Practical Application of Functional and Relational Methods for the Specification and Verification of Safety Critical Software

Mark Lawford[1]*, Jeff McDougall[2], Peter Froebel[3], and Greg Moum[3]

[1] Computing and Software, McMaster Univ., Hamilton, ON, Canada, L8S 4L7
`lawford@mcmaster.ca`
[2] JKM Software Technologies Inc., 17160 Keele Street, R.R.1, Kettleby, ON, L0G 1J0
`jkm@pathcom.com`
[3] Ontario Power Generation, 700 University Ave., Toronto, ON, Canada M5G 1X6
`peter.froebel, g.moum@ontariopowergeneration.com`

Abstract. In this paper we describe how a functional version of the 4-variable model can be decomposed to improve its practical application to industrial software verification problems. An example is then used to illustrate the limitations of the functional model and motivate a modest extension of the 4-variable model to an 8-variable relational model. The 8-variable model is designed to allow the system requirements to be specified as functions with input and output tolerance relations, as is typically done in practice. The goal is to create a relational method of specification and verification that models engineering intuition and hence is easy to use and understand.

1 Introduction

The CANDU Computer Systems Engineering Centre of Excellence Standard for Software Engineering of Safety Critical Software [4] states the following as its first fundamental principle: "*The required behavior of the software shall be documented using mathematical functions in a notation which has well defined syntax and semantics.*" In order to achieve this, Ontario Power Generation Inc.[1] (OPGI) and Atomic Energy of Canada Limited (AECL) have jointly defined a detailed software development process to govern the specification, design and verification of safety critical software systems. The software development process results in the production of a coherent set of documents that allow for static analysis of the properties of the design described in the Software Design Description (SDD), comparing them against the requirements described in the Software Requirements Specification (SRS).

In this work we review how this functional verification has been done in practice [5]. We first describe how a functional version of the 4-variable model of

* Partially supported by NSERC grant 217249-99.
[1] OPGI is the electricity generation company created from Ontario Hydro.

[11] can be decomposed to facilitate tool support and to reduce the manual effort required to perform and document the specification and verification tasks. An example from [5] is then used to motivate the extension to a relational model. The relational model is necessary to rigorously account for tolerances that must normally be considered when trying to implement a system to meet an ideal system behavior.

We propose the 8-variable model, a modest extension of the 4-variable model that takes into consideration input and output tolerances while still permitting the use of functional requirements specification and design descriptions. The model formalizes the engineering practice of appealing to tolerances when necessary. By attempting to formalize the process, the authors hope to provide a sound basis for tool support of the entire verification process and provide opportunities for further applications of fundamental relational algebraic concepts.

Section 2 provides an overview of the basic concepts and notation required by the paper. Section 3 explains the (functional) Systematic Design Verification (SDV) procedure and its limitations regarding tolerances. These limitations motivate the 8-variable model of Section 4.

2 Preliminaries

Functions and relations are shown in italics (e.g., f, REQ). All sets of time series vectors from the 4- and 8-variable models are shown bold (e.g., $\mathbf{BM}$). All other mathematical terms are shown in italics (e.g., $bm \in \mathbf{BM}$).

For a set V_i, we will denote the *identity map on the set* V_i by id_{V_i} (i.e., $id_{V_i} : V_i \to V_i$ such that $v_i \mapsto v_i$). Given functions $f : V_1 \to V_2$ and $g : V_2 \to V_3$, we will use $g \circ f$ to denote *functional composition* (i.e., $g \circ f(v_1) = g(f(v_1))$). The *cross product* of functions $f : V_1 \to V_2$ and $f' : V_1' \to V_2'$, defines a function $f \times f' : V_1 \times V_1' \to V_2 \times V_2'$ such that $(v, v') \overset{f \times f'}{\mapsto} (f(v), f'(v'))$.

It will also be convenient to consider the operation of *relational composition*. For $F \subseteq V_1 \times V_2$ and $G \subseteq V_2 \times V_3$, $F \bullet G = \{(v_1, v_3) \in V_1 \times V_3 | (\exists v_2 \in V_2) : (v_1, v_2) \in F \wedge (v_2, v_3) \in G\}$. Thus for the functions f and g as defined above, $g \circ f = f \bullet g$. A relation F is said to be *total* if $(\forall v_1 \in V_1)(\exists v_2 \in V_2) : (v_1, v_2) \in F$.

