Precise Documentation and Validation of Requirements

Jonathan S. Ostroff, Chen-Wei Wang and Simon Hudon

Technical Report CSE-2013-08

August 27 2013

Department of Computer Science and Engineering
4700 Keele Street, Toronto, Ontario M3J 1P3 Canada

PRECISE DOCUMENTATION & VALIDATION OF REQUIREMENTS

JONATHAN S. OSTROFF, CHEN-WEI WANG, SIMON HUDON

Abstract. This paper outlines an approach to precise documentation and validation of system requirements. Precise documentation of requirements is important for developing and certifying mission critical software. Function tables have been used to document specifications of software components that are complete and disjoint. In this paper, function tables are embedded in an event based structure allowing requirements validation by proving invariants expressing safety properties. Function tables usually involve the use of total functions, or partial functions that are extended to be total. However, it is often convenient to use queries involving partial functions in specifications with preconditions defining the valid domain. Their use in tabular expressions raises the issue of whether the expressions in a table are well-defined. We organize queries involving partial functions in modules and specify them with contracts. We then propose a method for precise documentation, where requirements elicited from customers are expressed as atomic, natural language descriptions that are translated into tabular expressions referencing the specification modules, and into global properties expressed as invariants. We develop a calculus to prove that the tabular expressions are well-defined and that they entail the global properties. A biomedical device is used to illustrate our method for precise documentation and validation.

Contents

1. Introduction

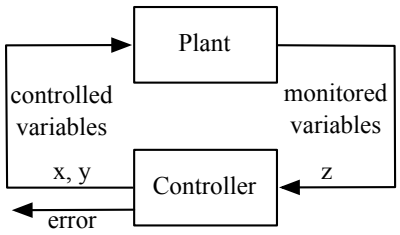
Precise documentation of requirements is important for developing and certifying mission critical software, e.g. medical devices, nuclear reactors and high assurance business systems [12]. It is known that significant problems are likely to occur at the requirements stage [11] thus underscoring the need for precise requirements. Standards such as IEEE 7-4.3.2 (nuclear), ISO 26262 (automotive) and DO-178C (avionics) recommend the use of formal methods, which would also depend on precisely documented requirements.

1.1. **Tabular expressions.** Experience with documentation written in a variety of natural languages has shown natural language to be inadequate for the task of precise requirements specification, as there are usually unsuspected ambiguities. Ambiguities of this sort are resolved by the use of precise mathematical descriptions of the product, which offer the promise of concise, precise description [18].

Tabular expressions have been used to provide mathematical descriptions of requirements, such as those of a nuclear power plant [18, 22]. The computer controller is, at first, represented by a "black box", which relates responses generated by the system, to stimuli received by the system. The relationship is described by a mathematical function, specified by "function tables" (also called tabular expressions). The function tables are more practical than conventional mathematical notation because the functions usually have a great many points of discontinuity and the discontinuities can occur at arbitrary points in the domain [18].

Consider a computer controller embedded in a larger environment as in the context diagram Fig. 1(a). In [22], stimuli from the plant (environment) are referred to as monitored variables and responses are controlled variables. A variable such as z (in Fig. 1(a)) represents the current value of a monitored variable and $z_{\text{-}1}$ refers to its value in the previous state. The system behaviour is modelled as a finite state machine. At discrete points in time, the system detects the *current* values of all monitored variables, and uses the current state of the machine (and, possibly, past history) to generate the current values of the controlled variables and the next state of the machine. Fig. 1(d) provides an (artificial) example of a function table for our small system. For example, if $z \geq 0$

(a) Context diagram



(b) Module

module *system* **use** MATH
variables
$x, y, z : \mathbb{R}$
$error : \mathbb{B}$
init $x, y, z := 0, 0, 0$
event
execute
any r
when $-43.2 \leq r$
then z := r
||« Table 1(*d*) »
end

(c) ADT for Real Numbers

type MATH
const $\epsilon : \mathbb{R}$
axiom $0 < \epsilon \leq 0.001$
query $sqrt(r : \mathbb{R}) : \mathbb{R}$
require $r \geq 0$
ensure
$-\epsilon \leq r - Result^2 \leq \epsilon$
$0 \leq Result$

(d) Function table for module *system* in Fig. 1

	$x : \mathbb{R}$	$error : \mathbb{B}$	$y : \mathbb{R}$ |
$z \geq 0$	$sqrt(z)*x_{\text{-}1} + y$	*false*	$y_{\text{-}1}*x_{\text{-}1} \leq y$
$-43.2 \leq z < 0$	no change	*true*	

assume: $-43.2 \leq z$

Figure 1. A Small System

Input conditions		*Output* r |
$C_1(x)$	$C_{11}(x)$	$R_1(x,r)$
	$C_{12}(x)$	$R_2(x,r)$
$C_3(x)$		$R_3(x,r)$
assume: A		

Given : $P_1 \triangleq C_1(x) \wedge C_{11}(x)$
$P_2 \triangleq C_1(x) \wedge C_{12}(x)$
$P_3 \triangleq C_3(x)$
$Q_i \triangleq R_i(x,r)$ for $i \in 1\,..\,3$

(a) **Meaning of Table** : $A \ \wedge (\forall i \in 1\,..\,3 \bullet P_i \Rightarrow Q_i)$
(b) **Completeness** : $A \ \wedge (\exists i \in 1\,..\,3 \bullet P_i)$
(c) **Disjointness** : $A \ \wedge (\forall i, j \in 1\,..\,3 \mid i \neq j \bullet \neg(P_i \wedge P_j))$
(d) **Well-definedness** : $\mathcal{D}(A) \ \wedge \ (A \wedge (\forall i \in 1\,..\,3 \bullet \mathcal{D}(P_i) \wedge (P_i \Rightarrow \mathcal{D}(Q_i))))$

FIGURE 2. Completeness, Disjointness and Well-definedness of tabular expressions

(the current value of z is not negative), then output x is described by the before-after predicate $x = sqrt(z)*x_{\_1} + y$, i.e., the predicate expresses how the current value of output x depends on the current values of y and input z and the previous value of x.

Periodically, the plant generates a new value, say $z \in \mathbb{R}$, which is monitored by the controller. In response, the controller generates a new value for the controlled variables $x, y \in \mathbb{R}$ based on the monitored variables and past history. A real system will have many more monitored and controlled variables than shown in Fig. 1(a). However, our small system can be used to illustrate some aspects of our method for precise documentation of software requirements, representing many real software intensive products embedded in a larger system.

The state machine description is an idealized view of the required behaviour (suitable for a requirements document) as outputs are generated instantaneously once the input is received. Where necessary, accuracy and timing tolerances can be supplied within which the final implementation must operate (see [24]). Tabular expressions usually involve the use of total functions: partial functions need to be expanded to total functions, or dependent subtyping is used, to keep the use of the functions *well-defined*.

In any real system it will not be possible to describe the behaviour in a single function. Instead, the requirements include a number of inter-acting functions, which themselves are represented by function tables. As stated in [22, 20, 10, 23], the function tables must be complete and disjoint as shown in Fig. 2. Completeness ensures that all possible inputs are covered. Disjointness ensures that there are no conflicts in the outputs.

A mathematically precise requirements document is an essential prerequisite for the development of safety critical software. Domain experts can review them for correctness. Programmers can use them for design and coding. For regulatory authorities, such a document provides greater assurance that the software is precisely specified in a way that will not exhibit unintended, unsafe behaviour.

1.2. **Contribution of this paper to precise requirements.** In Fig. 1(a), the plant periodically generates new values for the monitored variable z. In order to describe and analyze the behaviour of the system consisting of both the action of the plant and the controller, we embed the function table of the controller in an Event-B style machine as shown in Fig. 1(b).

Event-B [1] is a notation and method for discrete systems modelling by refinement. Event-B models are described in terms of the two basic constructs *contexts* and *machines*. Contexts contain the static part of a model (e.g. carrier sets, constants, axioms and theorems) whereas machines contain the dynamic part (e.g. state variables, invariants and events). An initialization statement establishes an initial state of the system by assigning suitable values to these variables. The events determine what can happen in the system during an execution and are described via a before-after predicate. The execution terminates when no action is enabled anymore.

Let variables v define the state of the system. An event is composed of a guard $G(r, v)$ and an action $S(r, v)$ where r is parameter: a value chosen non-deterministically every time the event is executed. An event is represented using the syntax: **any** r **where** $G(z, v)$ **then** $S(r, v)$. Actions are relations between two states that may be specified with assignments that may be non-deterministic. For example for state variables x, y and z, the action $x := x + z \parallel y := y - x$ may also be written as the before-after predicate $x, y :| x = x_{\text{-}1} + z_{\text{-}1} \ \wedge \ y = y_{\text{-}1} - x_{\text{-}1}$.

Queries (with pre/post conditions) such as *sqrt* in Fig. 1(c) are not directly supported by function tables and Event-B. As mentioned above, function tables are constructed from total functions. However, it is often more convenient to allow the use of partial functions (specified with a precondition) directly without any conversion. The precondition of a partial function captures in one place where the function can meaningfully be applied, thus providing a logical "firewall" between the specifier and the client so that functions are not misused on meaningless inputs. See the discussion on design-by-contract in [14].

Contributions of this paper:

(1) *Description of system behaviour with events and function tables.* It is often the case that many details of a complex environment (the plant) can be ignored by implementors and reviewers if they are given a complete and precise specification of the black-box input-output behaviour of the computer controller (based on the feedback of domain experts). We use an Event-B style machine to provide a precise description of the system behaviour involving the plant and the controller as in Fig. 1(b). The controller action is specified with a function table (for completeness and disjointness). The underlying event structure provides us with machinery to describe and analyze the system behaviour under the specified descriptions, before design and implementation.

(2) *Queries and well-definedness.* We provide a method for introducing queries such as *sqrt* in Fig. 1(c), defined via pre/post conditions. Queries are useful in the construction of complex expressions in events, invariants and function tables. Queries introduce the possibility of a the query result being undefined if it is used in a context that does not satisfy its precondition. We thus develop a theory of well-definedness to ensure that the expressions (in function tables, guards and invariants) are well-defined.

(3) *Decomposition into modules and types.* We allow variables and associated queries to be collected in modules. This is particularly useful when describing complex systems. If a module does not contain any variables then we call it a type (e.g. type MATH in Fig. 1(c)). Only the main module *system* may have events. Other modules do not contain any events, only related variables, queries and invariants. If module m_2 uses module m_1, then the queries and invariants of m_2 may use the variables and queries of m_1. In a large system, we partition the state into modules so as to allow a separation of concerns. We may always flatten all the modules into a single Event-B machine.

