YORK
UNIVERSITÉ
UNIVERSITY

redefine THE POSSIBLE.

Precise Documentation of Requirements and Executable Specifications

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

Technical Report CSE-2012-03

June 11 2012

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

Precise Documentation of Requirements and Executable Specifications

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

Abstract

We propose a format for precise documentation of requirements to drive the development of dependable software products and to provide evidence for their certification. Requirements are elicited from customers and expressed informally as atomic English descriptions. To analyze the consistency of the requirements, we translate them into a software specification consisting of model contracts and tabular expressions. Model contracts describe queries as pre/post-conditions using mathematical constructs (e.g. quantifiers, sets, relations, sequences) which make them more expressive than classical implementation contracts. Tabular expressions use these queries to provide complete black-box descriptions of the system input-output relation. We validate the requirements via proofs of (a) the completeness, disjointness, and well-definedness of the specification and (b) the consistency between the specification and the atomic requirements. The model contracts are translated into an executable specification using MSL (model specification language). The executable specification plays a dual role. Even before code production, the specification is used to validate the requirements. Once the code is produced, we verify that the code satisfies the specification via runtime assertion checking.

Keyword: precise documentation of requirements, model contracts, well-definedness of tabular expressions, executable specifications, validation, runtime verification, certification.

Contents

List of Figures

List of Tables

1 Introduction

Software certification has emerged as an important issue for governments, consumers and software developers of safety or mission critical software such as medical devices, nuclear reactors or high assurance business systems. One challenge is how to develop software in a way that facilitates certification. The other is how to collect and use evidence about software products to evaluate whether they should or should not be certified for use. An effective standard is one that helps developers to produce systems of acceptable reliability and safety. It should also help certifiers to determine compliance with the standard.

A necessary part of product-based certification is the need for artifacts such as requirements documents, design specifications, and arguments showing that requirements have been validated and the software verified for compliance with the specifications [11]. In this paper we propose a format for precise documentation using model contracts. The format helps to validate the requirements. Requirements are translated into executable specifications that can help to check that code satisfies the requirements. The methods of this paper are applicable to transformational systems or reactive systems in which it is sufficient to check one step transition functions [9].

As a running example, we will use recent work with an industrial partner. The partner provided a few hundred lines of code taken from their software for a biomedical device where the interest is in monitoring vital signs such as blood pressure, heart rate and temperature. In Fig 3 we have identified the boundary of the pulse software and its operating environment. A reading from the device arrives as a sampled pulse. The software is required to calculate parameters of the pulse as defined by the IEEE Standard 181 on Transitions, Pulses, and Related Waveforms (see Fig. 1).

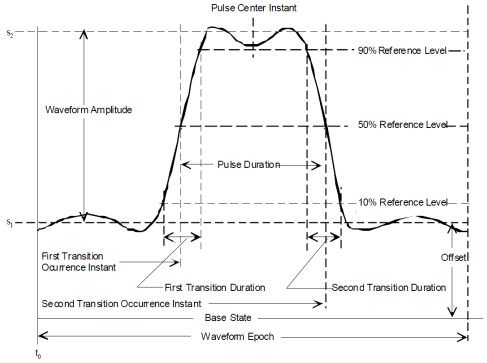

Figure 1: IEEE-181 standard (2011) for a Single Positive Pulse

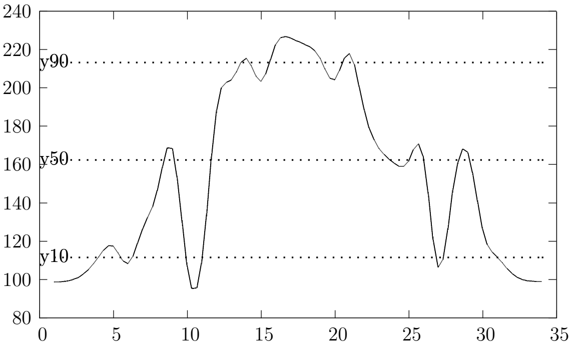

Figure 2: Is this a single positive pulse?

The sampled waveform is a plot of pressure levels (vertical y axis) versus time instants (horizontal t axis). In the IEEE-181 standard, a single positive pulse 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 times that are in-between the sampled time instants.

The software calculates a variety of parameters using floating point arithmetic. For each transition of the pulse this includes: the 10%, 50% and 90% levels/instants and transition duration. Pulse parameters such as amplitude, duration etc. must also be calculated, all according to IEEE-181 requirements.

The code and five sample pulses were provided as a small example of the challenges that our industrial partner is faced with in developing much bigger systems. This pulse example, although small, will be used to illustrate the use of precise documentation to drive software quality and provide some evidence for certification. Our industrial partner had the following questions:

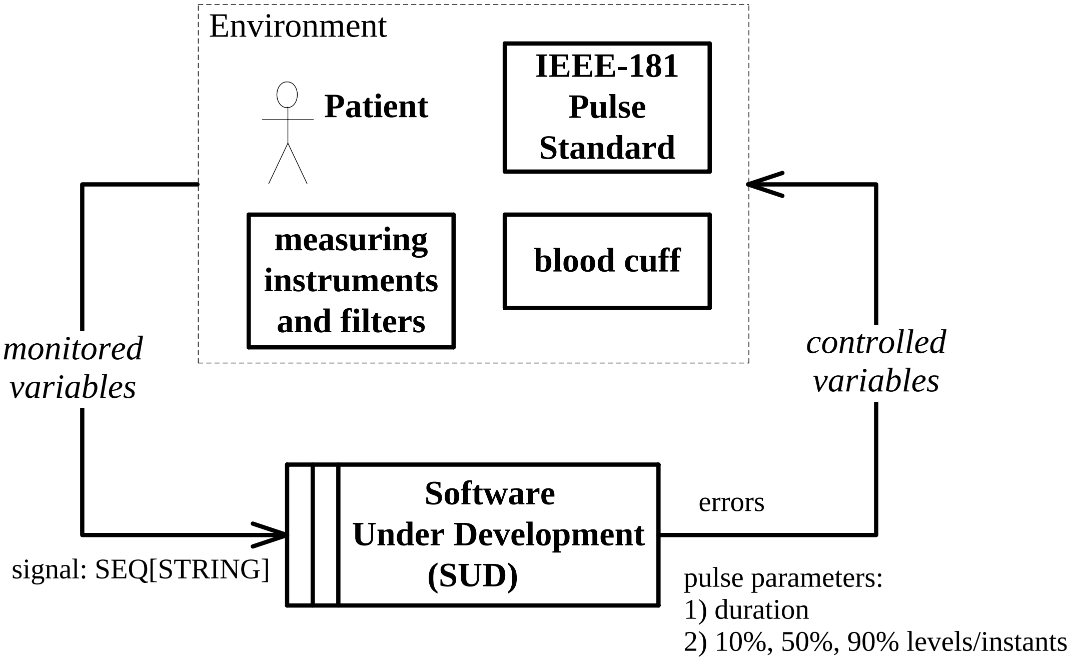

Figure 3: Context diagram for pulse software

1. How can we increase confidence that our code is correct. Are there static tools that can be used to check the code mechanically (such as detecting division by zero)? The need to deal with floating point arithmetic is an additional challenge.
2. How can we present arguments to certifying agencies such as the FDA that the code is safe and fit for use?
3. One of the major problems is detecting if an input signal is a single positive pulse. In healthy patients a blood pressure signal might be close to Fig. 1 representing a classical positive single pulse. But what about the signal in Fig. 2? For ill patients there may be significant variance from the classical shape and at some point we need to flag that the signal does not really represent a legal pulse. The code does have some error checking which is an implicit statement of what is acceptable as input. For example, the code signals an error when there are zero data points thus ruling out such input. However, providing a more complete specification is challenging. There are data sets where the code produces spurious results. On the other hand, unless we can state what we are calculating, we are unlikely to be able to demonstrate compliance with standards

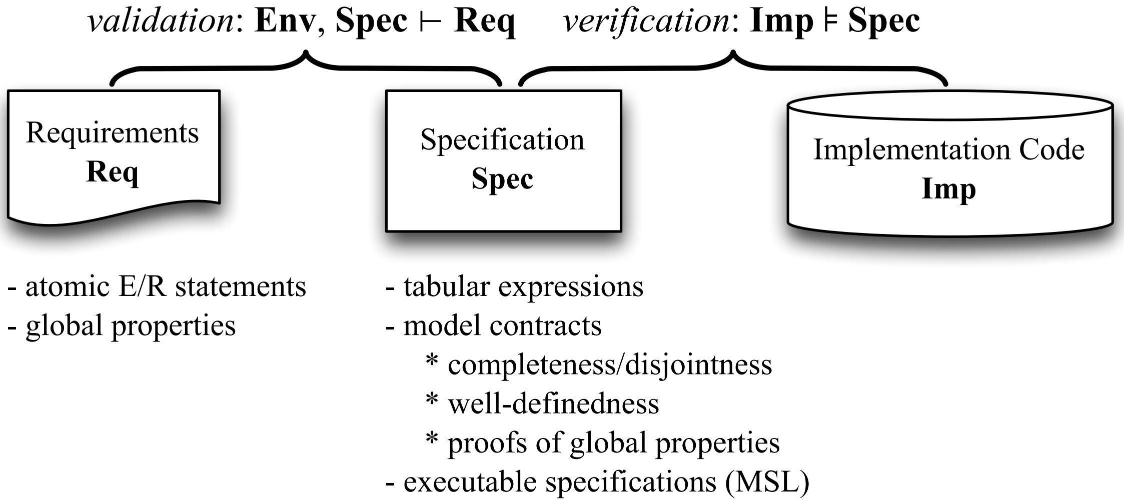

Figure 4: Validation and Verification entail: $Env, Imp \vdash Req$

Input conditions		Output r
$C_1(x)$	$C_{11}(x)$	$R_1(x,r)$
	$C_{12}(x)$	$R_2(x,r)$
$C_3(x)$		$R_3(x,r)$

Given : $P_1 \triangleq C_1(x) \wedge C_{11}(x)$
$P_2 \triangleq C_2(x) \wedge C_{12}(x)$
$P_3 \triangleq C_3(x)$
$Q_i \triangleq R_i(x,r)$ for $i \in 1\,..\,3$

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

Figure 5: Completeness, Disjointness and Well-definedness of tabular expressions

requiring that medical devices be safe and efficacious. Thus a complete specification of behaviour is imperative. But how do we do that? The IEEE standard provides no guidance on when a signal is a a valid single pulse or not.

Static checkers are capable of finding inconsistencies in the implementation (null pointers, indices out of bound, division by zero, etc.). Likewise strongly typed languages provide certain consistency guarantees in implemented code. But even if the implementation is internally consistent we still do not know that we are building the right product, one that satisfies the goals of our customers. The most severe problems occur at the requirements stage and at the software's interface with the rest of the system [10]. Accidents involving software occur, not because the software failed to meet its requirements, but because the requirements weren't the right requirements.