Denote the set of all equivalence relations on V by $Eq(V)$. Any function $f : V \to R$ induces an equivalence relation $\ker(f) \in Eq(V)$, the equivalence kernel of f, given by $(v_1, v_2) \in \ker(f)$ if and only if $f(v_1) = f(v_2)$. We define the standard partial order on equivalence relations as follows. Given equivalence relations $\theta_1, \theta_2 \in Eq(V)$, we say that θ_1 *is a refinement of* θ_2, written $\theta_1 \leq \theta_2$, iff each cell (equivalence class) of θ_1 is a subset of a cell of θ_2 (i.e., $(v, v') \in \theta_1$ implies $(v, v') \in \theta_2$ for all $(v, v') \in V \times V$). We can now formally state a basic existence claim for functions that will be used later in the paper.

Claim. Given two functions with the same domain, $f : V_1 \to V_3$ and $g : V_1 \to V_2$, there exists $h : V_2 \to V_3$ such that $f = h \circ g$ iff $\ker(g) \leq \ker(f)$.

3 Functional Systematic Design Verification

This section provides an overview of the (functional) SDV procedure used in [5]. We highlight elements of the process, such as the decomposition of proof obligations, that facilitate tool support and reduce the effort required to perform and document the SDV procedure. Although the SDV procedure covers other types of verification problems, we will concentrate on the verification of simple functional properties that often compose the majority of system requirements. The reader is referred to [8] for the complete procedure. The section concludes with an example that illustrates the limitations of the functional approach.

3.1 SDV Procedure Overview

The software engineering process described here is based upon the *Standard for Software Engineering of Safety Critical Software* [4] that was jointly developed by OPGI and AECL. This standard requires that the software development and verification be broken down into series of tasks that result in the production of detailed documents at each stage. The software development stages relevant to this paper are governed by the Software Requirements Specification Procedure [3] and the Software Design Description Procedure [6]. These procedures respectively produce the Software Requirements Specification (SRS) and Software Design Description (SDD) documents. In addition to other methods, these documents make use of a form of Parnas' tabular representations of mathematical functions [2, 10] to specify the software's behavior. Tables provide a mathematically precise notation (see [1] for the formal semantics) for the SRS and SDD in a visual format that is easily understood by domain experts, developers, testers, reviewers and verifiers alike [5].

The underlying models of both the SRS and SDD are based upon Finite State Machines (FSM). The SDD adds to the SRS functionality the scheduling, maintainability, resource allocation, error handling, and implementation dependencies. The specification technique for defining the implementation is based on a virtual machine which will execute the source code which is to be implemented. The primary difference between this virtual machine and the FSM describing the SRS is that execution is not instantaneous, but takes a finite amount of time, and thus the order of execution must be specified to avoid race conditions. The SRS is produced by software experts with the help of domain experts. It is used by lead software developers to produce the SDD which is then used by all the developers to produce the actual source code.

The software engineering standard [4] requires that the SDD be formally verified against the SRS and then the code formally verified against the SDD to ensure that the implementation meets the requirements. These formal verifications are governed by the SDV Procedure and Systematic Code Verification (SCV) Procedure. For the purposes of this paper we will concentrate on the SDV process.

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

Fig. 1. Commutative diagram for 4 variable model

The objective of the SDV process is to verify, using mathematical techniques or rigorous arguments, that the behavior of every output defined in the SDD, is in compliance with the requirements for the behavior of that output as specified in the SRS. It is based upon a variation of the four variable model of [11] that verifies the functional equivalence of the SRS and SDD by comparing their respective one step transition functions. The resulting proof obligation in this special case:

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

is illustrated in the commutative diagram of Figure 1. Here REQ represents the SRS state transition function mapping the monitored variables $\mathbf{M}$ (including the previous pass values of state variables) to the controlled variables and updated (current) state represented by $\mathbf{C}$. The function SOF represents the SDD state transition function mapping the behavior of the implementation input variables represented by statespace $\mathbf{I}$ to the behavior of the software output variables represented by the statespace $\mathbf{O}$. The mapping IN relates the specification's monitored variables to the implementation's input variables while the mapping OUT relates the implementation's output variables to the specification's controlled variables. The following section briefly outlines the refinement of the relational methods in [11] to the simple functional case in (1).

3.2 Specialization of the 4-Variable Model

In the 4-variable model of [11], each of the 4 "variable" state spaces $\mathbf{M}$, $\mathbf{I}$, $\mathbf{O}$, and $\mathbf{C}$ is a set of functions of a single real valued argument that return a vector of values - one value for each of the quantities or "variables" associated with a particular dimension of the statespace. For instance, assuming that there are n_M monitored quantities, which we represent by the variables $m_1, m_2, \ldots, m_{n_M}$, then, the possible timed behavior of the variable m_i can be represented as a function $m_i^t : \mathbb{R} \rightarrow Type(m_i)$, where $m_i^t(x)$ is the value of the quantity m_i at time x. We can then take $\mathbf{M}$ to be the set of all functions of the form $m^t(x) = (m_1^t(x), m_2^t(x), \ldots, m_{n_M}^t(x))$. Thus the relations corresponding to the arrows of the commutative diagram then relate vectors of functions of a single real valued argument.