(4) *Validation of requirements via proofs of invariants.* To our knowledge, the literature does not discuss the proof of invariants in systems specified by function tables. These invariants may describe important system safety requirements. Using our calculus of well-definedness we can prove these invariants in the framework developed above. Suppose we would like to prove the invariant $J(v)$ where v is the state variables of the system. As in Event-B, the proof obligation is $J(v_{\text{-}1}) \wedge G_{execute}(v_{\text{-}1}) \wedge BA_{execute}(v_{\text{-}1}, v) \Rightarrow J(v)$, where $G_{execute}$ is the event guard and $BA_{execute}$ is the before-after predicate of the event action specified by the function table.

(5) E/R descriptions. It is useful to retain informal English language statements describing the system requirements (R-descriptions) and relevant phenomena and constraints on the environment (E-descriptions).

As mentioned above in the first contribution, we embed the black-box function table describing the controller in event *execute* as shown in Fig. 1(b). Suppose the domain experts inform us of that there is an environmental constraint such that the plant monitored variable z will always satisfy $(-42.3 \leq z)$. This constraint, we document as an E-description. We can constrain the system behaviour accordingly via a guard $(-42.3 \leq r)$ for event *execute*. When analyzing the function table in Fig. 1(d) for completeness and disjointness, we may add an assume clause with the relevant constraint on z.

Event *execute* thus defines the system behaviour as follows. The action of the plant is modelled using the **any** construct, the event generates an arbitrary value for parameter r constrained only by its guard. Because of assignment action $z := r$, the event guard places constraint $(-42.3 \leq z)$ on the plant monitored variable z. At the same time, the event updates the controlled variables x and y (modelling the action of the controller) as specified by the function table in Fig. 1(d).

Event-B was designed to act as a simple foundation on top of which other constructs (such as our function tables, queries and modules) can be built. The synergy of function tables and events means that: (a) the controller specification is complete and disjoint, and (b) the overall system behaviour is precisely defined by Event-B semantics.

As a simple example of a requirement, the domain experts might specify that $(0 \leq x \ \wedge \ 0 \leq y)$ hold in all states. This is a system safety property that we must prove given the specification of the computer controller. We document this requirement as an R-description. We may express the safety requirement as invariant $J(x, y.z)$ where $J(x, y, z) \triangleq (0 \leq x \ \wedge \ 0 \leq y)$. The event guard is $G_{execute} \triangleq -43.2 \leq z$ and before-after predicate is $BA_{execute} \triangleq G_{execute} \Rightarrow \beta$, where

$$\begin{array}{rcl} \beta & = & (z \geq 0 \Rightarrow x = Sqrt(z){*}x_{-1} + y) \\ & \wedge & (-43.2 \leq z < 0 \Rightarrow x = x_{-1}) \\ & \wedge & y_{-1}{*}x_{-1} \leq y \end{array}$$

By propositional logic, we know $G_{execute} \wedge BA_{execute} \equiv G_{execute} \wedge \beta$. A simple proof that the invariant holds is as follows.

Prove: $0 \leq x_{-1} \wedge 0 \leq y_{-1} \wedge G_{execute} \wedge \beta \ \Rightarrow \ 0 \leq x \wedge 0 \leq y$

Proof of $0 \leq y$

$$\begin{array}{ll} & 0 \leq y \\ \Leftarrow & \quad \langle \text{ transitivity } \rangle \\ & 0 \leq x_{-1}{*}y_{-1} \wedge x_{-1}{*}y_{-1} \leq y \\ = & \quad \langle \ x_{-1}{*}y_{-1} \leq y \ \equiv \ \mathit{true} \text{ by BA } \rangle \\ & 0 \leq x_{-1}{*}y_{-1} \wedge \mathit{true} \\ \Leftarrow & \quad \langle \text{ arithmetic } \rangle \\ & 0 \leq x_{-1} \ \wedge \ 0 \leq y_{-1} \end{array}$$

Proof of $0 \leq x$

Case $0 \leq z$:

$$\begin{array}{ll} & 0 \leq x \\ = & \quad \langle \text{ BA for } x \ \rangle \\ & 0 \leq \mathit{sqrt}(z){*}x_{-1} + y \\ \Leftarrow & \quad \langle \ 0 \leq \mathit{sqrt}(z) \ \rangle \\ & 0 \leq y \end{array}$$

Case $z < 0$:

$$\begin{array}{ll} & 0 \leq x \\ = & \quad \langle \text{ BA for } x \ \rangle \\ & 0 \leq x_{-1} \end{array}$$

The well-definedness of the above proof obligation (as will be explained in the sequel) reduces $-43.2 \ \leq \ z \wedge z \ \geq \ 0 \Rightarrow z \ \geq \ 0$. The proof obligation and its well-definedness condition can be discharged using a theorem prover. We show this for z3 SMT solver in Appendix B (on p20).

The rest of the paper is divided into the following sections. In Section 2, we show how we support queries and well-definedness allowing for expressive ease in describing complex systems. In section 3, we present a case study of a biomedical device using the framework developed in this paper. In Section 4, we dicuss related and future work.

1.3. **Notation.** The basic types are $\mathbb{B}$ (boolean), $\mathbb{N}$ (naturals), $\mathbb{Z}$ (integers), $\mathbb{R}$ (reals), and $\mathbb{S}$ (strings). We use the mathematical conventions of Event-B for sets, relations and functions

(e.g. dom, ran, #, functional application, etc.). $D \to R$ is the set of all total functions with domain D and range a subset of R. $D \nrightarrow R$ is the corresponding set of all partial functions. If f_1 and f_2 are functions, and S is a set, then $S \lhd f_1$ is domain subtraction and the over-riding operator is $f_1 \lhd\!\!\lhd f_2$.

The integer interval $m \mathinner{..} n$ is defined as follows: $m \mathinner{..} n \triangleq \{i : \mathbb{Z} \bullet m \leq i \leq n\}$. The real interval is defined as follows: $[x, y] \triangleq \{z : \mathbb{R} \bullet x \leq z \leq y\}$. We let $x \uparrow y$ (respectively, $x \downarrow y$) stand for the maximum of x and y (respectively, the minimum). We allow quantification for symmetric and associative binary operators such as $\uparrow$ as in [4]. Event-B's mechanized theorem prover does not support real numbers, so we stick to handwritten proofs using the equational logic of [4] using the one point rule, Leibniz, etc. It is possible to use other theorem provers or SMT solvers, but that is beyond the scope of this paper.

Event-B does not directly support parameterized data structures such as $SEQ[G]$, which is sequence in generic parameter G. We thus introduce the appropriate machinery for parameterized data structures that also allow for the use of queries, as illustrated in Fig. 5(a). A sequence in generic parameter G is an element of $1 \mathinner{..} n \to G$, where $n \in \mathbb{N}$. We eliminate parameter n by using generalized union: $SEQ[G] \triangleq (\bigcup n : \mathbb{N} \bullet 1..n \to G)$. The expression $(\bigcup n : \mathbb{N} \bullet 1..n \to G)$ is the set of all total functions whose domain is the contiguous interval of integers starting at 1 and whose range is a subset G. Suppose we define a variable $v\colon SEQ[\mathbb{R}]$ in a module. This means that the module has invariant $v \in (\bigcup n : \mathbb{N} \bullet 1..n \to G)$. If a query q in the sequence type has a first argument of this type, then we may use the dot notation $v.q$ instead of $q(v)$. For example, we may write $v.has(2.5)$ instead of $has(v, 2.5)$

2. Well-definedness of Expressions with Partial Functions

It is often useful to have queries whose values are not defined for all their possible arguments. It is the case for example with the *sqrt* query shown above: no meaningful result can be given for negative numbers. This raises the question of what status to give to expressions like $\sqrt{-1}$.

In classical tabular expressions [9, 19], all partial functions are transformed into total functions by extending the range of functions with a special undefined value. However, the logic used is still a two-valued predicate logic. This is achieved by defining any expression involving an undefined term to evaluate to false in an assignment. Predicates are identified with their satisfying assignments (so that $1 \div x = 1 \div x$ effectively reduces to $x \neq 0$). Advantages of the approach are that the logic is kept simple, the assigned meanings are consistent with intuitive interpretations, and the expressions are simpler in certain cases while preserving two valued logic. However, complements will not always work (e.g. $\sqrt{x} > \sqrt{y}$ and $\sqrt{x} \leq \sqrt{y}$ can be simultaneously false) and complexity reappears in the axiomatic definitions of the functions (requiring the introduction of an undefined value). Also, conventional simplification rules, and hence some automatic simplifiers and verifiers would need to be modified or used with caution as they are often based on the implicit assumption that functions are total. Even worse, it allows the expression of nonsensical properties in specifications without flagging any problem.

The use of queries presumes that functions will be partial. We thus seek a logic where we can introduce and reason with partial functions without the need to constantly convert them into total functions. In the logic that we adopt in the sequel, the predicate $1 \div x = 1 \div x$ does not pass a well-definedness check (done using proof obligations in a standard theorem prover). However, $(x \neq 0) \wedge (1 \div x = 1 \div x)$ is well-defined and it can then be submitted to the theorem prover as if all functions were total (the prover will fail to prove it as a theorem). We thus are able to introduce partial functions (without converting them into total functions) while using standard tools and mathematical conventions.

Given an expression *exp*, we provide in the technical report [15] a recursive definition of the predicate $\mathcal{D}(exp)$ which holds when *exp* is well-defined. For example, for a variable x we have that

$\mathcal{D}(x) \triangleq true$. The well-definedness of a query application $q(x)$ with precondition $C_q(x)$ is defined as $\mathcal{D}(q(x)) \triangleq \mathcal{D}(x) \wedge C_q(x)$.

As shown in [15] whenever we are asked to prove predicate β_q holds, where β_q involves a query q, we need to discharge two proof obligations. The first obligation is that we must show that $\mathcal{D}(\beta_q)$ holds (this is usually relatively simple). The second obligation is to show that β_q holds. Both proofs can be conducted using a normal prover that treats all functions as total. From there, it is simple to envision applying the $\mathcal{D}$ operator to the meaning of function tables shown in Fig. 2. More technical details on well-definedness are provided in Appendix A.