At the very least we need two artifacts: the program code and a *specification* that captures customers goals. The code describes what executions will do. The specification describes what they are supposed to do. The terms verification and validation are commonly used in software engineering to mean two different types of analysis (Fig. 4). In validation we ask: Are we building the right system? In verification we ask: are we building the system right? Validation is the process of checking whether the specification captures the customers needs, while verification is the process of checking that the software meets the specification. Both types of analysis require the existence of a complete and unambiguous specification.

The specification for the pulse code is supposedly the IEEE-181 standard. However, that standard itself is ambiguous in some places and not always complete. For example, there is no guarantee that an input signal will be monotonically increasing in a positive transition (see Fig. 2). There may thus be multiple 50% instants. The standard specifies that the first one must be used. This makes sense for the positive transition but seems an inconsistent choice for the negative transition, in which case the last 50% seems more appropriate. At the very least, the standard should explicitly specify which instant to choose for each transition. More such examples will emerge in the sequel.

The pulse code provided by our industrial partner fails to signal input errors and does not calculate parameters correctly due to the absence of precise requirements. If one submits the input signal $\langle 0, 1 \rangle$ to their code, no error is returned and the parameters are thus calculated. One might imagine that all is in order. But, when we examine outputs such as pulse duration and other parameters what is returned is NaN (not a number) due to division by zero and other issues. Input signal $\langle 0, 1, 0 \rangle$ produces no error (which is correct) and no NaN in the outputs. But, $\langle 0, 1, 1 \rangle$ also produces no error (which is incorrect) even though may of the parameters calculated return NaN. Precise requirements and validation of the requirements (e.g. via checks for completeness, disjointness and well-definedness) will address these issues.

2 Precise documentation: overview of our contribution

The lack of precise requirements is the most critical problem in the pulse software to the point that the developed code fails to signal input errors and does not calculate the parameters correctly. The effort that went into building the code was premature. The first question to address is "are we building the right system"?

Even medium sized projects typically involve hundreds of requirements needing organization into a proper hierarchical structure. In addition to proper organization, we propose a format for rigorous mathematical documentation for validating requirements. An overview of our notion of precise documentation is shown in Fig 4. In our approach, a requirements document (RD) contains a context diagram, E/R-descriptions (atomic environment and requirement descriptions), and a formal *specification* using model contracts and tabular expressions.

Given that requirements are about the phenomena of the environment, there is a need to discover the system boundary with monitored variables as the inputs and controlled variables as the output [6, 14, 4]. The context diagram (Fig. 3) and English-language atomic E-statements capture assumptions about the environment in which the software product must work and delineate the system boundary. In the pulse example, E-statements (Section 3) differentiate between valid and invalid signals. R-statements (Section 5) describe the required calculation of parameters for valid pulses and the error codes for invalid pulses. Each R-description should be atomic (each description carries a single traceable element), clear (everyday language is perhaps the only medium that all users and developers share), verifiable (there is some way to test it) and abstract (does not impose a specific solution or implementation).

How can we organize our requirements in such a way that we can argue for their completeness? Black-box input-output relations specified as tabular expressions (Fig. 5) have straightforward proof obligations for completeness and disjointness [17]. But, complex systems will usually require the use of auxiliary functions in tabular expressions. A critical issue is how to consistently introduce model contracts involving partial functions. We contribute the following:

- We use model contracts (Section 6) involving pre/post conditions to document auxiliary functions used in the tabular expressions. The pre/post conditions use standard mathematics (predicate logic, set theory, etc.). Modules (Section 4) are used to organize the functions into related units for simplicity and compositional reasoning. A *specification* is the tabular expression documenting the input-output function together with the model contracts. Specifications will be used to validate the E/R-statements. For the pulse example, the complete specification is less than two pages (Fig. 6 and Table 1).
- Preconditions entail that our functions are partial and thus tabular expressions may not be well-defined. We provide proof obligations for determining when tabular expressions are well-defined. The well-definedness of a tabular expression is provided in Fig. 5 (see (d)). A table is well-defined if all its rows are well-defined. Section 9 discusses the necessary proof obligations.
- As analysis proceeds, we are usually able to derive important global properties. These global properties are documented as atomic R-statements (for confirmation by our customers and domain experts). We provide the proof obligations for establishing that the specification (model contracts and tabular expressions) entails the global properties (see Section 8).

The software to be developed addresses a specific domain of discourse. It might be nuclear reactors or medical devices and pulse signals as in our case. There is a need to devise a suitable (and, if possible, re-usable) theory to deal with the domain in the spirit of abstract data types where we list the applicable operations and their properties. Meyer has pointed out that: "If we do not treat this task as a separate step, we end up with the kind of specification that works for toy examples but quickly becomes unmanageable for real-life applications. Most of the verification literature, unfortunately, relies on such specifications. They lack abstraction since they keep using the lowest-level mathematical objects and constructs, such as numbers and quantified expressions. They are to specification what assembly language is to modern programming"[13]. At the specification level we need the full expressive capability of abstract mathematics including the freedom to invent new notation. During the development of the formal specification, it is essential to find a suitable abstraction for describing the essential properties of the system under development. For the pulse software, a real-value interpolation function is the appropriate abstraction for linking the input (sampled waveform) with the calculation of the output parameters (see Section 7).

What evidence can we provide for validating of the requirements and verifying the software? In the previous sections of this paper we provided a method for developing a precise requirements document (RD) using model contracts and tabular expressions. Given such an RD, we have a hierarchy of increasing assurance as follows:

1. *Testing and inspection of the software against the* RD: Testing commences once the code is produced. This is late in the process. If defects are discovered the rework can significantly impact on the cost and timescale of the process, especially if there are defects or inconsistencies in the design or the RD.
2. *The use of executable specifications* via runtime assertion checking (see MSL below): Before any code is developed, executable specifications do automatic type checking and are used to check for completeness, disjointness and well-definedness of the tabular expressions. They are also used to check the consistency of global properties (such as REQ9). While developing production code, executable specifications are used to check that the implementations satisfy the specification.
3. *Proofs*: The highest level of assurance is where both validation and verification are done via proofs. In this paper we developed an appropriate calculus for such proofs (e.g. see Fig 8) based on our notion of documentation and model contracts. While substantial progress has been made in mechanizing such proofs, there are still many challenges [5].

Where the highest level of assurance is needed, we provide the necessary proof obligations. In Section 11, we also provide executable specifications by extending MSL (Model Specification Language [15]) to support floating point arithmetic and real-valued functions. Model contracts can be stated in MSL allowing for more abstract descriptions than classical contracts (which are more suitable for checking implementation consistency). Such executable specifications are used to validate requirements by checking completeness, disjointness, well-definedness and consistency between tabular expressions and global properties. This may be done before any code is developed. Once the software is developed, the implemented code can be checked against the executable specifications. In the conclusion (Section 12) we briefly review how the methods of this paper can be applied to the challenges presented by the pulse example.

3 Atomic E-descriptions

As mentioned earlier, we use the pulse example to illustrate our notion of precise documentation. Consider the context diagram for the pulse code in Fig. 3 (using a notation adapted from [7]). The context diagram describes the boundary between the software system under development (SUD) and the environment in which that software operates. Requirements must take into account phenomena in the environment. In fact, the software satisfies its requirements if it produces the required effect in the problem domain, i.e. in the environment. Such requirements will be documented with atomic R-descriptions. We will also have E-descriptions to document assumptions that we make about the environment. We now illustrate R/E-descriptions for the pulse example.

On the one hand we had (a): the code provided by our industrial partner. We also had (b): the IEEE-181 standard, which was used to produce a requirements document sufficiently rigorous to derive (predict) the output given an arbitrary input signal (e.g. the input in Fig. 2). We then compared the results of (b) with (a). There are some remarkable divergences that illustrate the issues we have raised.

One divergence is in the calculation of the pulse amplitude and the resulting 10%, 50% and 90% levels of the input signal. For example, the code (a) yielded a 50% level of 162.3461 mmHg for the positive transition and 162.5000 for the negative transition whereas our document (b) yielded 162.3461 for both transitions. Why this divergence?

Section 5.1 of the standard asserts that the 50% level of a two state waveform is calculated from the pulse amplitude, i.e. the difference between the high state $s2$ and the low state $s1$ of the signal. A variety of ways are provided for determining the low and high levels including histograms, Short-estimators and peak methods as in Section 5.2.3.1: "Determine the maximum peak and minimum peak values of ... the *single pulse waveform*: 1) Take the minimum peak value as the low or first state level. 2) Take the maximum peak value as the high or second state level."

Contra (a), the standard does not mention calculations of different peaks for each transition. Rather there is a single peek for the pulse and hence the same 50% level for each transition as per (b). When questioned, our industrial partner answered that their input signals are somewhat asymmetric and thus they decided to calculate the amplitude and levels per transition. The standard would be clearer if it had explicitly addressed all possibilities including asymmetric pulses. In fact, the methods discussed in this paper might very well be used by standards organizations and would help to ensure that the 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.	A pulse is valid iff $s3 \wedge um \wedge t50?$ (modular specification Fig. 6)

The crucial point is that assumptions about the environment in which the software will be operating need to be explicitly documented in atomic descriptions such as ENV1. Without that we will not be able to check the correctness of the code.

ENV1 defines a valid single positive pulse. There must be at least 3 samples, a unique maximum and both 50% levels must exist. Without these constraints, we cannot partition the input pulse into two transitions and thus an error must be signalled. ENV2 documents our assumption that 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.	See levels $y10$, $y50$ and $y90$ in Table 2 and in Fig. 6 (waveform module).

An atomic description consists of at last three parts: the description number (e.g. ENV1), an informal statement in English and a cross reference to the mathematical model. The English statement allows us to communicate with our customers when we validate the RD and it may also describe global safety properties that the system as a whole must satisfy. The description number provides traceability through to the design, the code and the acceptance tests. (An atomic description might have more attributes such as the date it was documented, the author, the basis for the assumption and so on; and there are tools such as DOORS for keeping track of such statements).

The input is a file which is a sequence of strings ($SEQ[\mathbb{S}]$, where $\mathbb{S}$ is the set of all strings). The input file represents a sampled waveform of blood pressure from the patient. Each string in the sequence is supposed to represent a double precision value that satisfies the IEEE-758 real number specification. However, there is no guarantee that the input data has not been corrupted, or there may have been overflow in upstream instrumentation so that strings might represent *special* values such as *+Infinity*, *-Infinity*, and *NaN*. We let $\mathbb{F}$ denote the set of all *finite* floats (i.e. no specials). The boolean query vf (valid file) holds precisely when all the strings in the file can be converted into finite floats.[1]

4 Modular specifications

It is easier to understand and analyze specifications if they can be decomposed into modules. Fig. 6 provides modular specifications such as s-SIGNAL. SIGNAL is a module template and s is an instance of that template. The sampled input waveform is represented as $swf : SEQ[\mathbb{F}]$ with precondition vf.