In order to simplify the 4-variable model to a FSM setting, we restrict ourselves to the case where each of the 4 "variables" $\mathbf{M}$, $\mathbf{I}$, $\mathbf{O}$, and $\mathbf{C}$ is a set of "time series vectors". For example, $\mathbf{M}$ actually refers to all possible sets of observations ordered (and equally spaced) in time, each observation being a vector of n_M values. We will use the term *monitored variable* to refer to quantity m_i which is the ith element in the vector ($i \in \{1, \ldots, n_M\}$). Let $m \in \mathbf{M}$ be a time series vector of observations of the monitored variables. With a slight abuse of notation, we will use $m_i(z)$ to denote the zth observation of the ith element ($z \in \{0, 1, 2, \ldots\}$) of the monitored variables for the time series vector m. Similarly $m(z)$ represents the zth observation of the n_M values in the monitored variable vector for time series m.

For this model, the time increment between each of the observations is defined to be the positive real value $\delta > 0$. Thus observation z corresponds to time $(z * \delta)$. The increment δ is taken to be at least an order of magnitude less than any time measurements of interest. The value of m_i at any point between two observations (i.e., in the range of time $[z * \delta, (z + 1) * \delta)$) is defined to be equal to $m_i(z)$.

Each of the "variables" $\mathbf{M}, \mathbf{C}, \mathbf{I}, \mathbf{O}$ in the specialized 4-variable model has the same frequency of observation, but may have a different number of values in its vector. The value n_M is defined to be the number of elements in $\mathbf{M}$, which are observed over time, while n_I is defined to be the number of elements in $\mathbf{I}$, which are observed over time. Normally $n_I = n_M$. Similarly n_O is defined to be the number of elements in $\mathbf{O}$, which are observed over time and n_C is defined to be the number of elements in $\mathbf{C}$, which are observed over time. Normally, $n_C = n_O$.

Requirements (REQ): The required behavior of the subsystem is described with REQ. At OPGI REQ is modeled as a FSM, defining a relation over $\mathbf{M} \times \mathbf{C}$. While, in general, the FSM could be nondeterministic, much of the system behavior can be modeled by a deterministic FSM with the result that for the verification of these properties we can assume that REQ is a function (i.e., $REQ : \mathbf{M} \to \mathbf{C}$).

In this case a new set of time series vectors, $\mathbf{S}$, is introduced to describe the state of the FSM. Let $c \in \mathbf{C}$, $m \in \mathbf{M}$, $s \in \mathbf{S}$, and $z \in \{0, 1, 2, \ldots, \}$. The zth value of a controlled variable time series vector $c(z)$ depends on both the values of $m(z)$ and $s(z)$, related by the vector function $OUTPUT$ (i.e., $c(z) = OUTPUT(m(z), s(z))$). Also, the value $s(z + 1)$ depends on both the values of $m(z)$ and $s(z)$, related by the vector function $NEXTSTATE$. (i.e., $s(z + 1) = NEXTSTATE(m(z), s(z))$.

The SRS procedure [3] shows how a set of functions $f_1, f_2, ...f_j$ can be defined such that when a subset of them are composed, they define the OUTPUT function. When a different, though not necessarily disjoint, set of them are composed, they define the $NEXTSTATE$ function. We have called the process of defining these functions the "decomposition of REQ".

Design (SOF): The implemented behavior of the subsystem is described with SOF. SOF can be modeled as a directed graph with $p + 2$ nodes. Within this graph, each node is either one of p FSM, or $\mathbf{I}$, or $\mathbf{O}$, and each edge represents data

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

Fig. 2. Vertical decomposition: Isolation of hardware hiding proof obligations

flow between two of these. The node containing $\mathbf{I}$ must not be the destination of any edge. The node containing $\mathbf{O}$ must not be the source of any edge. In this way, SOF defines a relation over $\mathbf{I} \times \mathbf{O}$. If the design is produced following the SDD procedure, then each of the FSMs represents a program called from the mainline in the design. We assume a constant mainline loop structure, with each program called 1 or more times within the loop. If called more than once, calls are assumed to be evenly spaced within the loop. The loop is assumed to take a constant amount of time to execute.

For a large number of the implementation properties, the FSMs composing SOF can be modeled as deterministic FSM allowing us to consider the special case when SOF defines a function. In this case, when both REQ and SOF are functions, if we are also able to restrict ourselves to functional maps for IN and OUT, we can verify the commutative diagram of Figure 1 by comparing the one step transition functions of the FSMs defining REQ and SOF. More detailed descriptions of the underlying SRS and SDD models can be found in [3] and [6], respectively, as well as [8].