3. Case Study: A Biomedical Device

We apply our proposed approach to document the precise requirements of a biomedical, pulse device supplied by our industrial partner. In Fig. 3(a) we have identified the boundary of the pulse software and its operating environment. The device monitors vital signs such as blood pressure, heart rate and temperature. A reading from the device arrives as a sampled pulse (e.g., see Fig. 3(c))). This is the monitored variable *swf* (sampled waveform) which we represent as a finite sequence of real numbers ($SEQ[\mathbb{R}]$). The sampled waveform is a plot of pressure levels (vertical y axis) versus time instants (horizontal x axis). As shown in Fig. 3(b), given a sampled pulse, the software is required to generate three outputs: 1) a detailed report on parameters whose values

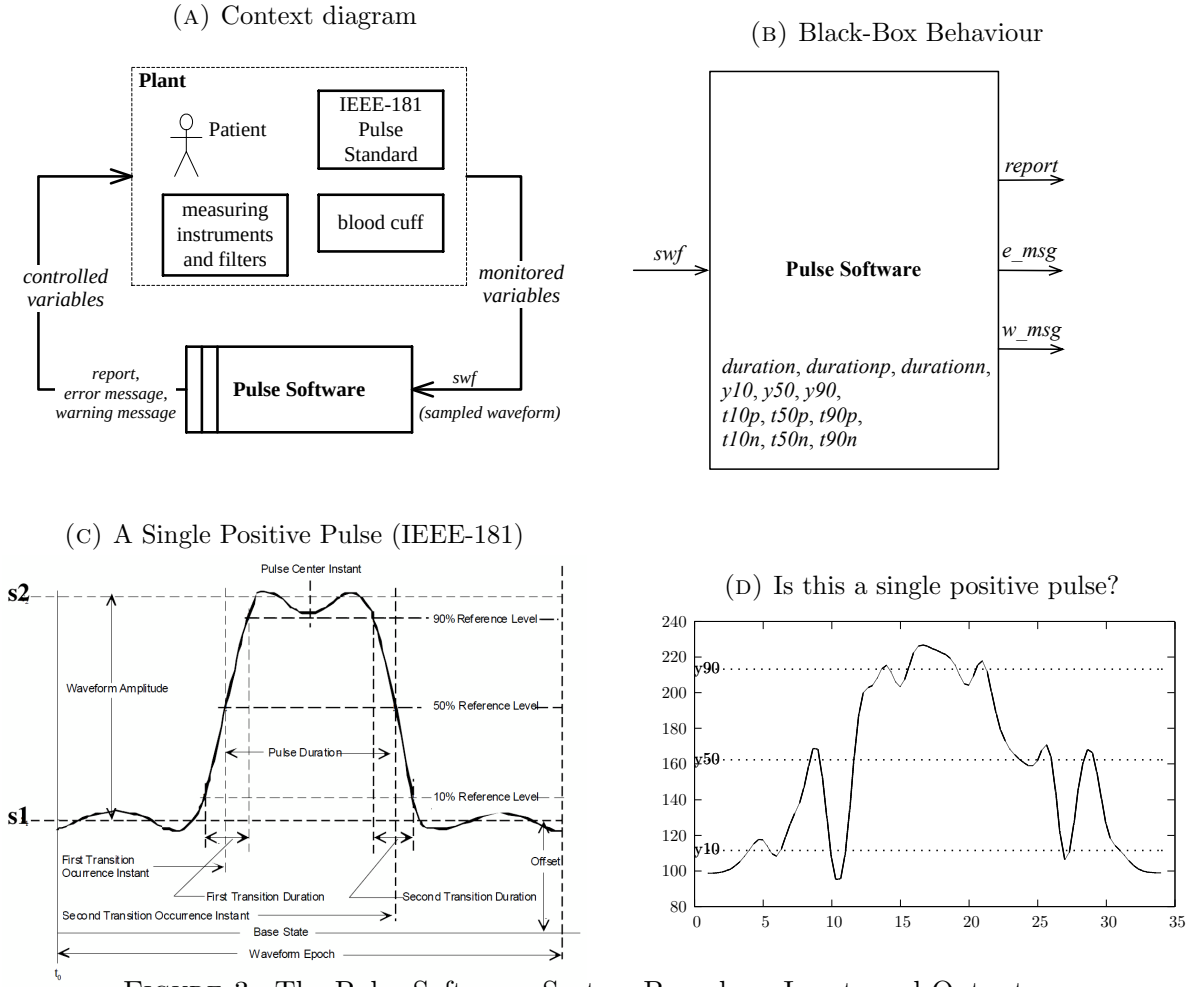


FIGURE 3. The Pulse Software: System Boundary, Inputs and Outputs

are well-defined; 2) an appropriate error message, if any; and 3) an appropriate warning message, if any. For output 1), the software attempts to calculate the various pulse and transition parameters (whose values are calculated by "in-box" queries shown in Fig. 3(b)) as defined by the *IEEE Standard 181 on Transitions, Pulses, and Related Waveforms.*

In the IEEE-181 standard, a single positive pulse (see Fig. 3(c)) is divided into a positive-going transition (one whose terminating level $s2$ is more positive than its originating level $s1$), and a negative going transition (one whose terminating state is more negative than its originating state). The standard specifies that we use linear interpolation to obtain (real-valued) times that are in-between the sampled time instants.

At the requirements level, the software calculates—according to IEEE-181 requirements—a variety of parameters (typed to the set $\mathbb{R}$ of real numbers). For each pulse we must calculate its duration, as well as the 10%, 50% and 90% levels. Moreover, for each (positive or negative) transition of the pulse, we must calculate its duration, as well as the 10%, 50% and 90% instants. Table 1 summarizes abbreviations that we adopt for these parameters.

Pulse	**Abb.**	**Pos. Tran.**	**Abb.**	**Neg. Tran.**	**Abb.**
pulse duration	*duration*	duration	*durationp*	transition duration	*durationn*
10% level	$y10$	10% instant	$t10p$	10% instant	$t10n$
50% level	$y50$	50% instant	$t50p$	50% instant	$t50n$
90% level	$y90$	90% instant	$t90p$	90% instant	$t90n$

TABLE 1. Abbreviations for Pulse and Transition Parameters

Our industrial partner was faced with various questions with their developed code. They wanted to know how to increase their confidence that their code was correct and at least satisfied IEEE-181. Pulses from ill patients (e.g. Fig. 3(d)) show significant variance from the classical shape (Fig. 3(c)). They found it difficult to write their code to deal with such variances and flag that the signal does not really represent a legal pulse (in some cases their code produced spurious results). They wanted to know how they could argue to certifying agencies, e.g. the FDA, that their code is safe and fit for use. Having precise documentation of the requirements is a prerequisite for answering these questions consistently.

Where there are multiple ten and 90 percent instants, IEEE-181 specifies that we take the 10%, and 90% instants that are closest to the 50% instant. However, for some pulses this would result in an ordering $t90 < t50p < t10p$ which gives a negative duration for the transition. The interpolation formula in IEEE-181 standard, besides being overly-complicated, does not include a description of its limitations: it includes a division by an expression that might be zero without specifying what to do in cases where it is. See the technical report [15] for more decision of the ambiguities and limitations in our customer's code and the standard.

Our proposed method helps address the above issues. For example, the limitations in the interpolation formula in the IEEE standard could have been specified in a query's precondition. our version of the interpolation formula is total (see our abstraction RFUN). E-descriptions (Section 3.1) differentiate between valid and invalid signals, thus helping to remove ambiguities. Given that the standard was not always clear, we made what we thought are relevant assumptions for the sake of the presentation. R-descriptions (Section 3.2) describe the required calculation of parameters for valid pulses and the errors or warnings for invalid pulses. The complete specification is less than two pages (Fig. 6 on p13 and Table 2 on p12).

3.1. **Atomic E-descriptions.** E-descriptions document environmental assumptions. An atomic description consists of two parts: (1) the description number (e.g. ENV1) providing traceability to the design, the code, and acceptance tests; and (2) an informal statement in natural language.

By introducing a number of E-descriptions, our proposed method may also be used by standards organizations to ensure that their standards are complete.

ENV1	A *valid pulse* consists of at least 3 samples, has a unique maximum and each transition has at least one 50% instant.

Also, the levels are the same across both the positive and negative transitions.

ENV2	The unique maximum partitions the waveform into a positive transition and a negative transition. The 10%, 50% and 90% levels are the same for both the positive and negative transitions.

Furthermore, as the linear interpolation formula for calculating in-between instants (e.g. the 50% instant for positive transition) provided by the IEEE-181 standard is undefined for integer values of t (representing the original samples) due to division by zero. We need a well-defined query that returns a level for an arbitrary instant t.

ENV3	The pulse waveform is sampled into a discrete number of measurements (swf) used to approximate the pulse. The approximation (wf) is built through linear interpolation of consecutive sample points.

3.2. **Atomic R-descriptions.** Having defined what a valid pulse is as E-descriptions (Section 3.1), we document the required system behaviour by considering three cases:

REQ4	**ok**: If the input pulse is *valid* and the 10% levels of both transitions exist then output all the parameters: **(a)** For the waveform: 10%, 50% and 90% levels. **(b)** For each transition: 10%, 50% and 90% instants. **(c)** For each transition: the transition duration (i.e. time from the 10% instant to the 90% instant). **(d)** The pulse duration (time from the 50% instant of the positive transition to the 50% instant of the negative transition).

REQ5	**Warning**: If the input pulse is *valid* and at least one of the 10% levels is missing, output all the parameters except for the missing 10% levels and instants (and associated transition duration) and issue a warning.

REQ6	**Error**: If the input pulse is *invalid* then no parameters are calculated and appropriate error messages are printed.

Furthermore, there may be multiple 50% instants, and the standard specifies that the first one must be selected, which is appropriate for the positive transition but not for the negative transition, in which case the last 50% seems more appropriate (if the two transitions were meant to be treated symmetrically). The 10% and 90% will then be defined accordingly.

REQ7	In the case where more than one 50% instants are present in the positive transition (respectively, negative transition), the first (respectively, the last) 50% instant is to be selected for output in conformity with REQ4.

REQ8	Output the 10% and 90% instants closest to the 50% instants such that REQ9 is satisfied.

Finally, we document an important property about transitions:

REQ9	- For positive transition: $t10p < t50p < t90p$ - For negative transition: $t90n < t50n < t10n$

In the next two sections we present a complete specification for the pulse software, consisting of abstract data types (Section 3.3.1), modules (Section 3.3.2), and tabular expressions that reference the declared module variables and queries (Section 3.4). Furthermore, we will revisit (Section 3.3.3) the above E- and R-descriptions by examining how they are reflected in the modular specification.

