

Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 00-024

Representing the Unknown in Specification Languages

Michael W. Whalen and Mats P. Heimdahl

March 30, 2000

Representing the Unknown in Specification Languages *

Michael W. Whalen
Department of Computer Science and
Engineering
University of Minnesota
4-192 EE/CS Building
Minneapolis, MN 55455
(612) 625-1381
whalen@cs.umn.edu

Mats P.E. Heimdahl
Department of Computer Science and
Engineering
University of Minnesota
4-192 EE/CS Building
Minneapolis, MN 55455
(612) 625-2068
heimdahl@cs.umn.edu

ABSTRACT

During the operation of software-controlled physical system, there are times when the values of environmental variables are not known by the control software. To correctly specify and reason about such systems, a specification language must allow variables to take a special *undefined* value that signifies that the value of the variable is unknown. Adding an undefined value to the type system of a language, however, complicates the semantics of the language because it causes many of the arithmetic operators to become partial functions. In this paper we discuss different approaches to managing undefined values and present our approach for the specification language RSML^{-e}. We provide a loose semantics that allows simulation/execution of incomplete models, and a tight semantics, which, given a completed model, is used for code-generation and static analysis. To prevent misuse of undefined values, we present a test that ensures that predicates in RSML^{-e} cannot evaluate to undefined, and that variables cannot implicitly take on undefined values.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features; D.2.1 [Software Engineering]: Requirements/Specifications—*Languages*; F.3.1 [Theory of Computation]: Semantics of Programming Languages

General Terms

Partial functions, undefined values, formal specifications

1. INTRODUCTION

Executable specification languages, such as Statecharts [6], SCR [11, 10], RSML [16], and SpecTRM-RL [17], are becoming more widely used in the specification of safety-critical

*This work has been partially supported by NSF grants CCR-9624324 and CCR-9615088

reactive systems. A particular advantage of these methods is that they are based on easy to read, graphical and tabular notations, and that they are designed to be understood by people without extensive training in computer science and formal methods. The formalisms are supported with static analysis procedures and they are executable, so that specifications can be tested and explored before they are complete.

For these specification languages to be effective, they must provide a rich enough semantics that the issues in the system to be controlled can be easily expressed in the language. Since the languages are intended to model systems in unpredictable environments, a specification language must include the ability to specify situations where the values of environmental variables may be unknown. In these situations, such as system startup, or when a sensor or actuator fails, any actual value assigned to the model variables would be a misrepresentation — the variables should assume a special undefined value. The inclusion of an undefined value does, however, complicate the semantics of a specification language. In this paper we provide an overview of approaches to model undefined values and present how we handle the problem in our fully formal specification language RSML^{-e}.

In the early nineties, Nancy Leveson led an effort to model the behavior of a complex avionics system, TCAS II, using the RSML [16] language. Based on our experience from this effort and subsequent projects, we have iteratively improved the language by simplifying and finalizing the semantics. Our work has resulted in the development of two languages: RSML^{-e} (Requirements State Machine Language without events) [21] developed by Mats P.E. Heimdahl *et al.* at the University of Minnesota, and SpecTRM-RL (SpecTRM Requirements Language) [17] developed by Nancy Leveson *et al.* at MIT, that share a common semantics (although they have significant syntactic differences). One of the areas that Nancy identified as needing improvement was the ability to specify that variables may take on unknown values. Leveson argues that *every* variable shall be treated as having an unknown value until a reliable measure of its value can be established [17]. This report discusses the semantic choices we made to support undefined values in RSML^{-e}.

When defining the semantics for undefined values in RSML^{-e}, we were constrained by the goals of the language.

- The semantics of undefined must be intuitive and easy to understand.
- An analyst must be able to execute and dynamically evaluate specifications early on, even when the specifications are incomplete, inconsistent, and contain many undefined values and partial functions.
- The semantics must lend themselves to static analysis by standard tools such as model checkers and theorem provers.
- A specification must be easy to implement in a standard imperative programming language (possibly through correctness preserving code generation).

These requirements have defined the constraints on our treatment of undefined variables and expressions.

In the next section, we will provide a short description of the RSML^{-e} language. In Section 3, we will describe different uses for undefined values in specifications. In Section 4, we describe the different ways in which undefined values are used in RSML^{-e}. Adding an undefined value to numeric types causes all of the standard arithmetic operators to become partial functions. Therefore, we examine approaches for formalizing partial functions in existing formal languages in Section 5, and describe our approach for RSML^{-e} in Section 6. Finally, we describe an algorithm for ensuring that undefined values are used correctly within RSML^{-e} specifications in Section 7.