3.3 Decomposing the Proof Obligations

In Figure 2 we decompose the proof obligation (1) to isolate the verification of hardware interfaces. The $\mathbf{M}_p$ and $\mathbf{C}_p$ state spaces are the software's internal representation of the monitored and controlled variables, referred to as the *pseudo-monitored* and *pseudo-controlled variables*, respectively. The proof obligations associated with SDV then become

$$Abst_C \circ REQ = SOF_{req} \circ Abst_M \tag{2}$$
$$Abst_M = SOF_{in} \circ IN \tag{3}$$
$$id_{\mathbf{C}} = OUT \circ SOF_{out} \circ Abst_C \tag{4}$$

The first of these equations represents a comparison of the functionality of the system and should contain most of the complexity of the system. The last two represent comparisons of the hardware hiding software of the system. These obligations are often fairly straightforward and are discharged manually.

As an example to help the reader interpret the above decomposition, suppose an actual physical monitored plant parameter belonging to $\mathbf{M}$ is the temperature of the primary heat transport system which might have a current value of 500.3 Kelvin. The hardware corresponding to the temperature sensors and A/D converters might map this via IN to a value of 3.4 volts in a parameter in $\mathbf{I}$. A hardware hiding module might then process this input corresponding to map SOF_{in}, producing a value of 500 Kelvin in the appropriate temperature variable belonging to the software state space $\mathbf{M}_p$. Further "vertical" decomposition is performed by isolating outputs and in effect restricting $\mathbf{M}$ and projecting $\mathbf{C}$ to the variables relevant to a particular subsystem such as the pressure sensor trip described in the Section 3.4.

The observant reader may have noted that the controlled variable abstraction function is defined as $Abst_C : \mathbf{C} \to \mathbf{C}_p$ which is seemingly the "wrong" direction. The proof obligation (4) forces $Abst_C$ to be invertible, preventing the possibility of trivial designs for SOF_{req} being used to satisfy the main obligation (2). As we will see below, this allows the verifier to define only one abstraction mapping for each pair of corresponding SRS and SDD state variables that occur as both inputs and outputs in the decomposition. The SDV procedure provides recourse for the case when there is not a 1-1 correspondence between $\mathbf{C}$ and $\mathbf{C}_p$ through the use of a pseudo-SRS that can be defined to more closely match the SDD. The interested reader is referred to [7] for further details.

Typically the verification of a subsystem as represented by (2), the inner part of the commutative diagram, can be decomposed "horizontally" at both the SRS and SDD level into a sequence of intermediate verification steps, thereby reducing the larger, more complex proof obligation into a number of smaller, more manageable verification tasks. This is represented in Figure 3 where each equality of the form

$$SOF_i \circ Abst_{V_{i-1}} = Abst_{V_i} \circ REQ_i \qquad (5)$$

becomes a verification block. The price paid for this vertical and horizontal decomposition is that for each block the verifier must provide a cross reference between the internal variables making up the $\mathbf{V}_{i-1}, \mathbf{V}_i$ state spaces at the SRS level and the internal variables making up the $\mathbf{V}_{(i-1)p}, \mathbf{V}_{ip}$ state spaces at SDD level, as well as defining the abstraction functions, $Abst_{V_{i-1}}$ and $Abst_{V_i}$. Now the benefits of defining all the abstraction functions, including $Abst_C$, from top to bottom (SRS to SDD) in Figures 2 and 3 becomes more apparent. The values of many of the controlled variables from the previous execution pass of the SRS and SDD often become inputs to the calculation of current internal state and output variables. Similarly, state variables that are the output of one sequential block become the input of the following block. Defining all abstraction functions from top to bottom and then only performing the check for invertibility at the outputs embodied by (4) allows the verifier to use the same abstraction functions whether a state variable occurs at the input or output of a block. This technique reduces the number of abstraction functions required by up to one half.

$$\begin{tikzcd}
\mathbf{M} \arrow[r, "REQ_1"] \arrow[d, "Abst_M"'] \arrow[rrrrr, bend left, "REQ"] & \mathbf{V}_1 \arrow[r, "REQ_2"] \arrow[d, "Abst_{V_1}"'] & \mathbf{V}_2 \arrow[d, "Abst_{V_2}"'] & \cdots & \mathbf{V}_{n-1} \arrow[r, "REQ_n"] \arrow[d, "Abst_{V_{n-1}}"] & \mathbf{C} \arrow[d, "Abst_C"] \\
\mathbf{M}_p \arrow[r, "SOF_1"] \arrow[rrrrr, bend right, "SOF_{req}"'] & \mathbf{V}_{1p} \arrow[r, "SOF_2"] & \mathbf{V}_{2p} & \cdots & \mathbf{V}_{(n-1)p} \arrow[r, "SOF_n"] & \mathbf{C}_p
\end{tikzcd}$$