Boolean queries $s3$ (are there at least 3 samples?) and um (is there a unique maximum?) are also defined. For example $ymax$ (the maximum peek level of the pulse) is defined as $(\uparrow i | 1 \leq i \leq n \bullet swf(i))$ where $\uparrow$ is the maximum quantifier, and with precondition $s3 \triangleq n \geq 3$ (where $n \triangleq swf.count$) to ensure that $ymax$ is well-defined. The unique maximum query um is defined as $(\# i | 1 \leq i \leq n \bullet swf(i) = ymax) = 1$, using the counting quantifier.

Our module specifications are like UML object diagrams in terms of state sharing, e.g. module p (an instant of PULSE) and *negative* and *positive* (instances of TRANSITION) share the module instance w. However, in another sense, module diagrams are unlike object diagrams. Object diagrams usually show only object attributes. Module templates provide all the features of the module. More importantly, objects (instances of classes) can be created and linked dynamically and thus their diagrams only illustrate examples of state. By contrast, module instances and their interconnections exist right from the beginning, i.e. the structure is static throughout the computations of the system, and their diagrams characterize the general state. Thus $p.negative.w$ always refers to the same module as $p.positive.w$ (the state of module w is thus shared by modules *positive* and *negative*). This removes the need for any aliasing analysis between modules (additional constructs will be needed for those situations in which aliasing is required to be dynamic).

Modules thus simplify specifications by allowing us to partition the state and encapsulate the resulting parts (and their relevant operations). Implementations are free to choose a different partition. The main effect of the use of modules is on the intellectual complexity of the specification (a "bad" partition will thus make it harder on the developer). Module composition is thus defined as the union of sub-module variables and operations.

Although the analogy is not precise, we have used the UML composition relation

[1] Programing languages come with built-in functions to check if a string is a finite float and to convert from $\mathbb{S}$ to $\mathbb{F}$.

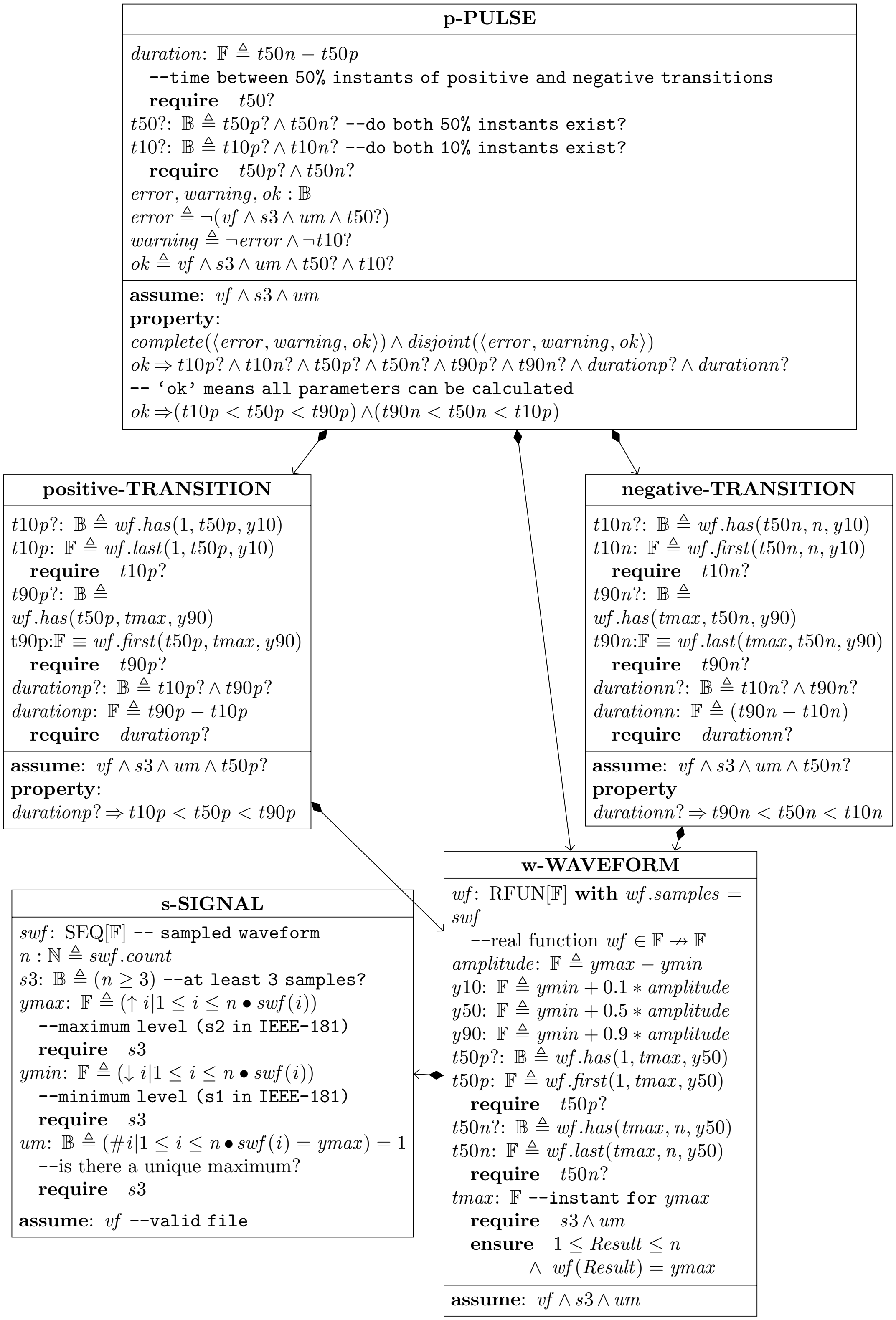
p-PULSE
$duration$: $\mathbb{F} \triangleq t50n - t50p$
--time between 50% instants of positive and negative transitions
require $t50?$
$t50?$: $\mathbb{B} \triangleq t50p? \wedge t50n?$ --do both 50% instants exist?
$t10?$: $\mathbb{B} \triangleq t10p? \wedge t10n?$ --do both 10% instants exist?
require $t50p? \wedge t50n?$
$error, warning, ok : \mathbb{B}$
$error \triangleq \neg(vf \wedge s3 \wedge um \wedge t50?)$
$warning \triangleq \neg error \wedge \neg t10?$
$ok \triangleq vf \wedge s3 \wedge um \wedge t50? \wedge t10?$
assume: $vf \wedge s3 \wedge um$
property:
$complete(\langle error, warning, ok \rangle) \wedge disjoint(\langle error, warning, ok \rangle)$
$ok \Rightarrow t10p? \wedge t10n? \wedge t50p? \wedge t50n? \wedge t90p? \wedge t90n? \wedge durationp? \wedge durationn?$
-- 'ok' means all parameters can be calculated
$ok \Rightarrow (t10p < t50p < t90p) \wedge (t90n < t50n < t10p)$
positive-TRANSITION
$t10p?$: $\mathbb{B} \triangleq wf.has(1, t50p, y10)$
$t10p$: $\mathbb{F} \triangleq wf.last(1, t50p, y10)$
require $t10p?$
$t90p?$: $\mathbb{B} \triangleq$
$wf.has(t50p, tmax, y90)$
t90p:$\mathbb{F} \equiv wf.first(t50p, tmax, y90)$
require $t90p?$
$durationp?$: $\mathbb{B} \triangleq t10p? \wedge t90p?$
$durationp$: $\mathbb{F} \triangleq t90p - t10p$
require $durationp?$
assume: $vf \wedge s3 \wedge um \wedge t50p?$
property:
$durationp? \Rightarrow t10p < t50p < t90p$
negative-TRANSITION
$t10n?$: $\mathbb{B} \triangleq wf.has(t50n, n, y10)$
$t10n$: $\mathbb{F} \triangleq wf.first(t50n, n, y10)$
require $t10n?$
$t90n?$: $\mathbb{B} \triangleq$
$wf.has(tmax, t50n, y90)$
$t90n$:$\mathbb{F} \equiv wf.last(tmax, t50n, y90)$
require $t90n?$
$durationn?$: $\mathbb{B} \triangleq t10n? \wedge t90n?$
$durationn$: $\mathbb{F} \triangleq (t90n - t10n)$
require $durationn?$
assume: $vf \wedge s3 \wedge um \wedge t50n?$
property
$durationn? \Rightarrow t90n < t50n < t10n$
w-WAVEFORM
wf: RFUN[$\mathbb{F}$] **with** $wf.samples = swf$
--real function $wf \in \mathbb{F} \nrightarrow \mathbb{F}$
$amplitude$: $\mathbb{F} \triangleq ymax - ymin$
$y10$: $\mathbb{F} \triangleq ymin + 0.1 * amplitude$
$y50$: $\mathbb{F} \triangleq ymin + 0.5 * amplitude$
$y90$: $\mathbb{F} \triangleq ymin + 0.9 * amplitude$
$t50p?$: $\mathbb{B} \triangleq wf.has(1, tmax, y50)$
$t50p$: $\mathbb{F} \triangleq wf.first(1, tmax, y50)$
require $t50p?$
$t50n?$: $\mathbb{B} \triangleq wf.has(tmax, n, y50)$
$t50n$: $\mathbb{F} \triangleq wf.last(tmax, n, y50)$
require $t50n?$
$tmax$: $\mathbb{F}$ --instant for $ymax$
require $s3 \wedge um$
ensure $1 \leq Result \leq n$
$\wedge\ wf(Result) = ymax$
assume: $vf \wedge s3 \wedge um$
s-SIGNAL
swf: SEQ[$\mathbb{F}$] -- sampled waveform
$n : \mathbb{N} \triangleq swf.count$
$s3$: $\mathbb{B} \triangleq (n \geq 3)$ --at least 3 samples?
$ymax$: $\mathbb{F} \triangleq (\uparrow i | 1 \leq i \leq n \bullet swf(i))$
--maximum level (s2 in IEEE-181)
require $s3$
$ymin$: $\mathbb{F} \triangleq (\downarrow i | 1 \leq i \leq n \bullet swf(i))$
--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 $s3$
assume: vf --valid file

Figure 6: Modular Specification for IEEE-181 single positive pulses

between modules to denote module dependencies. Following the dependency arrows, module instance *positive* depends on w which depends on module s. Thus, for example, an expression *tmax* in module *positive* should really be denoted $w.s.tmax$. However, where there is no ambiguity, we may use *tmax* without the module qualifiers. Section 6 provides more detail of modular specifications.

5 Atomic R-descriptions

Having defined what a valid (and invalid) pulse is, we can now write the appropriate R-descriptions.