3.3. **Modular Specification.** We propose the modular structure in Fig. 4, where the two rounded boxes denote abstract data *types* (i.e. $SEQ[G]$ and $RFUN$), and the other square boxes denote *modules*. Each arrow in Fig. 4 denotes the direction of dependency and corresponds to a **use** clause in Fig. 6 (on p13). A module has access to variables and queries of all modules it uses.

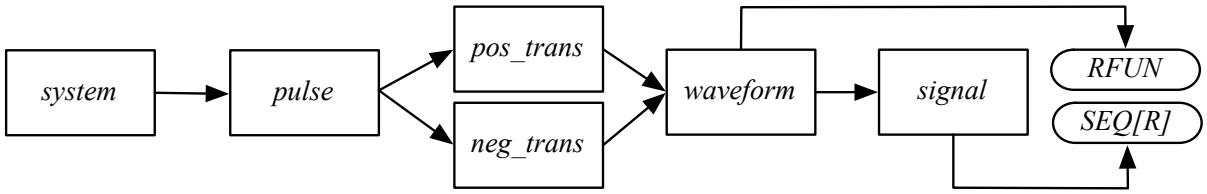


FIGURE 4. Modular Specification of the Pulse Software

3.3.1. *Abstract Data Types.* We often need to query the input sample sequence *swf* about the level of a real-valued instant x that satisfies $1 \leq x \leq swf.count$ but does not actually exist in the discrete domain of *swf*. The IEEE-181 standard recommends the use of linear interpolation; however, the formulas supplied by the standard may suffer from the division-by-zero error, and the standard does not give a unified formula applicable to all input instants.

This leads us to the introduction of a new data type *RFUN* (Fig. 5(b)) that enables us to abstract the input sample sequence from its discrete, finite domain. Consequently, we have a single point of querying about the level of any given real-valued instant. In Fig. 5(b), by writing $RFUN \triangleq (\bigcup x, y : \mathbb{R} \mid x \leq y \bullet [x, y] \rightarrow \mathbb{R})$, we introduce a new data type *RFUN* that is synonymous with the set of total functions, each of which has its domain as a contiguous, real-valued interval and has its range as a set of real numbers.

The new type *RFUN* supports a query *seq2rfun* that converts from a finite sequence of real numbers (e.g. $swf : SEQ[\mathbb{R}]$) to a continuous function. A real-valued instant in the domain of $seq2rfun(swf)$ is projected to a value that is calculated using an improved version of linear interpolation[1] that is free from the division-by-zero error. We observe that both *swf* and $seq2rfun(swf)$ agree on their projected levels from the integer domain of *swf*, i.e. $swf = \mathbb{N} \lhd seq2rfun(swf)$. Queries *first* and *last* are defined for us to select instants, within a given range, that are projected to the same given level.

3.3.2. *Modules.* The specification of modules is included in Fig. 6 (on page13). The *system* module is the top-level unit of the pulse software, where we declare three variables to correspond to the expected outputs: 1) $report : \mathbb{S} \nrightarrow \mathbb{R}$ that maps names of pulse or transition parameters to their values, if they exist; 2) *e_msg* that stores an error message, if any; and 3) *w_msg* that stores a warning message, if any. The *system* module has access to all queries and variables that are declared in its decedent modules. We specify an event *execute* whose occurrence assigns an arbitrary input pulse (i.e. **any** $p \in SEQ[\mathbb{R}]$) to the state variable *swf* (as declared in the module *signal*), and it updates variables *report*, *e_msg*, and *w_msg* according to a separate tabular expression (Table 2 on page 12) that references the accessible queries.

We distribute queries that are responsible for calculating the pulse and transition parameters into modules that *system* uses, e.g. *duration* in the *pulse* module, $t10p$ in the *pos_trans* module,

[1]See the post-condition of query *seq2rfun*. Given a real t and a natural number n, $\lfloor t + n \rfloor = \lfloor t \rfloor + n$. Thus $\lfloor t+1 \rfloor = \lfloor t \rfloor + 1$. In the definition of $seq2rfun(s)(t)$ the coefficients always add up to one, i.e. $(\lfloor t+1 \rfloor - t) + (t - \lfloor t \rfloor) = 1$. This eliminates the possibility of division by zero and avoids the case analysis in the IEEE-181 standard.

(A) Generic Sequence

type SEQ[G] $\triangleq$ $(\bigcup n : \mathbb{N} \bullet 1..n \to G)$
queries
$count(s\colon SEQ[G])\colon \mathbb{N} \triangleq \#\,\mathrm{dom}(s)$
$\epsilon\colon SEQ[G] \triangleq \varnothing$
infix $\triangleright\,(s\colon SEQ[G]; g\colon G)\colon SEQ[G]$
`--append operator`
$\triangleq s \lhd\!\!\lhd \{s.count{+}1 \mapsto g\}$
$has(s\colon SEQ[G]; g\colon G)\colon \mathbb{B} \triangleq g \in \mathrm{ran}(s)$
$head(s\colon SEQ[G])\colon G \triangleq s(1)$
require $s \neq \epsilon$
$last(s\colon SEQ[G])\colon G \triangleq s(s.count)$
require $s \neq \epsilon$
$tail(s\colon SEQ[G])\colon SEQ[G]$
require $s \neq \epsilon$
ensure
$(Result.count = s.count - 1)$
$\wedge\ (\forall i\colon s.domain \mid i > 1 \bullet$
$s(i-1) = Result(i))$
theorems
$\forall s\colon SEQ[G]; g\colon G \bullet s \triangleright g \neq \epsilon$
$\forall s\colon SEQ[G]; g\colon G \bullet last(s \triangleright g) = g$

(B) Real-Valued Function

type RFUN $\triangleq$ $(\bigcup x, y : \mathbb{R} \mid x \leq y \bullet [x, y] \to \mathbb{R})$
queries
$ubound\ (rf\colon RFUN)\colon \mathbb{R}$ `--upper bound of the domain`
$lbound\ (rf\colon RFUN)\colon \mathbb{R}$ `--lower bound of the domain`
axiom $(\forall rf : RFUN \bullet \mathrm{dom}(rf) = [rf.lbound, rf.ubound])$
queries
$has(rf\colon RFUN;\ x_1,\ x_2,\ y\colon \mathbb{R})\colon \mathbb{B}$
$\triangleq (\exists x\colon \mathrm{dom}(rf) \mid x_1 \leq x \leq x_2 \bullet rf(x) = y)$
`--does y occur between x1 and x2?`
$first\ (rf\colon RFUN;\ x_1,\ x_2,\ y\colon \mathbb{R})\colon \mathbb{R}$
$\triangleq (\downarrow x\colon \mathbb{R} \mid x_1 \leq x \leq x_2 \wedge rf(x) = y \bullet x)$
`--1st instant of level y in interval [x1,x2]`
require $rf.has(x_1, x_2, y)$
$last\ (rf\colon RFUN;\ x_1,\ x_2,\ y\colon \mathbb{R})\colon \mathbb{R}$
$\triangleq (\uparrow x\colon \mathbb{R} \mid x_1 \leq x \leq x_2 \wedge rf(x) = y \bullet x)$
`--last instant of level y in interval [x1,x2]`
require $rf.has(x_1, x_2, y)$
$seq2rfun(s\colon SEQ[\mathbb{R}])\colon RFUN$ `--linear interpolation`
ensure $dom(Result) = [1, s.count]$
$\wedge\ (\forall\ x : [1, s.count] \bullet$
$Result(x) = \quad s(\lfloor x \rfloor){*}(\lfloor x+1 \rfloor - x)$
$+\ s(\lceil x \rceil){*}(x - \lfloor x \rfloor))$

FIGURE 5. Abstract data types for the pulse system

etc. In those modules we also declare queries that calculate the intermediate results, e.g. *ymax* in the *signal* module. The result of each query is precisely defined either by an equality expression ($\triangleq \ldots$) or by a post-condition (an **ensure** clause). A query may only be used in a context where its precondition (the **require** clause) holds, for otherwise its result is not *well-defined*. For example, queries *ymax* and $t50p$ are only well-defined when $s3$ and $t50p?$ hold, respectively. We also use a **require** clause at the module level to specify constraint that is to be included as part of the preconditions of all queries. For example, in the *waveform* module, each query should include the constraint $s3 \wedge um$ as part of its precondition.

The *signal* module is the basic unit of the pulse software where we declare the input sampled pulse as a variable $swf : SEQ[\mathbb{R}]$, by instantiating the generic sequence to be a real-valued sequence. The module that uses *signal*, i.e. module *waveform*, gains the access to variable *swf*. Furthermore, according to ENV 3, module *waveform* abstracts sequence *swf* from its discrete, integer-valued domain by defining a query $wf : RFUN$. More precisely, the definition of query *wf* uses the abstraction function *seq2rfun*. Consequently, details of performing linear interpolation on sequence *swf* are encapsulated in one single query, i.e. *wf* in *waveform*. Moreover, module *waveform* and all its parent modules are able to calculate the pulse and transition parameters by using the higher-level, abstract function $wf : RFUN$ rather than $swf : SEQ[\mathbb{R}]$. For example, queries $t50p$ in module *waveform* and $t10n$ in module *neg_trans* are defined in terms of query *wf*.

3.3.3. *Revisiting E- and R-descriptions.* We discuss how the informal E-descriptions (Section 3.1) and R-descriptions (Section 3.2) are formalized as module queries and invariants, and entries in the tabular specification (Section 3.4). Texts of this discussion may be integrated into a third, cross-reference compartment for each ENV or REQ box in Sections 3.1 to 3.2.

For ENV1, in the *pulse* module, we define a Boolean query *ok* whose definition corresponds to what qualify as a valid input pulse. Furthermore, the last two invariants in module *pulse* specify that if the input pulse is valid, then all queries that calculate the pulse and transition parameters are well-defined and the various instants appear in the right orders. For ENV2, in the *waveform*

This tables is used in the context of module *system* in Fig.6 on p13 .

conditions on input					$e\_msg : 0..3$	$w\_msg : 0..2$	$report : \mathbb{S} \nrightarrow \mathbb{R}$
$s3$	um	$t50?$	$t10?$		0	0	$format \lhd S_1 \lhd S_2$
			$\neg t10?$	$\neg t10p?$	0	1	$format \lhd S_2$
				$\neg t10n?$	0	2	$format \lhd S_1$
		$\neg t50?$			1	0	$\varnothing$
	$\neg um$				2	0	
$\neg s3$					3	0	

where $S_1 = \{\text{“}t10p'' \mapsto t10p, \text{“}durationp'' \mapsto durationp\}$; similarly for S_2 on neg. trans.