Fig. 3. Horizontal (sequential) decomposition of proof obligations

3.4 Limitations of a Functional Model

The following example uses a simplified sensor trip to demonstrate how the verification task can be partitioned, and highlights the limitations of the functional SDV procedure regarding support for tolerances. Currently the verification tool suite described in [5] only supports functional verification. Work remains to be done on the incorporation of tolerances on inputs and outputs through the use of relational methods. Often SRS and SDD behaviors are not functionally equivalent, but they are within specified tolerances. Currently in these cases, separate rigorous arguments must be made, appealing to tolerances to explain any differences in functionality. Ideally, one would like to be able to use formal mathematical proofs incorporating the tolerances when necessary without an excessive increase in proof complexity and workload associated with the documentation. In many cases it should be possible to add existential quantifiers for variables with tolerances and then make minor modifications to the original theorem statements.

We now describe the verification of a simplified pressure sensor trip that monitors a pressure sensor and initiates a reactor shutdown when the sensor value exceeds a normal operating setpoint. We will use tabular specifications for their readability. In all of the tables of Figure 4, the functions return the value in the right column when the condition in the left column is satisfied.

Tables $f\_PressTrip$ and $PTRIP$ in Figure 4 give the proposed SRS and SDD implementations respectively for the sensor trip. The SRS specification of the pressure sensor trip makes use of deadbands to eliminate sensor chatter. In the function definitions, $f\_PressTripS1$ and $PREV$ play corresponding roles as the arguments for the previous value of the state variable computed by the function. The verification is performed using SRI's Prototype Verification System (PVS) automated proof assistant [9, 12] to handle typechecking and proof details.

sentrip : THEORY
BEGIN

k_PressSP : int = 2450

k_DeadBand : int = 50

KDB : int = k_DeadBand

KPSP : int = k_PressSP

Trip : TYPE = {Tripped, NotTripped}

AI : TYPE = subrange(0, 5000)

f_PressTrip((Pressure : posreal), (f_PressTripS1 : Trip)) : Trip = TABLE

Pressure ≤ k_PressSP − k_DeadBand	NotTripped
k_PressSP − k_DeadBand < Pressure ∧ Pressure < k_PressSP	f_PressTripS1
Pressure ≥ k_PressSP	Tripped

ENDTABLE

PTRIP((PRES : AI), (PREV : bool)) : bool = TABLE

PRES ≤ KPSP − KDB	FALSE
KPSP − KDB < PRES ∧ PRES < KPSP	PREV
PRES ≥ KPSP	TRUE

ENDTABLE

Trip2bool((TripVal : Trip)) : bool = TABLE

TripVal = Tripped	TRUE
TripVal = NotTripped	FALSE

ENDTABLE

posreal2AI((x : posreal)) : AI = TABLE

$x \leq 0$	0
$0 < x \wedge x < 5000$	floor(x)
$x \geq 5000$	5000

ENDTABLE

Sentrip1 : THEOREM
  (∀ (Pressure : posreal, f_PressTripS1 : Trip) :
    Trip2bool(f_PressTrip(Pressure, f_PressTripS1)) =
      PTRIP(posreal2AI(Pressure), Trip2bool(f_PressTripS1)))

END sentrip

Fig. 4. Formatted PVS specification for pressure sensor trip example

Figure 4 also contains the supporting type, constant and abstraction function definitions for the verification block. The abstraction function $posreal2AItype$ models the A/D conversion of the sensor values by taking the integer part of its input using the built in function $floor(x)$ from the PVS prelude file. It is used to map the real valued SRS input $Pressure$ to the discrete SDD input $PRES$ which has type $AIType$. $AIType$ consists of the subrange of integers between 0 and 5000, denoted by $subrange(0, 5000)$ in Figure 4.

At the bottom of the specification, the theorem $Sentrip1$ is an example of a *block comparison theorem* that is used to prove a specific instance of the general block verification equation (5) that relates the SRS and SDD inputs and outputs. If $Pressure$ and $PRES$ were both real numbers, related by the identity map, then the block comparison theorem $Sentrip1$ would be easily proved, but in this case, where $PRES$ is a discrete input, when attempting the block comparison, PVS reduces the problem to attempting to prove that for all input values the following equation holds:

$$\neg(f\_PressTripS1 = Tripped \wedge floor(Pressure) \leq 2400 \\ 2400 < Pressure < 2450)$$

For any value of $Pressure$ in the open interval $(2400, 2401)$ when $f\_PressTrip$ was tripped in the previous pass, the above formula is $FALSE$. The problem occurs because whenever $2400 < Pressure < 2401$, the abstraction function $posreal2AIType$ maps $Pressure$ to the same value as 2400, but when $f\_PressTripS1 = Tripped$, the SRS function $f\_PressTrip$ maps $Pressure$ values greater than 2400 to $Tripped$ while 2400 gets mapped to $NotTripped$. In other words, $\ker(posreal2AIType \times Trip2bool) \not\leq \ker(f\_PressTrip)$ so we know by the claim from Section 2 that there is no SDD design that can satisfy the block comparison theorem $Sentrip1$. The interested reader is referred to [5] for further details on the use of PVS in this example and the OPGI systematic design verification process in general.