REQ3	**ok**: If the input pulse is *valid* and the 10% levels of both transitions exist then output all the parameters: (a) For each transition: 10%, 50% and 90% levels and instants. (b) For each transition: the transition duration (i.e. time from the 10% instant to the 90% instant). (c) The pulse duration (time from the 50% instant of the positive transition to the 50% instant of the negative transition).	See IEEE-181. The input pulse is valid and the 10% levels exist iff query *ok* in the pulse module (Fig. 6) holds. This corresponds to the first row of the tabular expression in Table 1. Output parameters are listed in Table 2 and specified in Fig. 6.

REQ4	**Warning**: If the input pulse is *valid* and at least one of the 10% levels is missing then output all the parameters except for the missing 10% levels and instants (and associated transition duration) and issue a warning.	See pulse module (Fig. 6) for query *warning*, corresponding to 3 warning rows in grey in Table 1. See Table 4.

REQ5	**Error**: If the input pulse is *invalid* then no parameters are calculated and appropriate error messages are printed.	See query *error* in pulse module Fig. 6. Error message are in Table 4.

6 Software specification as model contracts and tabular expressions

The modular specification of the pulse software in Fig. 6 contains queries defined by model contracts (pre/post-conditions). These queries simplify the description of the input-output behaviour of the software systems as a tabular expression (Table 1). The model contracts (organized by modules) and the tabular expression together constitute the software specification. This specification will be used to validate the requirements via checks for its completeness, disjointness and well-definedness and by proving that the specification entails global properties described in the requirements. Initially, some of the global properties were omitted in the atomic R-descriptions. The validation process caused us to realize that these requirements were necessary. We provide an example in the sequel.

We have already explained the the module s (instance of module template SIGNAL). We now explain some aspects of module w (instance of WAVEFORM) and module *positive* (instance of TRANSITION) in Fig. 6. Section 5.3.1 of the IEEE-181 standard provides the procedure for calculating the 50% level $w.y50$ and corresponding instances $t50p$ ($t50n$) of the positive (negative) transition.

The standard does not specify (and the code from our industrial partner does not check) that $w.y50$ can only be calculated if the precondition $vf \wedge s3 \wedge um$ holds. To address this defiency in the standard, we use an "**assume**: $vf \wedge s3 \wedge um$" clause in the waveform module to assert that the precondition $vf \wedge s3 \wedge um$ applies to all queries in module. The expression $w.y50$ may only be used in a context in which its precondition holds, and it is then said to be *well-defined* (see Section 9).

7 Critical abstraction in the pulse software specification

Section 5.3.1 (eqn. 5) of the IEEE-181 standard provides a linear interpolation formula for calculating in-between instants (e.g. the 50% instant $positive.t50p$). A problem with the standard is that the formula is undefined for integer values of t (representing the original samples) due to division by zero. Also, there may be multiple 50% instants. The standard specifies that the first one must be used. This makes sense for the positive transition but seems an inconsistent choice for the negative transition, in which case the last 50% seems more appropriate. At the very least, the standard should explicitly specify which instant to choose for each transition. To address these problems in the standard (see the environment in the context diagram Fig. 3), we introduce the following E-description:

ENV6	Associated with the sampled waveform swf, there is an approximation wf to the continuous waveform using linear interpolation.	wf: RFUN[$\mathbb{F}$] is in the waveform module in Fig. 6

Given an arbitrary sequence of float inputs (i.e. swf), we need a well-defined query that returns a level for an arbitrary instant t. The specification thus introduces an interpolated real wave-function wf: RFUN[$\mathbb{F}$] in the waveform module. Fig. 7 provides the relevant queries for such interpolated real functions. For example, $wf.image(t)$ returns the interpolated level for arbitrary instant t under the precondition $1 \leq t \leq n$. For simplicity, we abbreviate $wf.image(t)$ by $wf(t)$. Fig. 7 also provides the definitions for $wf.has(t_1, t_2, y)$, $wf.first(t_1, t_2, y)$ and $wf.last(t_1, t_2, y)$.[2] The remaining atomic requirements are:

REQ7	• If the positive transition has multiple 50% instants, then output the first such instance. • If the negative transition has multiple 50% instants, then output the last such instance.	Clarification of an ambiguity in IEEE-18. See queries $t50p$ and $t50n$ in the waveform module Fig. 6.

REQ8	Output the 10% and 90% instants closest to the 50% instants.	Stated in section 5.3.3.2 of IEEE-181

REQ9	• For the positive transition: $t10p < t50p < t90$. • For the negative transition: $t90n < t50n < t10n$.	Clarification of an ambiguity in IEEE-181. See global properties in pulse module Fig. 6.

[2] The relationship between the continuous waveform wf and the sampled waveform swf is $swf = 1..n \lhd wf$ (using the domain restriction operator). 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 $wf(t)$ 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.

RFUN[FLOAT]

$\mathit{samples}$: SEQ[$\mathbb{F}$] --sampled y values
count: $\{1..\mathit{samples}.\mathit{count}\}$
image (x: $\mathbb{F}$): $\mathbb{F} \triangleq (\mathit{samples}[\lfloor x \rfloor] \times (\lfloor x+1 \rfloor - x) + \mathit{samples}[\lceil x \rceil] \times (x - \lfloor x \rfloor))$
  --ordinate y for abscissa x
  require $1 \leq x \leq \mathit{count}$
$\mathit{has}(x_1,\ x_2,\ \mathrm{y}{:}\mathbb{F})$: $\mathbb{B}$ --does y occur between x_1 and x_2?
  ensure
    $(1 \leq x_1 \leq x_2 \leq \mathit{count}) \Rightarrow (\mathit{Result} \equiv (\exists x : \mathbb{F} | x_1 \leq x \leq x_2 \bullet \mathit{image}[x] = y))$
    $\neg(1 \leq x_1 \leq x_2 \leq \mathit{count}) \Rightarrow (\mathit{Result} \equiv \mathit{false})$
$\mathit{first}(x_1,\ x_2,\ \mathrm{y}{:}\ \mathbb{F})$: $\mathbb{F} \triangleq (\downarrow\ x : \mathbb{F} | x_1 \leq x \leq x_2 \wedge \mathit{image}[x] = y \bullet x)$
  --first instance of level y in interval $[x_1, x_2]$
  require $\mathit{has}(x_1, x_2, y)$
$\mathit{last}(x_1,\ x_2,\ \mathrm{y}{:}\ \mathbb{F})$: $\mathbb{F} \triangleq (\uparrow\ x : \mathbb{F} | x_1 \leq x \leq x_2 \wedge \mathit{image}[x] = y \bullet x)$
  --last instance of level y in interval $[x_1, x_2]$
  require $\mathit{has}(x_1, x_2, y)$

Figure 7: Specification of real functions in floats

Prove: $\mathit{durationp}? \Rightarrow (t10p < t50p)$

$$
\begin{array}{ll}
 & t10p < t50p \\
= & \quad \langle \text{defn. of } t10p \text{ in module } \mathit{positive} \text{ in Fig. 6 and } \mathit{durationp}? \Rightarrow t10p? \rangle \\
 & \mathit{wf}.\mathit{last}(1, t50p, y10) < t50p \\
= & \quad \langle \text{defn. of RFUN.}\mathit{last} \rangle \\
 & (\uparrow\ t : \mathbb{F} | 1 \leq t \leq t50p \wedge \mathit{wf}(t) = y10 \bullet t) < t50p \\
= & \quad \langle \uparrow \text{ and } < \rangle \\
 & (\forall t : \mathbb{F} | 1 \leq t \leq t50p \wedge \mathit{wf}(t) = y10 \bullet t < t50p) \\
= & \quad \langle \text{trading} \rangle \\
 & (\forall t : \mathbb{F} | 1 \leq t \wedge t \leq t50p \wedge t50p \leq t \bullet \mathit{wf}(t) \neq y10) \\
\Leftarrow & \quad \langle \text{drop first conjunct in range; anti-symmetry of } \leq \rangle \\
 & (\forall t : \mathbb{F} | t = t50p \bullet \mathit{wf}(t) \neq y10) \\
= & \quad \langle \text{one point rule} \rangle \\
 & \mathit{wf}(t50p) \neq y10 \\
= & \quad \langle \mathit{wf}(t50p) = y50 \text{ and defn. of } t50p \text{ in waveform} \rangle \\
 & \neg(y50 = y10) \\
= & \quad \langle \text{defn. of } \mathit{amplitude} \text{ in waveform with its assumption} \rangle \\
 & \neg((\mathit{ymin} + 0.5 * \mathit{amplitude}) = (\mathit{ymin} + 0.1 * \mathit{amplitude})) \\
= & \quad \langle \text{arithmetic and } \mathit{amplitide} \neq 0 \rangle \\
 & \neg(0.5 = 0.1) \\
= & \quad \langle 0.5 \neq 0.1 \rangle \\
 & \mathit{true}
\end{array}
$$

Figure 8: Proving a property of module *positive* that also validates REQ9

8 Validating requirements via proofs

REQ9 addresses a scenario in which two 10% (or 90%) instants are equally close to $t50$, e.g. where the first instant occurs before $t50$ and the second occurs after $t50$. During the analysis of the IEEE-181 standard via the modular specification (Fig. 6), we realized that the standard does not explicitly specify which instant to choose in this scenario; hence the need to introduce REQ9 asserting that the instants must be in the obvious order.

How do we reflect this new requirement into the specification? We do this by transforming REQ9 into properties in the modules in Fig. 6. A **property** of a module is a predicate that must be proved using just the definitions of features in the module and its dependents. For example, in the positive transition module we have the property $durationp? \Rightarrow t10p < t50p < t90p$ (likewise for the negative transition). Part of the proof is provided in Fig. 8.

How do we transform the atomic requirement REQ9 into a formal specification? It is declared as a property proof obligation $ok \Rightarrow (t10p < t50p < t90p) \wedge (t90n < t50n < t10n)$ in the pulse module (the top module in Fig. 6). The proof follows directly from the proofs already performed in the two lower transition modules. This proof demonstrates the consistency between the modular specification and the atomic description REQ9, hence, an important component of requirements validation.

Completeness and disjointness of tabular expressions are described in (b) and (c) of Fig. 5. The tabular expression in Table 1 specifies the input-output behaviour of the pulse software. The rows of the table can be divided into three disjoint groups. 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). The rows below the grey area correspond to query *error* (invalid input, thus no parameters can be calculated). Completeness ($error \vee warning \vee ok$) and disjointness of the pulse specification also appear as properties to be proved in the pulse module (see Fig. 6). These proofs are part of requirements validation.

Tabular expressions (e.g. Table 1) 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 of queries in Fig. 6 is used to prove that REQ9 holds as a logical consequence of the tabular expression.