TABLE 2. Requirements (see Table 4 for conditions and Table 5 for messages)

p	“duration”	“y10”	“y50”	“y90”	“t50p”	“t90p”	“t50n”	“t90n”
$format(p)$	$duration$	$y10$	$y50$	$y90$	$t50p$	$t90p$	$t50n$	$t90n$

TABLE 3. Formatting Pulse & Transition Parameters (“t10p” and “t10n” left out)

Ab.	**Meaning**
$s3$	are there at least 3 samples?
um	is there a unique maximum?
$t50?$	do both $t50$% instants exist?
$t10?$	do both t10% instants exist?
$t10p?$	does $t10p$ (positive transition instant for level $y10$) exist?
$t10n?$	does $t10n$ (negative transition instant for level $y10$) exist?

TABLE 4. Conditions

#	Error
0	no error
1	no 50% instant
2	no unique maximum level
3	input lacks 3 finite floats

#	Warning
0	no warning
1	No $t10p$ instant, $durationp$
2	No $t10n$ instant, $durationn$

TABLE 5. Errors/Warnings

module, we define real-valued queries $y10$, $y50$, and $y90$ whose definitions are accessible by modules *pos_trans* and *neg_trans*. For ENV3, we introduce the query $wf : RFUN$ that abstracts the input pulse $swf : SEQ[\mathbb{R}]$ using linear interpolation.

For REQ4, we declare queries $t50p$ and $t50n$ in module *waveform* and all other parameters as queries in modules *pos_trans*, *neg_trans*, and *pulse*. For REQ5, in the *pulse* module, we define a Boolean query *warning* whose definition corresponds to the case where there is at least one missing 10% levels. Moreover, in modules *pos_trans* and *neg_trans*, the preconditions of queries $t10p$, $durationp$, $t10n$, and $durationn$ specify that the 10% levels must exist for their values to be well-defined. For REQ6, see Section 3.4 for how messages are organized according to the error conditions. For REQ7, in the *waveform* module, query $t50p$ (and $t50n$) is defined to return the first (and the last) 50% instant. For REQ8, in the *pos_trans* module, query $t10p$ calculates the last, and hence the closest, instant with 10% level before the 50% instant $t50p$. Similarly, query $t90p$ calculates the 90% instant that is closest to $t50p$ by selecting the first one. Symmetric calculations apply to queries $t10n$ and $t90n$ in the *neg_trans* module. Finally, for REQ9, we declare invariants in both *pos_trans* and *neg_trans* modules about the 10%, 50%, and 90% instants.

3.4. **Using Module Queries in Function Tables.** Model-based contracts (pre/post-conditions) specified for module queries in Fig. 6 (on p13) facilitate the the input-output behaviour description of the pulse software as a tabular expression (Table 2). The contracts (organized by modules) and the tabular expression together constitute the software specification, used to validate the requirements via checks for its completeness, disjointness and well-definedness, and proofs of properties.

Since Table 2 is used in the context of the *system* module (Fig. 6 on p13), it has access to all queries that are declared in modules that it uses. The rows of the table can be divided into three disjoint groups, corresponding to REQ4 to REQ6. The rows below the grey area correspond to

module *system* **use** *pulse*
variables *report*, *e_msg*, *w_msg*: $\mathbb{S}$
event
execute
any $p \in SEQ[\mathbb{R}]$
then swf := p || « Table 2 on page 12 »
end

module pulse **use** pos_trans, neg_trans
queries
$duration: \mathbb{R} \triangleq t50n - t50p$
`--time between 50% instants of positive and negative transitions`
require $t50?$
`--intermediate queries`
$t50?: \mathbb{B} \triangleq t50p? \wedge t50n?$ `--do both 50% instants exist?`
$t10?: \mathbb{B} \triangleq t10p? \wedge t10n?$ `--do both 10% instants exist?`
require $t50p? \wedge t50n?$
$error: \mathbb{B} \triangleq \neg(s3 \wedge um \wedge t50?)$
$warning: \mathbb{B} \triangleq \neg error \wedge \neg t10?$
$ok: \mathbb{B} \triangleq s3 \wedge um \wedge t50? \wedge t10?$
invariant
$complete(\langle error, warning, ok\rangle) \wedge disjoint(\langle error, warning, ok\rangle)$
$ok \Rightarrow t10p? \wedge t10n? \wedge t50p? \wedge t50n? \wedge t90p? \wedge t90n? \wedge durationp? \wedge durationn?$
$ok \Rightarrow (t10p < t50p < t90p) \wedge (t90n < t50n < t10p)$

module pos_trans **use** waveform
require $t50p?$
queries
$t10p: \mathbb{R} \triangleq wf.last(1, t50p, y10)$
require $t10p?$
$t90p: \mathbb{R} \equiv wf.first(t50p, tmax, y90)$
require $t90p?$
$durationp: \mathbb{R} \triangleq t90p - t10p$
require $durationp?$
`--intermediate queries`
$t10p?: \mathbb{B} \triangleq wf.has(1, t50p, y10)$
$t90p?: \mathbb{B} \triangleq wf.has(t50p, tmax, y90)$
$durationp?: \mathbb{B} \triangleq t10p? \wedge t90p?$
invariant $durationp? \Rightarrow t10p < t50p < t90p$

module neg_trans **use** waveform
require $t50n?$
queries
$t10n: \mathbb{R} \triangleq wf.first(t50n, n, y10)$
require $t10n?$
$t90n: \mathbb{R} \equiv wf.last(tmax, t50n, y90)$
require $t90n?$
$durationn: \mathbb{R} \triangleq (t90n - t10n)$
require $durationn?$
`--intermediate queries`
$t10n?: \mathbb{B} \triangleq wf.has(t50n, n, y10)$
$t90n?: \mathbb{B} \triangleq wf.has(tmax, t50n, y90)$
$durationn?: \mathbb{B} \triangleq t10n? \wedge t90n?$
invariant $durationn? \Rightarrow t90n < t50n < t10n$

module signal **use** $SEQ[\mathbb{R}]$
variable $swf: SEQ[\mathbb{R}]$
queries
$n: \mathbb{N} \triangleq swf.count$
$s3: \mathbb{B} \triangleq (n \geq 3)$ `--at least 3 samples?`
$ymax: \mathbb{R} \triangleq (\uparrow i | 1 \leq i \leq n \bullet swf(i))$
`--maximum level (s2 in IEEE-181)`
require $s3$
$ymin: \mathbb{R} \triangleq (\downarrow i | 1 \leq i \leq n \bullet swf(i))$
`--minimum level (s1 in IEEE-181)`
require $s3$
$um: \mathbb{B} \triangleq (\#i | 1 \leq i \leq n \bullet swf(i) = ymax) = 1$
`--is there a unique maximum?`
require $s3$

module waveform **use** signal, RFUN
require $s3 \wedge um$
queries
$y10: \mathbb{R} \triangleq ymin + 0.1 * amplitude$
$y50: \mathbb{R} \triangleq ymin + 0.5 * amplitude$
$y90: \mathbb{R} \triangleq ymin + 0.9 * amplitude$
$t50p: \mathbb{R} \triangleq wf.first(1, tmax, y50)$
require $t50p?$
$t50n: \mathbb{R} \triangleq wf.last(tmax, n, y50)$
require $t50n?$
`--intermediate queries`
$wf: RFUN \triangleq seq2rfun(swf)$
$amplitude: \mathbb{R} \triangleq ymax - ymin$
$t50p?: \mathbb{B} \triangleq wf.has(1, tmax, y50)$
$t50n?: \mathbb{B} \triangleq wf.has(tmax, n, y50)$
$tmax: \mathbb{R}$ `--instant for ymax`
ensure $1 \leq Result \leq n \wedge wf(Result) = ymax$

FIGURE 6. Modular Specification: Variables, Queries, Assumptions, Invariants

query *error* (invalid input). The first row of the table corresponds to input signals that satisfy query *ok* (all parameters can be calculated). The grey rows correspond to query *warning* (most parameters can be calculated, with a warning for those that cannot be calculated, i.e. $t10p$ or $t10n$). We define a function *format* in Table 3 whose domain contains parameters that can always be calculated for non-erroneous input, excluding $t10p$, *durationp*, $t10n$, and *durationn*. The report is formatted via an override, e.g. $format \lhd \{\text{“}t10p'' \mapsto t10p, \text{“}durationp'' \mapsto durationp\}$.

Completeness and disjointness of the pulse specification appear as invariants to be proved in the *pulse* module (Fig. 6 on p13). The layout of of our tables makes it easy to prove that the referenced queries and variables are well-defined, as the condition rows correspond to their preconditions. For example, in the first row of Table 2, all parameters are calculated in the context where there are at least three samples, there is a unique maximum, and the 50% and 10% instants exist. To complete our validation of requirements, in the next section we establish that the tabular expressions entail global properties that are captured as R-descriptions (e.g. REQ9 on p10).

3.5. **Validating Tabular Expressions via Proofs.** The process of decomposing queries into modules, a critical step of our approach, revealed the need to introduce REQ9 (on p10) asserting that in the case where two 10% (or 90%) instants are equally close to $t50$, e.g. where the first instant occurs before $t50$ and the second occurs after $t50$, the appropriate instant should be chosen on the basis that the 10%, 50% and 90% instants must be in different orders for positive and negative transitions. The atomic requirement REQ9 is declared as a property proof obligation $ok \Rightarrow (t10p < t50p < t90p) \wedge (t90n < t50n < t10n)$ in the *pulse* module (Fig. 6 on p13).

Tabular expressions (e.g. Table 2) and atomic requirements (e.g. REQ9) play different roles. The tabular expression ensures that the input-output black-box relation is completely specified. However, it is not obvious from the tabular expression that REQ9 holds as a global safety property. The modular specification in Fig. 6 is used to prove that REQ9 holds as a logical consequence of Table 2. This demonstrates the consistency between the modular specification and the atomic description REQ9, and is thus an important component of requirements validation. This proof follows from the invariants declared in the modules *pos_trans* and *neg_trans*. For example, in the positive transition module we have the invariant $durationp? \Rightarrow t10p < t50p < t90p$ (likewise for the negative transition). Part of the proof of the above invariant declared for the *pos_trans* module is provided in Fig. 7.