This is an example of when functional equality is more strict than practically necessary. Due to the accuracy of the sensors and A/D hardware in the actual implementation, all input values have a tolerance of ± 5 units. In this case, the SDD function $PTRIP$ actually provides acceptable behavior.

The generalized relational version of the 4-variable model originally put forward in [11] easily handles this case by allowing REQ, IN and OUT from Figure 1 to be relations between state spaces consisting of sets of vectors of functions of time. We have found that the majority of system properties making up REQ are, in practice, easily specified and verified if they are modeled as deterministic state machines where the $NEXTSTATE$ and $OUTPUT$ functions have tolerances associated with their inputs and outputs as outlined below.

4 The 8-Variable Model

When trying to compare the ideal required behavior to a design, allowable tolerances must be specified precisely so that it can be determined whether a

$$\begin{tikzcd}
\mathbf{FM} \arrow[r, "MTRAN"] & \mathbf{BM} \arrow[r, "MTOL"] \arrow[d, "IN"'] & \mathbf{IM} \arrow[r, "REQ"] & \mathbf{IC} \arrow[r, "CTOL"] & \mathbf{BC} \arrow[r, "CTRAN"] & \mathbf{FC} \\
& \mathbf{I} \arrow[rrr, "SOF"] & & & \mathbf{O} \arrow[u, "OUT"'] &
\end{tikzcd}$$

Fig. 5. 8-Variable Model

proposed design exhibits acceptable behavior. The 8-variable model shown in Fig. 5 has been developed to take tolerances into account in such a way that the ideal required behavior from the functional 4-variable model of Section 3.2 needs no modification. Thus domain experts still specify the behavior functionally as REQ, with tolerances embodied by the $MTOL$ and $CTOL$ (i.e., REQ is a function) while $MTOL$ and $CTOL$ are tolerance relations. To facilitate the specification of tolerance relations, the monitored and controlled state spaces **M** and **C** of the original 4-variable model are replaced by pairs of state spaces **BM**, **IM** and **IC**, **BC** respectively. The additional state spaces **FM** and **FC** are used to model the actual physical quantities "in the field" and, although not always necessary for the verification task, they can be used to document the monitored and controlled quantities relation to the physical world to provide the system engineers with insight into the problem domain.

In this model, **FM** refers to the *Field Monitored Variables*. These are mathematical variables[2] which model properties of the environment that are being measured. For example, the heat transport temperature in degrees C might be modeled by a field monitored variable.

BM refers to the *Boundary Monitored Variables*. These are mathematical variables which model properties being measured at the limits of the subsystem being described. For example, if the subsystem was a trip computer, a voltage at the terminal blocks of the computer could be modeled by a boundary monitored variable. Although the required behavior of the subsystem only needs to describe behavior relative to boundary monitored variable values, it is convenient for the specifier writing the requirements to give each boundary monitored variable a name which reflects the property being modeled by the associated field monitored variable. For example, the SRS may name a boundary monitored variable *m_HT_Temperature*, even though it is a voltage value which is being modeled at the boundary. This allows the requirements specifier to concentrate on the problem domain when describing the required behavior of the subsystem. Typically there is a functional relationship $MTRAN$ between **FM** and **BM**.

[2] As opposed to software variables.

IM refers to the *Inner Monitored Variables.* Inner monitored variables are mathematical variables which represent the boundary monitored variables, with Monitored Variable Accuracy taken into account.

Monitored variable accuracy describes a range of values, such that the subsystem is required to respond as described to at least one of the values within that range. For example, if in the example above the boundary monitored variable accuracy is ± 0.5 V, and the value of the boundary monitored variable at some point in time is 2.5 V, then the requirements are saying that the subsystem may respond as if the value at that time is any one of the values in the range $[2.0, 3.0]$ V. In this way, monitored variable accuracy results in the SRS describing a set of allowable behaviors. Any design which exhibits one of the allowable behaviors, meets the requirements.