<table>
<tr><th colspan="6">conditions on input</th><th>Error</th><th>Warning</th><th>Parameters</th></tr>
<tr><td rowspan="7">vf</td><td rowspan="6">$s3$</td><td rowspan="5">um</td><td rowspan="4">$t50?$</td><td colspan="2">$t10?$</td><td>no</td><td>no</td><td>all (see Table 1)</td></tr>
<tr><td rowspan="3">$\neg t10?$</td><td>$\neg t10p?$</td><td>no</td><td>1</td><td>not $t10p$</td></tr>
<tr><td>$\neg t10n?$</td><td>no</td><td>2</td><td>not $t10n$</td></tr>
<tr><td>$\neg t10?$</td><td>no</td><td>3</td><td>no $t10$</td></tr>
<tr><td colspan="3">$\neg t50?$</td><td>1</td><td rowspan="5">no</td><td rowspan="5">none</td></tr>
<tr><td colspan="4">$\neg um$</td><td>2</td></tr>
<tr><td colspan="5">$\neg s3$</td><td>3</td></tr>
<tr><td rowspan="2">$\neg vf$</td><td colspan="5"><i>junk</i></td><td>4</td></tr>
<tr><td colspan="5"><i>specials</i></td><td>5</td></tr>
<tr><td colspan="6">output overflow</td><td>6</td><td>no</td><td>none</td></tr>
</table>

Table 1: Function table for requirements (see Table 3 for conditions)

Parameter	**Description**
duration	pulse duration
$y10$	10% level
$y50$	50% level
$y90$	90% level
durationp	positive transition duration
$t10p$	positive 10% instant
$t50p$	positive 50% instant
$t90p$	positive 90% instant
durationn	negative transition duration
$t10n$	negative 10% instant
$t50n$	negative 50% instant
$t90n$	negative 90% instant

Table 2: Output Parameters

Ab.	**Meaning**
vf	is input file valid?
$s3$	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 $t10p$ exist? ($t10p$ is positive transition instant for level $y10$)
$t10n?$	does $t10n$ exist? ($t10n$ is negative transition instant for level $y10$)

Table 3: Conditions

#	Error	Warning
1	file has junk strings	No $t10p$ instant, positive transition duration
2	file has float specials	No $t10n$ instant, negative transition duration
3	file lacks 3 finite floats	No $t10$ instants/duration for either transition
4	file lacks unique peak	
5	file lacks both 50 percent levels	
6	output overflow	

Table 4: Errors and Warnings

9 Well-definedness of queries in specification modules

What is the effect of adding a new query q in a specification module? It 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 [8, 16], 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. The theorem prover PVS has been used to provide tool support for tabular expressions [9, 19, 2]. In PVS, partial functions are converted into total functions using predicate subtyping which generates type checking proof obligations.

Model contracts presume that functions and relations will be partial. We thus seek a logic in which 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). We thus preserve the ability to introduce partial functions (without the complexity and bloat of converting them into total functions) while using standard tools and mathematical conventions. Consider the following query with its pre/post-conditions in a specification module:

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

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 [3]. The query can be safely introduced into our theory provided that 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 [1]. If query q is a function then we must also prove that:

$$R_q(x, r_1) \wedge R_q(x, r_2) \;\Rightarrow\; r_1 = r_2$$

As an example, suppose we have already defined a fragment of Peano arithmetic with constant "0", addition "+" and successor function $s \in \mathbb{N} \to \mathbb{N}$ with $\mathcal{A}$ containing axioms such as $s(x) = x + 1$, $s(x + y) = s(x) + y$ and $x = y \;\equiv\; s(x) = s(y)$. We would now like to introduce a predecessor (partial) function p as follows:

$p(x : \mathbb{N}) : \mathbb{N}$
    -- predecessor function
  require $x \neq 0$
  ensure $s(\mathit{Result}) = x$

Obviously $p(0)$ is undefined and so are expressions such as $p(0) \vee \neg p(0)$. The question is how to deal with such undefined expressions. Also, suppose

$$\begin{aligned} \phi_1 &\triangleq p(x + y) = p(x) + y \\ \phi_2 &\triangleq (x \neq 0) \;\Rightarrow\; (p(x + y) = p(x) + y) \end{aligned}$$

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 the former is undefined (if $x = 0$) and thus should not be provable.

Following the logic developed for Event-B [1], we inductively define the WD (well-definedness) operator $\mathcal{D}$ that maps formulas to their WD predicates. For a variable x we have that $\mathcal{D}(x) \triangleq \mathit{true}$. What about the well-definedness of a query $q(x)$? For a query, $\mathcal{D}(q(x)) \triangleq \mathcal{D}(x) \wedge C_q(x)$. 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 $\mathcal{D}(\mathcal{D}(\alpha)) \equiv \mathit{true}$, i.e. WD-predicates are themselves well-defined [1]. We introduce additional rules for the counting quantifier and maximums and minimums:

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

$$\mathcal{D}(x) \triangleq true \tag{2}$$

$$\mathcal{D}(q(x)) \triangleq \mathcal{D}(x) \wedge C_q(x) \tag{3}$$

$$\mathcal{D}(exp_1 = exp_2) \triangleq \mathcal{D}(exp_1) \wedge \mathcal{D}(exp_2) \tag{4}$$

$$\mathcal{D}(\neg P) \triangleq \mathcal{D}(P) \tag{5}$$

$$\begin{aligned}\mathcal{D}(P \wedge Q) \triangleq & \quad (\mathcal{D}(P) \wedge (P \Rightarrow \mathcal{D}(Q))) \\ & \vee (\mathcal{D}(Q) \wedge (Q \Rightarrow \mathcal{D}(P)))\end{aligned} \tag{6}$$

$$\begin{aligned}\mathcal{D}(P \Rightarrow Q) \triangleq & \quad (\mathcal{D}(P) \wedge (P \Rightarrow \mathcal{D}(Q))) \\ & \vee (\mathcal{D}(Q) \wedge (\neg Q \Rightarrow \mathcal{D}(P)))\end{aligned} \tag{7}$$

$$\begin{aligned}\mathcal{D}(P \vee Q) \triangleq & \quad (\mathcal{D}(P) \wedge (\neg P \Rightarrow \mathcal{D}(Q))) \\ & \vee (\mathcal{D}(Q) \wedge (\neg Q \Rightarrow \mathcal{D}(P)))\end{aligned} \tag{8}$$

$$\mathcal{D}(P \equiv Q) \triangleq \mathcal{D}(P) \wedge \mathcal{D}(Q) \tag{9}$$

$$\mathcal{D}(\forall x \bullet P) \triangleq \forall x \bullet \mathcal{D}(P) \vee (\exists x \bullet \mathcal{D}(P) \wedge \neg P) \tag{10}$$

$$\mathcal{D}(\exists x \bullet P) \triangleq (\forall x \bullet \mathcal{D}(P)) \vee (\exists x \bullet \mathcal{D}(P) \wedge P) \tag{11}$$

$$\begin{aligned}\mathcal{D}((\#i|\ p \leq i < q\ \wedge\ R(i) \bullet P(i))) \triangleq & \quad (\forall i \bullet \mathcal{D}(p \leq i < q\ \wedge\ R(i))) \\ & \wedge (\forall i \bullet p \leq i < q \wedge R(i)\ \Rightarrow\ \mathcal{D}(exp(i)))\end{aligned} \tag{12}$$

$$\begin{aligned}\mathcal{D}((\Sigma i|\ p \leq i < q\ \wedge\ R(i) \bullet exp(i))) \triangleq & \quad (\forall i \bullet \mathcal{D}(p \leq i < q\ \wedge\ R(i))) \\ & \wedge (\forall i \bullet p \leq i < q \wedge R(i)\ \Rightarrow\ \mathcal{D}(P(i)))\end{aligned} \tag{13}$$

$$\begin{aligned}\mathcal{D}((\Pi i|\ p \leq i < q\ \wedge\ R(i) \bullet exp(i))) \triangleq & \quad (\forall i \bullet \mathcal{D}(p \leq i < q\ \wedge\ R(i))) \\ & \wedge (\forall i \bullet p \leq i < q \wedge R(i)\ \Rightarrow\ \mathcal{D}(exp(i)))\end{aligned} \tag{14}$$

$$\begin{aligned}\mathcal{D}((\uparrow i|\ p \leq i < q\ \wedge\ R(i) \bullet exp(i))) \triangleq & \quad [\forall i \bullet \mathcal{D}(p \leq i < q\ \wedge\ R(i))] \\ & \wedge [\forall i \bullet\ p \leq i < q \wedge R(i)\ \Rightarrow\ \mathcal{D}(exp(i))] \\ & \wedge [\exists i \bullet\ p \leq i < q\ \wedge\ R(i)]\end{aligned} \tag{15}$$

$$\begin{aligned}\mathcal{D}((\downarrow i|\ p \leq i < q\ \wedge\ R(i) \bullet exp(i))) \triangleq & \quad [\forall i \bullet \mathcal{D}(p \leq i < q\ \wedge\ R(i))] \\ & \wedge [\forall i \bullet\ p \leq i < q \wedge R(i)\ \Rightarrow\ \mathcal{D}(exp(i))] \\ & \wedge [\exists i \bullet\ p \leq i < q\ \wedge\ R(i)]\end{aligned} \tag{16}$$

Intuitively, the above definitions enumerate possible conditions where a conjunction or implication in a predicate could be well-defined. Applying the rules to ϕ_1 we obtain:

Prove: $\mathcal{A} \vdash \mathcal{D}(\phi_1)$

$$
\begin{array}{ll}
 & \mathcal{D}(\phi_1) \\
= & \quad \langle \text{ Definition of } \phi_1 \ \rangle \\
 & \mathcal{D}(p(x+y) = p(x) + y) \\
= & \quad \langle \text{ using } D \text{ rule for equality } \rangle \\
 & \mathcal{D}(p(x+y)) \ \wedge \ \mathcal{D}(p(x) + y) \\
= & \quad \langle \text{ using } D \text{ rules for function application } \rangle \\
 & (\mathcal{D}(x+y) \ \wedge \ x + y \neq 0) \wedge (\mathcal{D}(x) \ \wedge \ x \neq 0 \ \wedge \ \mathcal{D}(y)) \\
= & \quad \langle \text{ Simplifying: } \mathcal{D}(x) = \mathcal{D}(y) = \mathit{true} = \mathcal{D}(x+y) \ \rangle \\
 & x + y \neq 0 \ \wedge \ x \neq 0 \\
= & \quad \langle \text{ Arithmetic and } x, y \in \mathbb{N} \ \rangle \\
 & x \neq 0
\end{array}
$$

So we are unable to prove that $\mathcal{D}(\phi_1)$ is a theorem. Hence predicate ϕ_1 does not pass the $\mathcal{D}$-filter. At this point we figure out that we need $x \neq 0$ either in the antecedent of ϕ_1 or as it appears in ϕ_2 (where $\phi_2 \triangleq x \neq 0 \Rightarrow \phi_1$). 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, now becomes:

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**:

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

where

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

Proof

Let $A_q \ \triangleq \ \mathcal{A} \wedge \textbf{QIA}$. Then

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P_5}\,\textsc{taut} \qquad \dfrac{\vdash_{\mathcal{D}} \mathcal{D}(A_q)}{P_3}\,\text{EQV}_{\mathcal{D}}}{\vdash \mathcal{D}(A_q)}\,\textsc{cut}}{P_0}\,\textsc{mon} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P_6}\,\textsc{taut}}{P_4}\,\textsc{mon} \quad \dfrac{A_q \vdash_{\mathcal{D}} \mathcal{D}(\beta_q)}{P_1}\,\text{EQV}_{\mathcal{D}}}{\mathcal{D}(A_q), A_q \vdash \mathcal{D}(\beta_q)}\,\textsc{cut} \quad \dfrac{A_q \vdash_{\mathcal{D}} \beta_q}{P_2}\,\text{EQV}_{\mathcal{D}}}{\mathcal{D}(A_q), A_q \vdash \beta_q}\,\textsc{cut}}{\mathcal{D}(A_q), \mathcal{D}(A_q) \Rightarrow A_q \vdash \beta_q}\,\text{MP}}{\mathcal{D}(A_q) \Rightarrow A_q \vdash \beta_q}\,\textsc{cut}$$

with

$$\begin{aligned}
P_0 &\triangleq \mathcal{D}(A_q) \Rightarrow A_q \vdash \mathcal{D}(A_q)\\
P_1 &\triangleq \mathcal{D}(A_q), \mathcal{D}(\mathcal{D}(\beta_q)), A_q \vdash \mathcal{D}(\beta_q)\\
P_2 &\triangleq \mathcal{D}(A_q), \mathcal{D}(\beta_q), A_q \vdash \beta_q\\
P_3 &\triangleq \mathcal{D}(\mathcal{D}(A_q)) \vdash \mathcal{D}(A_q)\\
P_4 &\triangleq \mathcal{D}(A_q), A_q \vdash \mathcal{D}(\mathcal{D}(\beta_q))\\
P_5 &\triangleq \ \vdash \mathcal{D}(\mathcal{D}(A_q))\\
P_6 &\triangleq \ \vdash \mathcal{D}(\mathcal{D}(\beta_q))
\end{aligned}$$

Q.E.D.

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

We can then reformulate the predicate logic rules in order to check the well-definedness of any new expressions that are introduced in a proof either through ∃-introduction (in the goal), ∀-introduction (in the hypothesis) or the cut rule [12, p46], e.g.

$$\begin{array}{c}\forall\textsc{-introduction in hypothesis}_{\mathcal{D}}\\ \dfrac{H \vdash_{\mathcal{D}} \mathcal{D}(e) \qquad H, P[x := e] \vdash_{\mathcal{D}} Q}{H, \forall x \bullet P \vdash_{\mathcal{D}} Q}\end{array} \qquad \begin{array}{c}\exists\textsc{-introduction in goal}_{\mathcal{D}}\\ \dfrac{H \vdash_{\mathcal{D}} \mathcal{D}(e) \qquad H \vdash_{\mathcal{D}} P[x := e]}{H \vdash_{\mathcal{D}} \exists x \bullet P}\end{array}$$

$$\begin{array}{c}\textsc{cut}_{\mathcal{D}}\\ \dfrac{H \vdash_{\mathcal{D}} \mathcal{D}(G) \qquad H \vdash_{\mathcal{D}} G \qquad G, H \vdash_{\mathcal{D}} P}{H \vdash_{\mathcal{D}} P}\end{array}$$

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 [12]. We supplement the usual predicate logic, with the inference rules of equational logic [3, 18] and prove the validity of the three new inference rules of equational logic:

$$\textsc{Equanimity}_{\mathcal{D}}\quad \dfrac{H \vdash_{\mathcal{D}} \mathcal{D}(P) \qquad H \vdash_{\mathcal{D}} P \qquad H \vdash_{\mathcal{D}} P \equiv Q}{H \vdash_{\mathcal{D}} Q} \qquad \textsc{Leibniz}_{\mathcal{D}}\quad \dfrac{H \vdash_{\mathcal{D}} \mathcal{D}(P \equiv Q) \qquad H \vdash_{\mathcal{D}} P \equiv Q}{H \;\; \vdash_{\mathcal{D}} \;\; \alpha[x := P] \;\equiv\; \alpha[x := Q]}$$

$$\textsc{Transitivity}_{\mathcal{D}}\quad \dfrac{H \vdash_{\mathcal{D}} \mathcal{D}(Q) \qquad H \vdash_{\mathcal{D}} P \equiv Q \qquad H \vdash_{\mathcal{D}} Q \equiv R}{H \vdash_{\mathcal{D}} P \equiv R}$$

Proof of Leibniz$_{\mathcal{D}}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{D}(H), H \vdash \mathcal{D}(\mathcal{D}(P \equiv Q))}\,\textsc{taut} \qquad \dfrac{H \vdash_{\mathcal{D}} \mathcal{D}(P \equiv Q)}{P_0}\,\textsc{eqv}_{\mathcal{D}}}{\mathcal{D}(H), H \vdash \mathcal{D}(P \equiv Q)}\,\textsc{cut} \qquad \dfrac{H \vdash_{\mathcal{D}} P \equiv Q}{P_1}\,\textsc{eqv}_{\mathcal{D}}}{\mathcal{D}(H), H \vdash P \equiv Q}\,\textsc{cut}}{\mathcal{D}(H), H \;\vdash\; \alpha[x := P] \;\equiv\; \alpha[x := Q]}\,\textsc{Leibnz}}{\mathcal{D}(H),\; \mathcal{D}(\alpha[x := P] \equiv \alpha[x := Q]),\; H \;\; \vdash \;\; \alpha[x := P] \;\equiv\; \alpha[x := Q]}\,\textsc{mon}}{H \vdash_{\mathcal{D}} \alpha[x := P] \;\equiv\; \alpha[x := Q]}\,\textsc{eqv}_{\mathcal{D}}$$

with

$$\begin{aligned} P_0 &\triangleq \mathcal{D}(H), \mathcal{D}(\mathcal{D}(P \equiv Q)), H \vdash \mathcal{D}(P \equiv Q) \\ P_1 &\triangleq \mathcal{D}(H), \mathcal{D}(P \equiv Q), H \vdash P \equiv Q \end{aligned}$$

When using Leibniz's rule with $\vdash_{\mathcal{D}}$, it is important to prove $\mathcal{D}(P \equiv Q)$ even though $\alpha[x := P] \;\equiv\; \alpha[x := Q]$ is well-defined. This is because the well-definedness of P and Q in α might depend on the surrounding terms. For example, with:

$$\alpha \;\triangleq\; x \wedge R$$

It is possible to prove $\mathcal{D}(\alpha[x := P])$ by proving $\mathcal{D}(R) \wedge (R \Rightarrow \mathcal{D}(P))$ whereas $\mathcal{D}(P)$ does not hold on its own, which makes it necessary to provide $\mathcal{D}(P \equiv Q)$ as a premise.

In the case of the predecessor p query, we can prove its well-definedness and feasibility as follows:

WD of Feasibility (simplified): $\mathcal{D}(x \neq 0) \;\wedge\; (x \neq 0 \Rightarrow \mathcal{D}(\exists r \bullet\; s(r) = x))$

We prove the two conjuncts individually.

$$\begin{aligned}
& \mathcal{D}(x \neq 0)\\
= & \quad \langle\ \mathcal{D} \text{ over } \neg \text{ and } =\ \rangle\\
& \mathcal{D}(x) \wedge \mathcal{D}(0)\\
= & \quad \langle\ \mathcal{D} \text{ applied to atomic formulae}\ \rangle\\
& \mathit{true} \wedge \mathit{true}\\
= & \quad \langle\ \text{identity of } \wedge\ \rangle\\
& \mathit{true}
\end{aligned}$$

$$\begin{aligned}
& \mathcal{D}(\exists r \bullet s(r) = x)\\
\Leftarrow & \quad \langle\ \mathcal{D} \text{ over } \exists\ \rangle\\
& \forall r \bullet \mathcal{D}(s(r) = x)\\
= & \quad \langle\ \text{see Lemma: } \mathcal{D}(s(r) = x) \equiv \mathit{true}\ \rangle\\
& \forall r \bullet \mathit{true}\\
= & \quad \langle\ r \text{ not free in } \mathit{true} \rangle\\
& \mathit{true}
\end{aligned}$$

Lemma:

$$\begin{aligned}
& \mathcal{D}(s(r) = x)\\
= & \quad \langle\ \mathcal{D} \text{ over } =\ \rangle\\
& \mathcal{D}(s(r)) \wedge \mathcal{D}(x)\\
= & \quad \langle\ \mathcal{D} \text{ over function application}\ \rangle\\
& C_s(r) \wedge \mathcal{D}(r) \wedge \mathcal{D}(x)\\
= & \quad \langle\ \text{precondition of } s;\ \mathcal{D} \text{ over atomic formulae}\ \rangle\\
& \mathit{true} \wedge \mathit{true} \wedge \mathit{true}\\
= & \quad \langle\ \text{identity of } \wedge\ \rangle\\
& \mathit{true}
\end{aligned}$$

Feasibility of p: $x \neq 0 \Rightarrow \exists r \bullet s(r) = x$.
We prove it by induction over x. In the base case ($x = 0$), the antecedent of the implication becomes $0 \neq 0$ which makes the formula true. All that is left is the induction step (assuming it holds for x, we prove it for $s(x)$):
Proof of Induction step:

$$\begin{aligned}
& s(x) \neq 0 \Rightarrow \exists r \bullet s(r) = s(x)\\
\Leftarrow & \quad \langle\ \text{instantiation with } r := x\ \rangle\\
& s(x) \neq 0 \Rightarrow s(x) = s(x)\\
= & \quad \langle\ \text{reflexivity of } = \rangle\\
& s(x) \neq 0 \Rightarrow \mathit{true}\\
= & \quad \langle\ \text{right zero of } \Rightarrow\ \rangle\\
& \mathit{true}
\end{aligned}$$

In the above proof we did not need the induction hypothesis $x \neq 0 \Rightarrow \exists r \bullet s(r) = x$.

