al.* (2007: 83).

The conclusions of the above works all support the main thesis of this work – that to fully reap the benefits of the rigorous software engineering methods advocated by the research community, we must design integrated methods and tools in the context of the complete software development process to be employed on the product, and the development process must in turn be designed while bearing in mind the methods and tools to be used.

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. For a number of years now software engineers have targeted reuse as a means to achieve reliability and a reduction in the cost of critical software applications. Integrated methods allow us to benefit from reuse of specifications, documentation and tools in a much more comprehensive way. The methods that we have 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. Many years after the work on developing the methodology, when we examined lessons learned from our experience in both creating the methods and in applying them to a real project, we realised the opportunities that had been opened to us as a result of this very simple principle. Industrial pressure also dictated that a good process and a good tool were good enough and, during the project, 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 at reasonable cost without adequate tool support. Tool support can vary from mundane labour savers to very sophisticated analysis and code generation tools. People talk about integrated tools and tool chains but we believe that current tool chains will seem meagre in the future if we concentrate on producing integrated tool chains from integrated methods. Our experience has been that these comprehensive and integrated methods drive completely different 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

Ainsworth, M. 2008. Prototyping versus formal development. In Redmill, F. & Anderson, T. (Eds) *Improvements in Systems Safety. Proccedings of the Sixteenth Safety-critical Symposium*, Bristol, UK, 5–7 February 2008. Springer. pp. 195–207.

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

Bartussek, W. & Parnas, D.L. 1978. Using assertions about traces to write abstract specifications for software modules. In: *Proceedings of the 2nd Conference of European Cooperation in Informatics, Venice, 1978*. *Lecture Notes in Computer Science* **65**: 211–236. Berlin, Springer; reprinted in Gehani, N. & McGettrick, A.D. (Eds), Software Specification Techniques 1985: 111–130.

Bharadwaj, R. & Heitmeyer, C.L. 2000. Developing high assurance avionics systems with the SCR requirements method. *Proceedings of the 19th Digital Avionics Systems Conference, Philadelphia*. pp. 1–8.

Darimont, R., Delor, E., Massonet, P. & Van Lamsweerde, A. 1977. GRAIL/KAOS: An environment for goal-driven requirements engineering. *Proceedings of the 19th International Conference on Software Engineering.* pp. 612–613.

Heitmeyer, C. 2002. Software cost reduction. In Marciniak, J.J. (Ed.) *Encyclopedia of Software Engineering,* 2nd edition. New York, John Wiley & Sons. pp. 1374–1380.

Heitmeyer, C., Bull, A., Gasarch, C. & Labaw, B. 1995. Toolset for specifying and analyzing requirements. *10th Annual Conference on Computer Assurance, Gaithersburg, Maryland.* pp. 109–122.

Heitmeyer, C., Kirby, J., Labaw B. & Bharadwaj, R. 1998. Toolset for specifying and analyzing software requirements. Proceedings of 10th International Conference on Computer Aided Verification, Vancouver, BC, Canada. *Lecture Notes in Computer Science* **1427**: 526–531.

Janicki, R., Parnas, D.L. & Zucker, J. 1997. Tabular representations in relational documents. In Brink, C., Kahl, W. & Schmidt, G., (Eds) *Relational Methods in Computer Science. Advances in Computing Science* **12**: 184–196.

Khedri, R. & Bourguiba, I. 2004. 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* pp. 356–365.

Lawford, M., McDougall, J., Froebel, P. & Moum, G. 2000. Practical application of functional and relational methods for the specification

and verification of safety critical software. Proceedings of AMAST 2000, Iowa City, Iowa, USA. *Lecture Notes in Computer Science* **1816**: 73–88.

Moum, G. 2000. Systematic design verification procedure, Revision 02, NK38-MAN-68000-001. Ontario Power Generation, Darlington NGD Shutdown System Trip Computer Software. Proprietary. 104 pp.

Owre, S., Rushby, J. & Shankar, N. 1997. 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**: 366–383.

Parnas, D.L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* **15**(12): 1053–1058.

Parnas, D.L. & Clements, P. 1996. A rational design process: how and why to fake it. *IEEE Transactions Software Engineering* **12**(2): 251–257.

Parnas, D.L. & Madey, J.1995. Functional documents for computer systems. *Science of Computer Programming* **25**: 41–61.

Post, H., Sinz, C., Merz, F., Gorges, T. & Kropf, T. 2009. Linking functional requirements and software verification. *Proceedings of the 17th IEEE International Requirements in Engineering Conference. pp. 295–302.*

Rumbaugh, J., Jacobson, I. & Booch, G. 2004. *The Unified Modeling Language Reference Manual,* 2nd edition. Boston, Pearson Higher Education. 567 pp.

Thompson, J.M., Heimdahl, M.P.E. & Miller, S.P. 1999. Specification-based prototyping for embedded systems. *Proceedings of the 7th European Software Engineering Conference/ACM SIGSOFT Foundations of Software Engineering*. *Lecture Notes in Computer Science* **1687**: 163–179.

Wassyng, A. & Lawford, M. 2003. 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**: 133–153.

Wassyng, A. & Lawford, M. 2006. Software tools for safety-critical software development. *International Journal of Software Tools for Technology Transfer* **8**(4–5): 337–354.

Whalen, M., Cofer, D., Miller, S., Krogh, B.H. & Storm, W. 2008. Integration of formal analysis into a model-based software development process. In Leue, S., Merino, P. (Eds) *Proceedings of Formal Methods in Industry Critical Systems, 2007. Lecture Notes in Computer Science* **4916**: 68–84.

Downloaded By: [Lawford, Mark][Canadian Research Knowledge Network] At: 14:05 2 November 2010