The ideal required behavior in the SRS (i.e., REQ), when applied to the inner monitored variables, provides a description of required behavior which accounts for decisions regarding accuracy. For example, the requirement specifier can use a condition $m\_HT\_Temperature > m\_HT\_Setpoint$, rather than explaining within the condition how to account for the accuracy of the variable $m\_HT\_Temperature$. Note that in some cases, **IM** may be the same as **BM**. For example, in a trip computer a configuration EPROM value does not change on-line and is represented as a digital value. Thus the accuracy would be ± 0. The $MTOL$ relation between **BM** and **IM** is typically not a functional relationship, since one value in a boundary monitored variable is related to many values in the corresponding inner monitored variable when there is a non-zero monitored variable accuracy associated with the variable.

I refers to the *Input Variables.* Input variables are mathematical variables which model the information available to the software. For example, a voltage which is converted to a digital value via an A/D converter may be made available to the software as a base-2 integer in a register. The value read from that register could be modeled as an input variable. The IN relation between **BM** and **I** is usually not a functional relationship. This relation takes into account quantization of values (i.e., loss of accuracy due to constructing a discrete representation of a continuous quantity) and hardware inaccuracies (e.g., an A/D converter tolerance).

O refers to the *Output Variables.* Output variables are mathematical variables which model the values set by the software. For example, if the software sets a bit in a register to indicate that a trip should occur, the bit could be modeled as an output variable.

IC refers to the *Inner Controlled Variables.* Inner controlled variables are mathematical variables which represent the boundary controlled variables, before Controlled Variable Accuracy is taken into account.

Controlled variable accuracy describes a range of values, such that the response of the subsystem is required to equal a value which is within that range around the value described by the ideal required behavior. For example, if a boundary controlled variable accuracy is ± 0.1 V, and the ideal required value of the controlled variable at a point in time is 2.1 V, then the requirements are saying that the subsystem may respond at that time with a result which is any one of the values in the range $[2.0, 2.2]$ V. As with monitored variable accuracy, controlled variable accuracy results in the SRS describing a set of allowable behaviors. Any design which exhibits one of the behaviors allowed is thereby meeting the requirements.

BC refers to the *Boundary Controlled Variables*. These are mathematical variables which model properties being controlled at the limits of the subsystem being described. For example, a voltage produced by the subsystem at the terminal blocks of the computer could be modeled by a boundary controlled variable. Note that, as with boundary monitored variables, it is convenient for the specifier writing the requirements to give each boundary controlled variable a name which reflects the property being modeled by the associated field controlled variable. The $CTOL$ relation between **IC** and **BC** is not functional since one value in an inner controlled variable can be related to many values in the corresponding boundary controlled variable. The OUT relation between **O** and **BC** is not typically functional since it takes into account hardware inaccuracies (e.g., D/A converter tolerance).

FC refers to the *Field Controlled Variables*. These are mathematical variables which model properties of the environment that are being controlled. Typically there is a functional relationship $CTRAN$ between **BC** and **FC**. If **BC** and **FC** are not the same variables, then the system level documentation should describe the transformation between them.

Collectively, the variables **FM**, **BM** and **IM** will be referred to as Monitored Variables. Similarly, **IC**, **BC** and **FC** will be referred to as Controlled Variables.

Monitored Variable Accuracy If each boundary monitored variable bm_i, has an accuracy requirement $+a_i/-b_i$, where $a_i \geq 0$ and $b_i \geq 0$, then $MTOL$ defines a relation over $\mathbf{BM} \times \mathbf{IM}$ such that $(bm, im) \in MTOL \iff$

$$(\forall z \in \{0,1,2,\ldots\})(\forall i \in \{1,2,\ldots n_M\}) bm_i(z) - b_i \leq im_i(z) \leq bm_i(z) + a_i$$

Controlled Variable Accuracy If each boundary controlled variable bc_i, has an accuracy requirement $+c_i/-d_i$, where $c_i \geq 0$ and $d_i \geq 0$, then $CTOL$ defines a relation over $\mathbf{IC} \times \mathbf{BC}$ such that $(ic, bc) \in CTOL \iff$

$$(\forall z \in \{0,1,2,\ldots\})(\forall i \in \{1,2,\ldots n_C\})(ic_i(z) - d_i \leq bc_i(z) \leq ic_i(z) + c_i)$$

4.1 Design Verification

From Fig. 5, there are two "paths" from **BM** to **BC**. The first path is via **IM** and **IC**, with $MTOL$, REQ, and $CTOL$. The second path is via **I** and **O**, with IN, SOF, and OUT. More precisely, when $MTOL$, REQ and $CTOL$ are composed, they describe the $REQUIREMENTS$ relation which is the subset of $\mathbf{BM} \times \mathbf{BC}$ given by: $REQUIREMENTS = MTOL \bullet REQ \bullet CTOL$

Similarly, when IN, SOF and OUT are composed, they describe the relation $DESIGN = IN \bullet SOF \bullet OUT \subseteq \mathbf{BM} \times \mathbf{BC}$.