When the query postcondition is in closed form "$\mathit{Result} = f(x)$" 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:

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

Proof: If $R_q(x, r)$ is in closed form, then query q is feasible:

Assume: $R_q(x, r) \triangleq r = f_q(x)$, i.e. $R_q(x, r)$ is in closed form

$$
\begin{array}{ll}
 & \exists r \ \bullet \ R(x,r) \\
= & \quad \langle \text{ Assumption: } R(x,r) \text{ is in closed form } \rangle \\
 & \exists r \ \bullet \ (r = f(x)) \\
= & \quad \langle \text{ one-point rule } \rangle \\
 & \mathit{true}
\end{array}
$$

WD of feasibility: $\mathcal{D}(C_q(x)) \ \wedge \ (C_q(x) \Rightarrow \forall r \ \bullet \ \mathcal{D}(R_q(x,r)))$

The only difference between the above formula and **CFF** is in the substitution of $\mathcal{D}(q(x))$ for $\forall r \ \bullet \ \mathcal{D}(R_q(x,r))$. We will therefore write our proof by strengthening the latter into the former.

Assume: $R_q(x,r) \triangleq r = f_q(x)$, i.e. $R_q(x,r)$ is in closed form

$$
\begin{array}{ll}
 & \forall r \bullet \mathcal{D}(R_q(x,r)) \\
= & \quad \langle \text{ Assumption: } R(x,r) \text{ is in closed form } \rangle \\
 & \forall r \bullet \mathcal{D}(r = q(x)) \\
= & \quad \langle \ \mathcal{D} \text{ over } = \ \rangle \\
 & \forall r \bullet \mathcal{D}(r) \wedge \mathcal{D}(q(x)) \\
= & \quad \langle \ \mathcal{D} \text{ over atomic formulae ; identity of } \wedge \ \rangle \\
 & \forall r \bullet \mathcal{D}(q(x)) \\
\Leftarrow & \quad \langle \ r \text{ is not free in } \mathcal{D}(q(x)) \ \rangle \\
 & \mathcal{D}(q(x))
\end{array}
$$

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. Only on formulas such as ϕ_2 which pass the filter do we then go on to the validity proof.

Prove: $\mathcal{A}, \textbf{QIA} \vdash_{\mathcal{D}} \phi_2$

Assume: $x \neq 0$

$$
\begin{array}{ll}
 & (p(x+y) = p(x) + y) \\
= & \quad \langle \text{ Let } r1 = p(x+y) \text{ and } r_2 = p(x) \ \rangle \\
 & (r1 = r2 + y) \\
= & \quad \langle \text{ From } \mathcal{A}\text{: } a = b \equiv s(a) = s(b) \ \rangle \\
 & s(r1) = s(r2 + y) \\
= & \quad \langle \text{ From } \mathcal{A}\text{: } s(a+b) = s(a) + b \ \rangle \\
 & s(r1) = s(r2) + y \\
= & \quad \langle \text{ By QIA } s(r1) = x + y \text{ and } s(r2) = x \ \rangle \\
 & x + y = x + y \\
= & \quad \langle \text{ reflexivity of equality } \rangle \\
 & \mathit{true}
\end{array}
$$

This proof justifies 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

$x \neq 0$ is not actually used in the validity proof. It is needed only to ensure that we pass the WD proof obligation.

10 Well-definedness of tabular expressions

We can now also justify the well-definedness condition (d) of the tabular expression in Fig. 5 (using the meaning definition (a) of a tabular expression).

Prove: (d) $\Rightarrow \mathcal{D}((\text{a}))$

$$
\begin{array}{ll}
 & \mathcal{D}(\forall i \in 1\,..\,3 \;\bullet\; P_i \Rightarrow Q_i) \\
\Leftarrow & \quad \langle \text{ using the } \mathcal{D} \text{ definition for } \forall \;\rangle \\
 & \forall i \in 1\,..\,3 \;\bullet\; \mathcal{D}(P_i \Rightarrow Q_i) \\
\Leftarrow & \quad \langle \text{ Definition of } \mathcal{D} \text{ for } \Rightarrow \;\rangle \\
 & \forall i \in 1\,..\,3 \;\bullet\; \mathcal{D}(P_i) \wedge (P_i \Rightarrow \mathcal{D}(Q_i))
\end{array}
$$

As an example, consider the introduction of query *ymax* in the signal module (Fig. 6) and assume that we have already demonstrated the feasibility of query $swf(i)$ (which must be done as part of the development of the theory for sequences SEQ[G]). Here is the proof that the query *ymax* is feasible:

Prove: $vf \wedge s3 \Rightarrow [\exists r \;\bullet\; r = (\uparrow i | 1 \leq i \leq n \bullet swf(i))]$

Assume: $vf \wedge s3$

$$
\begin{array}{ll}
 & \exists r \;\bullet\; r = (\uparrow i | 1 \leq i \leq n \bullet swf(i)) \\
= & \quad \langle \; ymax \text{ is closed form, thus by CFF (closed form feasibility) } \rangle \\
 & \mathcal{D}(\uparrow i | 1 \leq i \leq n \bullet swf(i)) \\
= & \quad \langle \text{ Definition of } \mathcal{D} \text{ for } \uparrow \;\rangle \\
 & [\exists i \bullet 1 \leq i \leq n] \;\wedge\; [\forall i \mid 1 \leq i \leq n \bullet \mathcal{D}(swf(i))] \\
= & \quad \langle \text{ By assumption } s3 \text{ and } s3 \equiv n \geq 3 \text{ we have that } (\exists i \bullet 1 \leq i \leq n) \text{ holds } \rangle \\
 & \forall i \mid 1 \leq i \leq n \bullet \mathcal{D}(swf(i)) \\
= & \quad \langle \; \mathcal{D}(swf(i)) \;\equiv\; \mathcal{D}(i) \wedge 1 \leq i \leq n \text{ and } \mathcal{D}(i) \equiv true \;\rangle \\
 & \forall i \mid 1 \leq i \leq n \;\bullet\; 1 \leq i \leq n \\
= & \quad \langle \text{ trading and reflexivity of } \Rightarrow \;\rangle \\
 & true
\end{array}
$$

Now consider the first row of the tabular expression in Table 1. The condition predicate for that row is $ok \triangleq vf \wedge s3 \wedge um \wedge t50? \wedge t10?$. We would like to demonstrate the well-definedness of this condition. We can work incrementally by first showing the well-definedness of $vf \wedge s3$ and then exploiting this to establish $\mathcal{D}(ok)$.

$$\begin{array}{ll}
 & \mathcal{D}(vf \wedge s3) \\
\Leftarrow & \quad \langle \text{ definition of } \mathcal{D} \text{ for conjunction } \rangle \\
 & \mathcal{D}(vf) \wedge (vf \Rightarrow \mathcal{D}(s3)) \\
= & \quad \langle\ \mathcal{D}(vf) = \mathit{true}\text{: any string sequences can be checked for conversion to floats } \rangle \\
 & vf \Rightarrow \mathcal{D}(s3) \\
= & \quad \langle\ \mathcal{D}(s3) = vf\text{, i.e. the precondition of } s3 \text{ in module } \textbf{p-PULSE} \text{ Fig 6 } \rangle \\
 & vf \Rightarrow vf \\
= & \quad \langle \text{ reflexivity of } \Rightarrow \rangle \\
 & \mathit{true}
\end{array}$$

Finally, we consider the proof in Figure 8. Since it is done in our equational logic for partial functions, we need to consider the well-definedness conditions raised by the repeated application of transitivity. Ignoring the steps where no partial functions are used (which are always well-defined), we come up with the following proof obligations:

- $\mathcal{D}(wf.last(1, t50p, y10) < t50p)$
- $\mathcal{D}((\uparrow t : \mathbb{F} | 1 \leq t \leq t50p \wedge wf(t) = y10 \bullet t) < t50p)$
- $\mathcal{D}((\forall t : \mathbb{F} | 1 \leq t \leq t50p \wedge wf(t) = y10 \bullet t < t50p))$
- $\mathcal{D}((\forall t : \mathbb{F} | 1 \leq t \wedge t \leq t50p \wedge t50p \leq t \bullet wf(t) \neq y10))$
- $\mathcal{D}((\forall t : \mathbb{F} | t = t50p \bullet wf(t) \neq y10))$
- $\mathcal{D}(wf(t50p) \neq y10)$

As an example, here is how we can prove the first one:

$$\begin{array}{ll}
 & \mathcal{D}(wf.last(1, t50p, y10) < t50p) \\
= & \quad \langle \text{ Definition of } \mathcal{D} \ \rangle \\
 & wf.has(1, t50p, y10) \\
= & \quad \langle \text{ Definition of } t10p? \ \rangle \\
 & t10p? \\
\Leftarrow & \quad \langle \text{ Definition of } durationp? \ \rangle \\
 & durationp?
\end{array}$$

Listing 1: Executable `PULSE` Specification

```
class PULSE feature
    sampled_signal: SIGNAL -- sampled input
    w: WAVEFORM -- waveform as a real function of levels vs. instants
    duration: FLOAT -- pulse duration
        require t50_ok
        ensure Result = (trans2.t50.value − trans1.t50.value)
    t10_ok: BOOLEAN -- do both 50% instants exist?
        ensure Result = (trans1.t10.ok and trans2.t10.ok)
    t50_ok: BOOLEAN -- do both 50% instants exist
        ensure
            s3 and um implies (Result = (trans1.t50.ok and trans2.t50.ok))
            not (s3 and um) implies not Result
    trans1: POSITIVE_TRANSITION
    trans2: NEGATIVE_TRANSITION
    error: BOOLEAN -- no parameters can be calculated
        ensure Result = not (s3 and then um and then t50_ok)
    warning: BOOLEAN -- 10% instants cannot be calculated
        ensure Result = (not error and then not t10_ok)
    ok: BOOLEAN -- all parameters can be calculated
        ensure Result = (not error and then not warning and then t10_ok)
    s3: BOOLEAN -- are there at least 3 samples?
        ensure Result = sampled_signal.s3
     um: BOOLEAN -- Is there a unique maximum
        ensure Result = sampled_signal.um
invariant
    completeness: error or else warning or else ok
    disjointness: not (error and warning) and
        not (error and ok) and not (warning and ok)
    case_ok: ok implies trans1.t10.ok and trans1.t50.ok and
        trans1.t90.ok and trans1.duration.ok and
        trans2.t10.ok and trans2.t50.ok and
        trans2.t90.ok and trans2.duration.ok
    req9: ok implies trans1.t10.value < trans1.t50.value and
        trans1.t50.value < trans1.t90.value and
        trans2.t10.value > trans2.t50.value and
        trans2.t50.value > trans2.t90.value end
end-- class PULSE
```