Prove: $t50p? \wedge durationp? \Rightarrow (t10p < t50p)$

$t10p < t50p$

$=$ ⟨ def. of $t10p$ in module *positive* in Fig. 6 (on p13) and $t50p? \wedge durationp? \Rightarrow t10p?$ ⟩

$wf.last(1, t50p, y10) < t50p$

$=$ ⟨ def. of RFUN.*last* ⟩

$(\uparrow t : \mathbb{R} \mid 1 \leq t \leq t50p \wedge wf(t) = y10 \bullet t) < t50p$

$=$ ⟨ $<$ over $\uparrow$; trading ⟩

$(\forall t : \mathbb{R} \mid 1 \leq t \wedge t \leq t50p \wedge t50p \leq t \bullet wf(t) \neq y10)$

$\Leftarrow$ ⟨ drop first conjunct in range; anti-symmetry of $\leq$; one point rule ⟩

$wf(t50p) \neq y10$

$=$ ⟨ $wf(t50p) = y50$; def. of $t50p$ in waveform ⟩

$\neg(y50 = y10)$

$=$ ⟨ def. of $y50$ and $y10$ in waveform ⟩

$\neg((ymin + 0.5 * amplitude) = (ymin + 0.1 * amplitude))$

$=$ ⟨ arithmetic and $amplitide \neq 0$ ⟩

true

FIGURE 7. Proving a property of module *positive* that also validates REQ9

3.6. **Using a SMT solver to discharge proof obligations.** SMT solvers such as Z3 [**?**] allow us to check the satisfiability of first-order predicates involving real numbers. When proving the predicate $P(x) \Rightarrow Q(x)$ as a theorem, we check that there are no witnesses that satisfy the negation of the predicate, i.e. there are no assignments to x that make $P(x) \wedge \neg Q(x)$ true. Z3 will answer `unsat` if the negation of the predicate has no witnesses, meaning that $P \Rightarrow Q$ is a theorem; `sat` if a counterexample is found; or `unknown` if no conclusions can be reached.

Using the Z3 SMT solver, we mechanize the invariant proof in Fig. 7 by checking the validity of each step. We represent steps in the proof structure like Fig. 7 as $S_0, S_1, \ldots, S_n$. Each step formula S_i is formed by $F_i \; R_i \; F_{i+1}$, where $i \geq 0$, F_i and F_{i+1} are predicates, and R_i is either an implication or an equivalence. We check that all steps are valid and they hold together to entail $H \vdash P$. For example, in Fig. 7, S_0 is $(t10p < t50p) \equiv (wf.last(1, t50p, y10) < t50p)$, and the theorem we aim to prove is $(t50p? \wedge durationp?) \vdash (t10p < t50p)$. The following proof tree structure is encoded in Z3:

$$\dfrac{\dfrac{S_0 \wedge S_1 \wedge \ldots \wedge S_{n-1} \vdash P}{H, S_0 \wedge S_1 \wedge \ldots \wedge S_{n-1} \vdash P}\,\text{MON} \qquad \dfrac{H \vdash S_0 \quad H \vdash S_1 \quad \ldots \quad H \vdash S_{n-1}}{H \vdash S_0 \wedge S_1 \wedge \ldots \wedge S_{n-1}}\,\text{SPLIT}}{H \vdash P}\,\text{CUT}$$

We use three deduction rules: CUT introduces and proves a new assumption, MON(otonicity) drops some hypotheses, and SPLIT divides the proof of a conjunction into the proofs of its constituents. The bottom sequent in the proof tree is the target theorem. The leaves are sequents stating that the steps establish the goal, and that the steps with their justifications are valid. We can ensure that the goal and the steps are well-defined by checking the sufficient condition: $\mathcal{D}(P) \wedge D(F_0) \wedge D(F_1) \wedge \ldots \wedge D(F_n)$. See Appendix C for Z3 script for proof in Fig. 7.

4. Conclusion and related work

Embedding function tables in an event system, and using queries organized in modules, allows our framework to describe system behaviour at the requirements levels in a way that supports precise documentation and validation of requirements, Validation is performed by (a) proving invariants (representing system safety properties) and (b) by the use of function tables to check that the computer controller specification is complete, disjoint and well-defined. As we mentioned in the previous section, the complete requirements for the biomedical device in the case study is less than two pages (Fig. 6 on p13 and Table 2 on p12). The rest of the previous section was spent explaining the application of the method and documenting the E/R-descriptions.

The novelty of our method also lies in the synthesis of well-established software engineering principles: the separation between the controller and its operating environment using context diagrams [8], the identification of monitored and controlled variables [7, 16, 5], and the use of tabular expressions to capture black-box, input-output relations [20, 10].

The theorem prover PVS has been used to provide tool support for tabular expressions [10, 23, 3]. In PVS, partial functions are converted into total functions using predicate subtyping which generates type checking proof obligations. Our calculus of well-definedness, on the other hand, extends Abrial's work [2] on model queries to the specification context of tabular expressions. While substantial progress has been made in mechanizing such proofs, there are still many challenges [6].

References

[1] Jean-Raymond Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[2] Jean-Raymond Abrial and Louis Mussat. On using conditional definitions in formal theories. In *ZB*, LNCS 2272. Springer-Verlag, 2002.

[3] C. Eles and M. Lawford. A tabular expression toolbox for matlab/simulink. In *NASA Formal Methods*, volume LNCS 6617, pages 494–499, 2011.

[4] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer Verlag, 1993.

[5] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, 2000.

[6] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, 2012.

[7] Michael Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. Addison-Wesley, New York, NY, USA, 1995.

[8] Michael Jackson. The operational principle and problem frames. In Cliff B Jones, A.W. Roscoe, and Kenneth R Wood, editors, *Reflections on the Work of C. A. R. Hoare*. Springer Verlag, 2010.

[9] Ying Jin and David Lorge Parnas. Defining the meaning of tabular mathematical expressions. *Science of Computer Programming*, 75(11):980–1000, November 2010.

[10] M. Lawford, P. Froebel, and G. Moum. Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. *Formal Methods in System Design*, 2004.

[11] Robyn R. Lutz. Analyzing Software Requirements Errors in Safety Critical Embedded Systems. pages 126–133, 1993.

[12] T. S. E. Maibaum and Alan Wassyng. A product-focused approach to software certification. *IEEE Computer*, 41(2):91–93, 2008.

[13] Farhad Dinshaw Mehta. *Proofs for the Working Engineer*. PhD thesis, ETH, Zurich, 2008.

[14] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[15] J. S. Ostroff, C.-W. Wang, and S. Hudon. Precise documentation of requirements and executable specifications. Technical report, Computer Science and Engineering, York University, CSE-2012-03, 2012.

[16] Jonathan S. Ostroff and Richard F. Paige. The Logic of Software Design. *Proc. IEE - Software*, 147(3):72–80, 2000. The Logic of Software Design.

[17] Jonathan S. Ostroff, Chen-Wei Wang, and Simon Hudon. Precise documentation and validation of requirements. Technical Report CSE-2013-08, EECS, York University, 2013.

[18] D. L. Parnas, G. J. K. Asmis, and J. Madey. Assessment of Safety-Critical Software in Nuclear Power Plants. *Nuclear Safety*, 32(2):189–198, 1991.

[19] David Parnas. Predicate logic for software engineering. *IEEE Trans. Softw. Eng.*, 19(9), 1993.

[20] David L. Parnas and Jan Madey. Functional Documentation for Computer Systems. *Science of Computer Programming*, 25:41–61, 1995.

[21] George Tourlakis. On the soundness and completeness of equational predicate logics. *J. Log. Comput.*, 11(4):623–653, 2001.

[22] Alan Wassyng and Mark Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. In *FME*, volume 2805 of *LNCS*, pages 133–153, 2003.

[23] Alan Wassyng and Mark Lawford. Software tools for safety-critical software development. *STTT*, 8(4-5):337–354, 2006.

[24] Alan Wassyng, Mark Lawford, and Xiayong Hu. Timing tolerances in safety-critical software. In *FM*, pages 157–172, 2005.

Appendix A. Well-definedness of Expressions with Partial Functions

Adding a new query q to our specification effectively introduces a new axiom into the theory and rules for how to calculate with expressions in q, including the case in which q is a partial function or relation. These partial relations are also used in tabular expressions. We need to ensure that wherever they are used in tables they are well-defined.

In classical tabular expressions [9, 19], all partial functions are transformed into total functions by extending the range of functions with a special undefined value. However, the logic used is still a two-valued predicate logic. This is achieved by defining any expression involving an undefined term to evaluate to false in an assignment. Predicates are identified with their satisfying assignments (so that $1 \div x = 1 \div x$ effectively reduces to $x \neq 0$). Advantages of the approach are that the logic is kept simple, the assigned meanings are consistent with intuitive interpretations, and the expressions are simpler in certain cases while preserving two valued logic. However, complements will not always work (e.g. $\sqrt{x} > \sqrt{y}$ and $\sqrt{x} \leq \sqrt{y}$ both evaluate to false) and complexity reappears in the axiomatic definitions of the functions (requiring the introduction of an undefined value). Also, conventional simplification rules, and hence some automatic simplifiers and verifiers would need to be modified or used with caution as they are often based on the implicit assumption that functions are total. Even worse, it allows the expression of nonsensical properties in specifications without flagging any problem.

Model contracts presume that functions and relations will be partial. We thus seek a logic where we can introduce and reason with partial functions without the need to constantly convert them into total functions. In the logic that we adopt in the sequel, the predicate $1 \div x = 1 \div x$ does not pass a well-definedness check (done using proof obligations in a standard theorem prover). However, $(x \neq 0) \wedge (1 \div x = 1 \div x)$ is well-defined and it can then be submitted to the theorem prover as if all functions were total (the prover will fail to prove it as a theorem). We thus are able to introduce partial functions (without converting them into total functions) while using standard tools and mathematical conventions. Consider the following query and its pre/post-conditions:

$q(x : T_x) : T_r$
`--introduce new query q into the theory`
require $C_q(x)$
ensure $R_q(x, Result)$

Let $\mathcal{A}$ be the set of axioms (and derived theorems) of our theory already in place before the introduction of query q. For our logic we use notations similar to that of [4]. The query can be safely introduced into our theory provided the special local variable *Result* (denoting values returned by the query) does not occur free in C_q, the free variables of R_q are limited to x and *Result*, the precondition C_q and postcondition R_q refer only to previously defined symbols, and the query is feasible:

$$x \in T_x \wedge C_q(x) \Rightarrow \exists r \in T_r \bullet R_q(x, r)$$

This entails that T_r is not empty. Under these conditions we can add the following axiom to $\mathcal{A}$, where r is a fresh variable:

Query Axiom: $x \in T_x \wedge r \in T_r \wedge C_q(x) \wedge (r = q(x)) \Rightarrow R_q(x, r)$
provided: $x \in T_x \wedge C_q(x) \Rightarrow \exists r \in T_r \bullet R_q(x, r)$

In the sequel we omit typing constraints assuming that variables and expressions are of the correct type. This is because correct typing is decidable and can be dealt with prior to well-definedness and validity [2].