Design verification of functional requirements with tolerances is then the process of showing two things:

$$DESIGN \subseteq REQUIREMENTS, \text{ and} \tag{6}$$
$$DESIGN \quad \text{is total} \tag{7}$$

Thus, by (6), all behaviors that the design may exhibit represent acceptable behavior according to the requirements, and by (7) the design is defined for all possible values of boundary monitored variables. Note that together conditions (6) and (7) guarantee that $REQUIREMENTS$ is total so that the acceptable behavior of the system has been completely specified for all possible monitored variable values.

It may be that the designer does not want or need $REQUIREMENTS$ to be complete (e.g., in case when there are input combinations that are physically impossible). This happens when the system being controlled, the "plant", places restrictions on how the monitored variables can be related to the controlled variables. In the standard 4-variable model of [11] this is modeled by the relation $NAT \subseteq \mathbf{M} \times \mathbf{C}$. In the case of the proposed 8-variable model we could have $NAT \subseteq \mathbf{BM} \times \mathbf{BC}$. In this case (7) could be replaced by the requirement:

$$NAT \cap REQUIREMENTS \subseteq DESIGN \tag{8}$$

to guarantee that (6) being met by a non-trivial design.

4.2 The Simplified Sensor Trip Revisited

For the simple sensor trip example in Section 3.4, $MTRAN$ can be used to define the mapping from the physical pressure in kPa to a real valued sensor output voltage while $CTRAN$ relates the $Tripped/NotTripped$ value of $f\_PressTrip$ to the $Open/Closed$ state of a physical relay. The ± 5 tolerance on the input due to the uncertainty in the sensors and A/D conversion is modeled by $MTOL$ and $CTOL$ is just the identity map due to the discrete nature of the trip output. The remaining maps REQ, SOF, IN and OUT are still be modeled by the functions $f\_PressTrip$, $PTRIP$, $posreal2AI$ and the inverse of $Trip2bool$, respectively.

Using PVS's dependent typing capabilities, the block comparison theorem $Sentrip1$ can be restated as the easily proved theorem:

```
Sentrip1 : THEOREM
   (∀ (Pressure : posreal, f_PressTripS1 : Trip)
       (∃ (Pressure2 : {(x : posreal)|Pressure − 5 ≤ x ≤ Pressure + 5}) :
     Trip2bool(f_PressTrip(Pressure2, f_PressTripS1)) =
       PTRIP(posreal2AI(Pressure), Trip2bool(f_PressTripS1))))
```

5 Conclusion

The main goals of this paper have been to provide insight into how relational methods can be adapted to increase their utility in practical applications. We have outlined a functional specification and verification technique for safety critical software based upon the four variable model of [11]. A simple example was used to illustrate the limitations of a functional model and motivate a modest extension of the theory to a relational setting that we call the 8-variable model. The main benefit of this model refinement of the relational 4-variable model is that the method is intuitive and easy to use for both requirements specification and design description since engineers typically prefer to think in terms of functions with tolerances when dealing with safety critical systems.

References

1. R. Janicki and R. Khédri. On a formal semantics of tabular expressions. *Science of Computer Programming*, 2000. To appear.
2. R. Janicki, D. Parnas, and J. Zucker. Tabular representations in relational documents. In C. Brink et al., editors, *Relational Methods in Computer Science*, Advances in Computing Science, ch. 12, p. 184–196. Springer Wien NewYork, 1997.
3. E. Jankowski and J. McDougall. Procedure for the Specification of Software Requirements for Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1001-PROC Rev. 1, July 1995.
4. P. Joannou *et al.* Standard for Software Engineering of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1, January 1995.
5. M. Lawford and P. Froebel. Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Submitted to FMSD.
6. J. McDougall and J. Lee. Procedure for the Software Design Description for Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1002-PROC Rev. 1, October 1995.
7. J. McDougall, M. Viola, and G. Moum. Tabular representation of mathematical functions for the specification and verification of safety critical software. In *SAFECOMP'94*, p. 21–30, Anaheim, October 1994. Instrument Society of America.
8. G. Moum. Procedure for the Systematic Design Verification of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1003-PROC Rev. 1, December 1997.
9. Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In Ed Brinksma, editor, *TACAS '97*, LNCS 1217, p. 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.
10. D. Parnas. Tabular representation of relations. Technical Report 260, Communications Research Laboratory, McMaster University, October 1992.
11. D. Parnas and J. Madey. Functional documentation for computer systems engineering. Technical Report CRL No. 273, Telecommunications Research Institute of Ontario, McMaster University, September 1991.
12. N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.