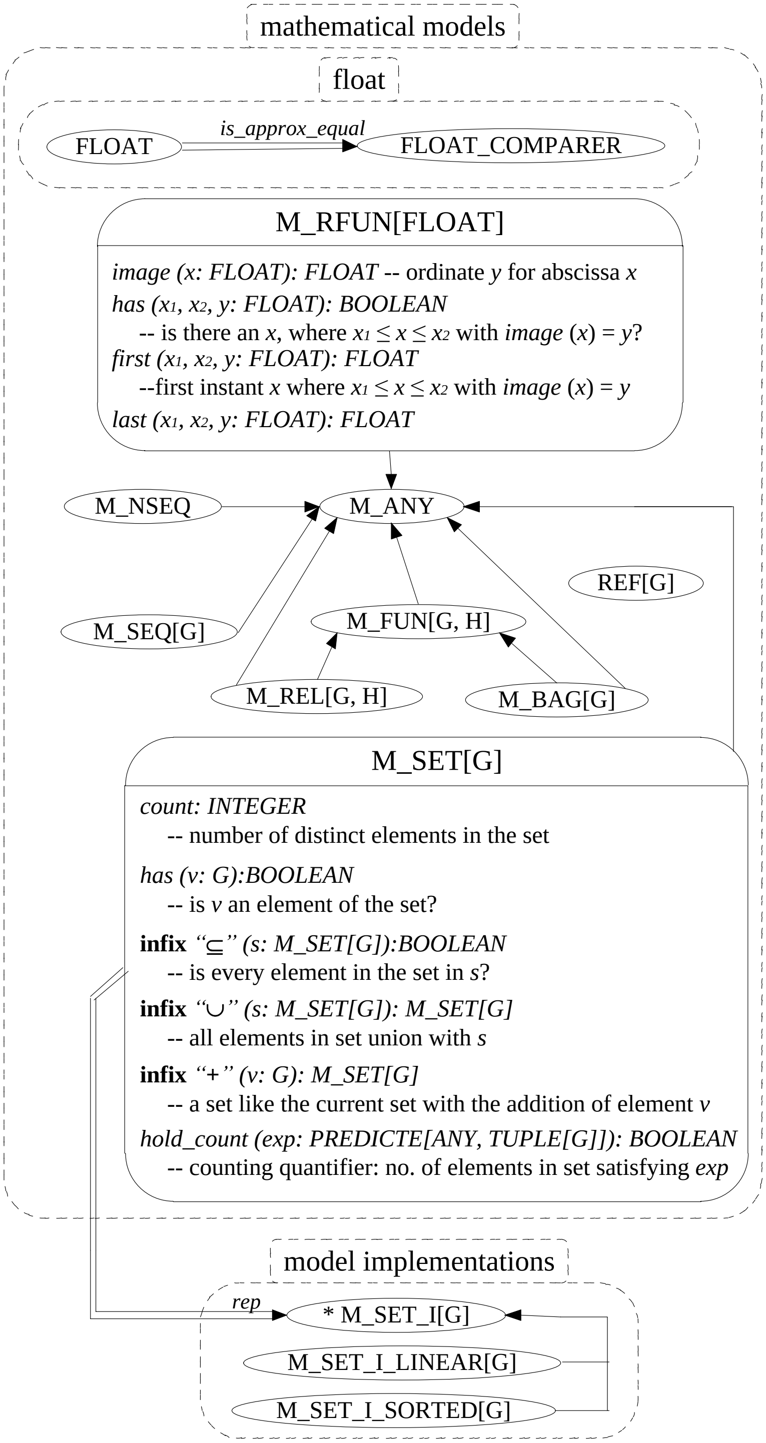
mathematical models
float
FLOAT
is_approx_equal
FLOAT_COMPARER
M_RFUN[FLOAT]
image (x: FLOAT): FLOAT -- ordinate y for abscissa x
has (x1, x2, y: FLOAT): BOOLEAN
-- is there an x, where x1 ≤ x ≤ x2 with image (x) = y?
first (x1, x2, y: FLOAT): FLOAT
--first instant x where x1 ≤ x ≤ x2 with image (x) = y
last (x1, x2, y: FLOAT): FLOAT
M_NSEQ
M_ANY
REF[G]
M_SEQ[G]
M_FUN[G, H]
M_REL[G, H]
M_BAG[G]
M_SET[G]
count: INTEGER
-- number of distinct elements in the set
has (v: G):BOOLEAN
-- is v an element of the set?
infix "⊆" (s: M_SET[G]):BOOLEAN
-- is every element in the set in s?
infix "∪" (s: M_SET[G]): M_SET[G]
-- all elements in set union with s
infix "+" (v: G): M_SET[G]
-- a set like the current set with the addition of element v
hold_count (exp: PREDICTE[ANY, TUPLE[G]]): BOOLEAN
-- counting quantifier: no. of elements in set satisfying exp
model implementations
rep
* M_SET_I[G]
M_SET_I_LINEAR[G]
M_SET_I_SORTED[G]

Figure 9: Design of Mathematical Specification Library

11 MSL – executable specifications

In the previous sections of this paper we provided a method for developing a precise requirements document (RD) using model contracts and tabular expressions. Before any code is developed, executable specifications using MSL (see below) do automatic type checking and are used to check for completeness, disjointness and well-definedness of the tabular expressions. They are also used to check the consistency of global properties (such as REQ9). While developing production code, executable specifications are used to check that the implementations satisfy the specification.

The original pulse software was only a few hundred lines of code. However, many warnings and error checks were omitted and there were ambiguities in the requirements. Thus more code will be needed than originally envisaged and the calculation of parameters will have to be changed. The need to deal with floating point arithmetic is also challenging. Thus where the highest assurance is not required, the use of executable specifications (in addition to testing) may provide sufficient evidence for certification, depending on the application. We describe below the use of MSL (model specification language) for executable specifications. The use of model contracts in the RD make the transition to MSL relatively seamless.

As described in [15], the BON class diagram for MSL is shown in Fig. 9. MSL collection classes for sets, functions, relations, bags, sequences etc. are immutable (using Eiffel's expanded construct) and void safe and thus have a specification-friendly value semantics. We have subsequently added to MSL appropriate machinery for dealing with floating point arithmetic including the ability to do approximate comparisons of two floating point values. The model-based specification in Fig. 6 relies on the notion of real-valued functions RFUN (Fig. 7) which is the abstraction at the heart of the mathematical model. MSL's version is M_RFUN[FLOAT] as shown in Fig. 9.

MSL can be used for validation and verification as follows. The model-based specification of Fig. 6 is converted to MSL. In Eiffel, classes are used for types and modules. So the MSL pulse specification has class names and features matching the abstract specification. Strong type checking via the compiler ensures that all the signatures and contracts are consistent. A fragment of class SIGNAL is:

```
class SIGNAL feature
  swf: M_NSEQ −− input sampled waveform as a sequence of floats
  s3: BOOLEAN −− are there at least 3 samples?
    ensure Result = (swf.count >= 3)
  ymax: FLOAT −− maximum level
    require s3 and um
    ensure Result = swf.maximum
  um: BOOLEAN −− is there a unique maximum?
    ensure Result = (swf.hold_count (agent swf [i] = swf.maximum) = 1)
```

Query *ymax* is obtained via the counting quantifier *hold_count* using Eiffel's agent mechanism for quantifiers. In class WAVEFORM we declare the real function of levels versus instants as *wf*: *M_RFUN*. This allows us to define instants with contracts, e.g.

The queries $t50p?$ and $t50p$ in the abstract model contracts (Fig. 6) are represented by a tuple in the executable specification. Executing the specification results in the contracts

```
wf: M_RFUN -- real function wf ∈ 𝔽 ⇸ 𝔽
t50p: TUPLE[ok:BOOLEAN;value:FLOAT]
    ensure
        Result.ok = wf.has (1,tmax,w.y50)
        Result.ok implies (Result.value = wf.first (1, tmax,w.y50))
    end
```

being checked. In addition, properties can be checked by declaring them as class invariants as shown in Listing 1. Invariants are used to check the completeness, disjointness and well-definedness of the tabular expression. They also check global properties such as REQ9. Once the code is produced, the code can be checked against the MSL specification, again via runtime assertion checking.

12 Conclusion

One way to obtain a precise requirements document is to include a context diagram describing the environment of the system under design and atomic E/R descriptions (that capture customer goals in informal English). The E-descriptions provide a method to document assumptions, constraints and business rules that come from the environment and that impact on the requirements.

A formal specification is needed to validate the requirements. A useful specification consists of model contracts and tabular expressions and a calculus for reasoning about the specification. The specification can be translated into an executable MSL specification. Beyond testing we have shown how to use proofs or runtime assertion checking to validate the requirements via completeness, disjointness, well-definedness and consistency checks against the atomic descriptions. Once we have some assurance that the requirements are precise and consistent, we can develop production code and verify the code against the specification. Certification of certain safety critical systems would suggest proofs, but other mission critical systems systems might admit the additional assurance provided by executable specifications.

The pulse code provided by our industrial partner illustrates some of the advantages of our approach. As mentioned earlier, if one submits input signals such as $\langle 0, 1, 1\rangle$ to their code, no error is returned and the parameters are thus calculated. The problem is that the pulse duration and some of the parameters are NaN (not a number) due to division by zero and other issues. There was no check done for completeness, disjointness and well-definedness. In order to achieve completeness (and full error checking) we need to first clarify the difference between valid and invalid pulses. The IEEE-181 standard was not clear on this. Our first job was to clarify these differences in atomic E-statements (e.g. see ENV1, ENV2) and then to specify when paremeters could be calculated and when errors or warnings must be raised in the atomic R-statements. From this analysis it emerged that we need three or more sample points, a unique maximum and a variety of other properties that the signal $\langle 0, 1, 1\rangle$ does not satisfy. We used the E/R-statements to construct a tabular expression (Table 1) based on model contracts (Fig. 6), from which completeness and the consistency of global properties can be checked. The changes at the requirements level indicate that the code would have to be significantly different than what was originally provided.

Of course, much more will be needed in the RD. Hazards must be analyzed and acceptance tests must be provided. But even testing would be incomplete without a precise documentation of the requirements. The rows in tabular expressions and the atomic descriptions of global properties suggest important acceptance test scenarios.

The methods of this paper are applicable to transformational systems or reactive systems in which it is sufficient to check one step transition functions [9]. In future work, we hope to show how to extend the methods of this paper to more general reactive systems.

References

[1] Jean-Raymond Abrial and Louis Mussat. On using conditional definitions in formal theories. In *ZB2002 Formal Specification and Development in Z and B*, LNCS 2272. Springer-Verlag, 2002.

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

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

[4] 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.

[5] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical report, Microsoft, 2010.

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

[7] 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.

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

[9] 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.

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

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

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

[13] Bertrand Meyer. Domain theory: the forgotten step in program verification. http://bertrandmeyer.com, April 2012.

[14] 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.

[15] Jonathan S. Ostroff and Faraz Ahmadi Torshizi. Testable Requirements and Specifications. In Bertrand Meyer and Yuri Gurevich, editors, *Tests and Proofs (TAP'07)*, volume LNCS 4454. Springer Verlag, 2007.

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

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

[18] George Tourlakis. On the Soundness and Completeness of Equational Predicate Logics. Technical report, Toronto, 1998. On the Soundness and Completeness of Equational Predicate Logics.

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