As an example, consider the case of the square root defined below.

$\sqrt{\cdot}(x : \mathbb{R}) : \mathbb{R}$
--square root function
require $0 \leq x$
ensure $Result^2 = x$

Obviously $\sqrt{-1}$ is undefined and so are expressions such as $\sqrt{-1} = x \ \vee \ \neg\sqrt{-1} = x$. The question is how to deal with such undefined expressions. Also, suppose $\phi_1 \triangleq \sqrt{x} = y$ and $\phi_2 \triangleq (0 \leq x) \Rightarrow \sqrt{x} = y$. How would we write proofs of sequents such as $\mathcal{A} \vdash \phi_1$ and $\mathcal{A} \vdash \phi_2$? We should be able to prove the latter but not the former since it is undefined when $x < 0$.

Following the logic developed for Event-B [2], we inductively define the WD (well-definedness) operator $\mathcal{D}$ that maps formulas to their WD predicates. For a variable x we have that $\mathcal{D}(x) \triangleq true$. The well-definedness of a query application $q(x)$ is defined as $\mathcal{D}(q(x)) \triangleq \mathcal{D}(x) \wedge C_q(x)$. This works on the assumption that the feasibility of q has already been demonstrated, i.e. that $C_q(x)$ is a legitimate precondition (see "provided" clause in the Query Axiom). Given any formula α we have that $\mathcal{D}(\mathcal{D}(\alpha)) \equiv true$, i.e. WD-predicates are themselves well-defined [2]. We introduce additional rules for the counting quantifier and maximums and minimums. We only list a few of the rules, the others are available in [2].

(1) $$\mathcal{D}(x) \triangleq true$$

(2) $$\mathcal{D}(q(x)) \triangleq \mathcal{D}(x) \wedge C_q(x)$$

(3) $$\mathcal{D}(P \Rightarrow Q) \triangleq (\mathcal{D}(P) \wedge (P \Rightarrow \mathcal{D}(Q))) \vee (\mathcal{D}(Q) \wedge (\neg Q \Rightarrow \mathcal{D}(P)))$$

(4) $$\mathcal{D}(\forall x \bullet P) \triangleq (\forall x \bullet \mathcal{D}(P)) \ \vee \ (\exists x \bullet \mathcal{D}(P) \wedge \neg P)$$

(5) $$\mathcal{D}((\#i|\ p \leq i < q \ \wedge \ R \bullet P)) \triangleq \forall i \bullet \mathcal{D}(R) \wedge (R \Rightarrow \mathcal{D}(P))$$

(6) $$\mathcal{D}((\uparrow i | R \bullet exp)) \triangleq [\forall i \bullet \mathcal{D}(R) \wedge (R \Rightarrow \mathcal{D}(exp))] \wedge [\exists i \bullet R]$$

Applying the rules to ϕ_1 we obtain $\mathcal{D}(\phi_1) \equiv 0 \leq x$, hence $\mathcal{D}(\phi_1)$ on it's own is not a theorem and predicate ϕ_1 does not pass the $\mathcal{D}$-filter. It follows that we need $0 \leq x$ either as an hypothesis or as it appears in ϕ_2 (where $\phi_2 \triangleq 0 \leq x \Rightarrow \phi_1$). If we redo the above proof but this time for $\mathcal{D}(\phi_2)$ we see that $\mathcal{D}(\phi_2)$ reduces to true and hence is a theorem. The query introduction axiom, QIA, formalizes the definition of new queries:

Axiom **QIA** for query q: $\quad r = q(x) \ \Rightarrow \ R_q(x, r)$
provided r is fresh and q is feasible, i.e. $C_q(x) \ \Rightarrow \ \exists r \bullet R_q(x, r)$ and feasibility is well-defined: $\mathcal{D}(C_q(x) \ \Rightarrow \ \exists r \bullet R_q(x, r))$ (If q is in closed form we can use CFF, see below)

Whenever we are asked to prove a sequent $\mathcal{A} \vdash \beta_q$, where β_q involves a query q, we show below the need to discharge two proof obligations **WD** and **Validity**:

WD: $\quad \mathcal{A}, \mathbf{QIA} \vdash_{\mathcal{D}} \mathcal{D}(\beta)$ $\qquad$ **Validity:** $\quad \mathcal{A}, \mathbf{QIA} \vdash_{\mathcal{D}} \beta$

($\text{eqv}_{\mathcal{D}}$) where $\qquad H \vdash_{\mathcal{D}} P \ \triangleq \ \mathcal{D}(H), \mathcal{D}(P), H \vdash P$

We have thereby separated the proof of $\mathcal{A} \vdash \beta$ into two separate ones: **WD** and **Validity**. In the validity proof, we drop the precondition $C_q(x)$ in the antecedent of QIA as the formula is guaranteed to be well-defined. For example, QIA specialized for the predecessor function p yields: $r = p(x) \ \Rightarrow \ s(r) = x$.

We can then reformulate the predicate logic rules, to check the well-definedness of any newly-introduced expressions in a proof either through $\exists$-introduction (in the goal), $\forall$-introduction (in the hypothesis) or the cut rule [13, p46]:

$$\frac{H \vdash_{\mathcal{D}} \mathcal{D}(e) \quad H, P[x := e] \vdash_{\mathcal{D}} Q}{H, \forall x \bullet P \vdash_{\mathcal{D}} Q} \ \forall\text{-INTRODUCTION IN HYPOTHESIS}_{\mathcal{D}} \qquad \frac{H \vdash_{\mathcal{D}} \mathcal{D}(e) \quad H \vdash_{\mathcal{D}} P[x := e]}{H \vdash_{\mathcal{D}} \exists x \bullet P} \ \exists\text{-INTRODUCTION IN GOAL}_{\mathcal{D}}$$

$$\frac{H \vdash_{\mathcal{D}} \mathcal{D}(G) \quad H \vdash_{\mathcal{D}} G \quad G, H \vdash_{\mathcal{D}} P}{H \vdash_{\mathcal{D}} P} \ \text{CUT}_{\mathcal{D}}$$

The critical idea is that both the well-definedness and validity proofs are done in this variant of predicate calculus without the need for special machinery such as 3-valued logic and without the need to convert partial functions into total functions. The new sequent allows us to use implicitly the fact that each predicate is well-defined.

This means that we can use this new logic as a logic that preserves well-definedness but for which we can prove the validity of the inference rules in the traditional predicate calculus as [13]. We supplement the usual predicate logic, with the inference rules of equational logic (see [4] and [21]) and prove the validity of the three new inference rules of equational logic:

$$\frac{H \vdash_{\mathcal{D}} \mathcal{D}(P) \quad H \vdash_{\mathcal{D}} P \quad H \vdash_{\mathcal{D}} P \equiv Q}{H \vdash_{\mathcal{D}} Q} \ \text{EQUANIMITY}_{\mathcal{D}} \qquad \frac{H \vdash_{\mathcal{D}} \mathcal{D}(P \equiv Q) \quad H \vdash_{\mathcal{D}} P \equiv Q}{H \ \vdash_{\mathcal{D}} \ \alpha[x := P] \ \equiv \ \alpha[x := Q]} \ \text{LEIBNIZ}_{\mathcal{D}}$$

$$\frac{H \vdash_{\mathcal{D}} \mathcal{D}(Q) \quad H \vdash_{\mathcal{D}} P \equiv Q \quad H \vdash_{\mathcal{D}} Q \equiv R}{H \vdash_{\mathcal{D}} P \equiv R} \ \text{TRANSITIVITY}_{\mathcal{D}}$$

In [15], a more extensive example is used to demonstrate proofs of well-definedness and feasibility of queries.

When the query postcondition is in closed form "$Result = f(x)$" and where *Result* does not occur in $f(x)$ and $f(x)$ only refers to already introduced queries that themselves have been shown to be feasible, we then have a simpler proof obligation for feasibility:

$$\boxed{\textbf{CFF}(\text{closed form feasibility}): \quad \mathcal{D}(C_q(x)) \wedge (C_q(x) \Rightarrow \mathcal{D}(q(x)))}$$

Theorem: If $R_q(x, r)$ is in closed form, then query q is feasible.[2]

Therefore, **CFF** is sufficient for proving the feasibility of a query when its specification is expressed in closed form.

We use the WD proof obligation to filter out formulas that are not well-defined. We only try to prove the validity of formulas such as ϕ_2 that pass the filter. This use of the notion of well-definedness is consistent with the less formal style of mathematicians who intuitively avoid ill-defined statements and argue about partial functions and relations directly using their definitions without the need to pay attention to their preconditions. The assumption is that $x \geq 0$ is not actually used in the validity proof. It is needed only to ensure that we pass the WD proof obligation.

Well-definedness of Tabular Expressions. The theory presented above can be used to flag problems in various components of specifications, including tabular expressions (Section 3.5), queries and proofs. The well-definedness of a table is simply $\mathcal{D}$ applied to its meaning ((a) in Fig. 2). As for proofs, the main concern for well-definedness lies in the $\mathcal{D}$-terms created by the repeated application of the transitivity rule. The justification of each step might also create $\mathcal{D}$-terms and a careful analysis of their logical structure is necessary in order to find exactly which $\mathcal{D}$-terms are generated. An example of proof of well-definedness of tables and proofs in the context of the present case study is given in [15].

[2] A proof of this theorem can be found in [15].

Appendix B. Proving Small System Invariant in Z3 SMT Solver

```
from z3 import *  #z3 SMT solver

#monitored and controlled variables
#we use x0 and x for pre-state and post-state
x  = Real('x'); x0 = Real('x0'); y  = Real('y'); y0 = Real('y0'); z  = Real('z')

#encode in Z3 sqrt(r: REAL): REAL query
r=Real('r') #argument
epsilon = Real('epsilon')
sqrt = Function('sqrt@', RealSort(), RealSort())

#axiom 0 encodes the pre/post condition of MATH.sqrt
axm0 = ForAll ([r], Implies(0 <= r,
                            And(- epsilon <= r - sqrt(r) ** 2,
                                r - sqrt(r) ** 2 <= epsilon,
                                0 <= sqrt(r))))

#axiom 1 as in MATH
axm1 = And(0 < epsilon, epsilon <= 0.001)

guard   = -43.2 <= z
x_row_1 = Implies(z >= 0, x == sqrt(z) * x0 + y)
x_row_2 = Implies(And(-43.2 <= z, z < 0), x == x0)
y_row   = y0 * x0 <= y
table   = And(x_row_1, x_row_2, y_row)

def BA(): return Implies(guard, table) #before-after predicate

def invariant(x, y): return And(0 <= x, 0 <= y)

# Proof 1: axm0 and axm1 do not conflict with each other
# we replace occurrences of sqrt by the Z3 Sqrt and prove that
# the definition of axm0 is correct
axm0b = ForAll ([r], Implies(0 <= r,
                            And(- epsilon <= r - Sqrt(r) ** 2,
                                r - Sqrt(r) ** 2 <= epsilon,
                                0 <= Sqrt(r))))

s = Solver(); s.add(Not(axm0b)); s.add(axm1)
print s; res = s.check()
if res == unsat:
    print "....proved"
elif res == unknown:
    print "... failed to prove (unknown)"
else:
    print "...failed to prove (proposition is false)"
    s.model()
```

```
#Proof 2: invariant preservation
inv_proof_obligation = Implies(And(invariant(x0, y0), guard, BA()), invariant(x, y))

print ">␣Axiom␣0␣" ; print axm0; print(">␣Axiom␣1"); print axm1
print '>␣inv_proof_obligation'; print(inv_proof_obligation)

s = Solver(); s.add(axm0); s.add(axm1); s.add(Not(inv_proof_obligation))
print s; res = s.check()
if res == unsat:
    print "....proved"
elif res == unknown:
    print "...␣failed␣to␣prove␣(unknown)"
else:
    print "...failed␣to␣prove␣(proposition␣is␣false)"
    s.model()
```

Appendix C. Proving Case Study Invariant in Z3 SMT Solver

```
(declare-fun wf (Real) Real)
(declare-fun wf_first (Real Real Real) Real)
(declare-fun wf_last (Real Real Real) Real)
(declare-const amplitude Real)
(declare-const tmax Real)
(declare-const n Real)

(define-fun continuous () Bool
    (forall ((x Real) (y Real))
            (=> (and (<= 1 x) (<= x y) (<= y n))
                (exists ((z Real))
                        (and (<= x z) (<= z y)
                             (<= (wf x) (wf z))
                             (<= (wf z) (wf y)))))))

    ; definition of first and last
(define-fun def-first-a () Bool
     (forall ((x Real) (y Real) (z Real) (w Real))
            (=> (and (<= x w) (<= w y) (= (wf w) z))
                (<= (wf_first x y z) w))))

(define-fun def-first-b () Bool
    (forall ((x Real) (y Real) (z Real))
;               (=> (exists ((w Real)) (and (<= x w) (<= w y) (= (wf w) z)))
                   (= (wf (wf_first x y z)) z)))

(define-fun def-first-c () Bool
    (forall ((i Real) (j Real) (z Real) (w Real))
;         (=> (exists ((x Real)) (and (<= i x) (<= x j) (= (wf x) z)))
            (= (< w (wf_first i j z))
               (forall ((x Real))
                       (=> (and (<= i x) (<= x j) (= (wf x) z))
                           (< w x))))))

(define-fun def-first-d () Bool
    (forall ((x Real) (y Real) (z Real))
;               (=> (exists ((w Real)) (and (<= x w) (<= w y) (= (wf w) z)))
                   (and (<= x (wf_first x y z))
                        (<= (wf_first x y z) y))))

(define-fun def-last-a () Bool
     (forall ((x Real) (y Real) (z Real) (w Real))
            (=> (and (<= x w) (<= w y) (= (wf w) z))
                (<= w (wf_last x y z)))))

(define-fun def-last-b () Bool
    (forall ((x Real) (y Real) (z Real))
```

```
;                (=> (exists ((w Real)) (and (<= x w) (<= w y) (= (wf w) z)))
                     (= (wf (wf_last x y z)) z)))

(define-fun def-last-c () Bool
    (forall ((i Real) (j Real) (z Real) (w Real))
;         (=> (exists ((x Real)) (and (<= i x) (<= x j) (= (wf x) z)))
            (= (< (wf_last i j z) w)
               (forall ((x Real))
                       (=> (and (<= i x) (<= x j) (= (wf x) z))
                           (< x w)))))))

(define-fun def-last-d () Bool
    (forall ((x Real) (y Real) (z Real))
;                (=> (exists ((w Real)) (and (<= x w) (<= w y) (= (wf w) z)))
                    (and (<= x (wf_last x y z))
                         (<= (wf_last x y z) y)))))

(define-fun y10 () Real (* 0.1 amplitude))
(define-fun y50 () Real (* 0.5 amplitude))
(define-fun y90 () Real (* 0.9 amplitude))
(define-fun t50p () Real (wf_first 1 tmax y50))
(declare-const t50n Real)
(define-fun t10p () Real (wf_last 1 t50p y10))
(declare-const t90p Real)
(declare-const t10n Real)
(declare-const t90n Real)
(define-fun ymax () Bool (= (wf tmax) amplitude))
(define-fun tmax-between () Bool (and (<= 1 tmax) (<= tmax n)))
(define-fun pred0 () Bool (< t10p t50p))
(define-fun wd0a () Bool (and (exists ((x Real)) (and (<= 1 x) (<= x tmax) (= (wf x) y50))
(define-fun wd0b () Bool (and (exists ((x Real)) (and (<= 1 x) (<= x t50p) (= (wf x) y10))
(define-fun wd0c () Bool (and (<= 1 t50p)
                              (<= t50p n)))
(define-fun wd0d () Bool (and (<= 1 t10p)
                              (<= t10p n)))
(define-fun pred1 () Bool (< (wf_last 1 t50p y10) t50p))
(define-fun wd1a () Bool wd0a)
(define-fun wd1b () Bool wd0b)
(define-fun wd1c () Bool (and (<= 1 t50p)
                              (<= t50p n)))
(define-fun pred2 () Bool
     (forall ((t Real))
               (=> (and (<= 1 t) (<= t t50p) (= (wf t) y10))
                   (< t t50p) )))
(define-fun wd2a () Bool wd0a)
(define-fun wd2b () Bool (and (<= 1 t50p)
                              (<= t50p n)))
(define-fun pred3 () Bool
        (forall ((t Real))
                (=> (and (<= 1 t) (<= t t50p) (<= t50p t))
```

```
                    (not (= (wf t) y10) ))))
(define-fun wd3a () Bool wd0a)
(define-fun wd3b () Bool (and (<= 1 t50p)
                              (<= t50p n)))
(define-fun pred4 () Bool
        (not (= (wf t50p) y10)))
(define-fun wd4a () Bool wd0a)
(define-fun wd4b () Bool (and (<= 1 t50p)
                              (<= t50p n)))
(define-fun pred5 () Bool (not (= y50 y10)))
(define-fun pred6 () Bool (not (= (* 0.5 amplitude) (* 0.1 amplitude))))
(define-fun pred7 () Bool true)
(push)
(echo ">␣WD␣0␣(a)")
(assert wd0a)
(assert (not wd0a))
(check-sat)
(pop)
(push)
(echo ">␣WD␣0␣(b)")
(assert wd0b)
(assert (not wd0b))
(check-sat)
(pop)
(push)
(echo ">␣WD␣0␣(c)")
(assert def-first-d)
;(assert def-first-b)
(assert tmax-between)
(assert (not wd0c))
(check-sat)
(pop)
(push)
(echo ">␣WD␣0␣(d)")
(assert wd0d)
(assert (not wd0d))
(check-sat)
(pop)
(push)
(echo ">␣WD␣1␣(a)")
(assert wd1a)
(assert (not wd1a))
(check-sat)
(pop)
(push)
(echo ">␣WD␣1␣(b)")
(assert wd1b)
(assert (not wd1b))
(check-sat)
(pop)
```

```
(push)
(echo ">␣WD␣1␣(c)")
(assert def-first-d)
(assert tmax-between)
(assert (not wd1c))
(check-sat)
(pop)
(push)
(echo ">␣WD␣2␣(a)")
(assert wd0a)
(assert (not wd2a))
(check-sat)
(pop)
(push)
(echo ">␣WD␣2␣(b)")
;(assert wd0c)
(assert def-first-d)
(assert tmax-between)
(assert (not wd2b))
(check-sat)
(pop)
(push)
(echo ">␣WD␣3␣(a)")
(assert wd0a)
(assert (not wd3a))
(check-sat)
(pop)
(push)
(echo ">␣WD␣3␣(b)")
(assert def-first-d)
(assert tmax-between)
(assert (not wd3b))
(check-sat)
(pop)
(push)
(echo ">␣WD␣4␣(a)")
(assert wd0a)
(assert (not wd4a))
(check-sat)
(pop)
(push)
(echo ">␣WD␣4␣(b)")
(assert def-first-d)
(assert tmax-between)
(assert (not wd4b))
(check-sat)
(pop)
(push)
(echo ">␣step␣0")
(assert (not (= pred0 pred1)))
```

```
(check-sat)
(pop)
(push)
(echo ">␣step␣1")
;(assert (exists ((w Real)) (and (<= 1 w) (<= w tmax) (= (wf w) y50))))
;(assert def-last-a)
;(assert def-last-b)
(assert def-last-c)
(assert (not (= pred2 pred1)))
(check-sat-using
    (or-else
        (then simplify smt)
        (then qe smt)
        (then smt)))
(pop)
(push)
(echo ">␣step␣2")
(assert (not (= pred2 pred3)))
(check-sat)
(pop)
(push)
(echo ">␣step␣3")
(assert (not (=> pred4 pred3)))
(check-sat)
(pop)
(push)
(echo ">␣step␣4")
(assert (exists ((w Real)) (and (<= 1 w) (<= w tmax) (= (wf w) y50))))
(assert def-first-b)
(assert continuous)
(assert (not (= pred4 pred5)))
(check-sat)
(pop)
(push)
(echo ">␣step␣5")
(assert (not (= pred6 pred5)))
(check-sat)
(pop)
(push)
(echo ">␣step␣6")
(assert (< 0 amplitude))
(assert (not (= pred6 pred7)))
(check-sat)
(pop)
(push)
(echo ">␣relation")
(assert (= pred0 pred1))
(assert (= pred2 pred1))
(assert (= pred2 pred3))
(assert (=> pred4 pred3))
```

```
(assert (= pred4 pred5))
(assert (= pred6 pred5))
(assert (= pred6 pred7))
(assert (not (=> pred7 pred0)))
(check-sat)
(pop)
```