2. OVERVIEW OF RSML^{-e}

RSML^{-e} is based on the the RSML language developed by the Irvine Safety Research group [16]. RSML^{-e} was developed as a requirements specification language specifically for embedded control systems. RSML^{-e} is based on hierarchical finite state machines and dataflow languages. Visually, it is somewhat similar to David Harel's Statecharts [8, 6, 7]. For example, RSML^{-e} supports parallelism, hierarchies, and guarded transitions. The main differences between RSML^{-e} and its precursor RSML are the addition in RSML^{-e} of rigorous specifications of the interfaces between the environment and the control software, and the removal of internal broadcast events. The removal of events was prompted by Nancy Leveson's experiences with with RSML and her new language SpecTRM-RL that she has evolved from RSML. These experiences have been chronicled in [17].

An RSML^{-e} specification consists of a collection of *input variables*, *state variables*, *input interfaces*, *output interfaces*, *functions*, *macros*, and *constants*, which will be briefly discussed below.

In RSML^{-e}, the state of the model is the set of assignment histories of all *variables* and *interfaces*. The state information is used to compute the values of a set of *state variables*, similar to mode classes in SCR [10]. These state variables can be organized in parallel or hierarchically to describe the current state of the system. Parallel state variables are used to represent the inherently parallel or concurrent concepts in the system being modeled. Hierarchical relationships allow *child* state variables to present an elaboration of a particular *parent* state value. Hierarchical state variables allow a specification designer to work at multiple levels of abstraction, and make models simpler to understand.

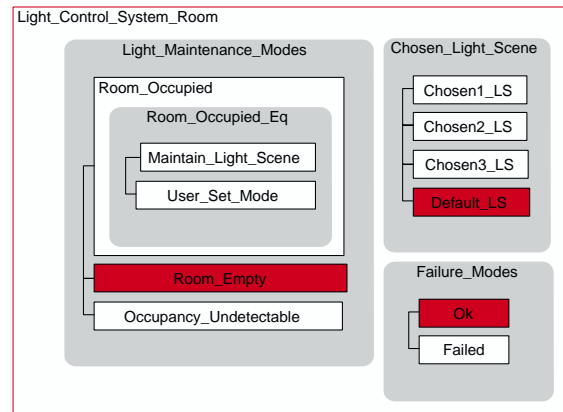


Figure 1: The state machine for the requirements model of a computerized Light Control System in room

For example, consider a software system to control the lights for a room. The state variable hierarchy used to model the requirements on this system could be represented as in Figure 1. This representation includes both parallel and hierarchical relationships of state variables. *Chosen_Light_Scene*, *Light_Maintenance_Modes* and *Failure_Modes* are three parallel state variables, and *Room_Occupied_Eq* is a child state variable of *Light_Maintenance_Modes*

Assignment functions in RSML^{-e} determine the value of state variables. These functions can be organized as *transitions* or *condition tables*. Condition tables describe under what condition a state variables *assumes* each of its possible values. Transitions describe the condition under which a state variable is to *change* value. A transition consists of a source value, a destination value, and a guarding condition. A transition is taken (causing a state variable to change value) when (1) the state variable value is equal to the source value, and (2) the guarding condition evaluates to true. The two assignment function types are logically equivalent; mechanized procedures exist to ensure that both functions are complete and consistent [9].

The state functions are placed into a partial order based on data dependencies and the hierarchical structure of the state machine. State variables are data-dependent on any other state variables, macros, or input variables that are named in their transitions or condition tables. If a variable is a child variable of another state variable, then it is also dependent on its parent variable. The value of the state variable can be computed after the items on which it is data-dependent have been computed. For example, the value of the *Room_Occupied_Eq* state variable would be computed after the *Light_Maintenance_Modes* state variable, because its value is dependent on whether or not *Light_Maintenance_Modes* is in the *Room_Occupied* state.

Conditions are simply predicate logic statement over the various states and variables in the specification. The conditions are expressed in disjunctive normal form using a notation called AND/OR tables [16] (see Figure 2). The far-left column of the AND/OR table lists the logical phrases. Each

of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated columns. An asterisk denotes “don’t care.”

:= 0 IF

..Light_Maintenance_Modes IN_STATE Room_Empty	T	T
TIME >= ..Light_Maintenance_Modes TIME_ENTERED Room_Empty + T3_InVar	T	*
MESSAGE_AT(FacM_Shutoff)	*	T

Figure 2: An example condition

Input variables in the specification allow the analyst to record the the values reported by various external sensors. They are assigned based on the messages received by *input interfaces*.

To further increase the readability of the specification, RSML^{-e} contains many other syntactic conventions. For example, they allow expressions used in the predicates to be defined as functions (e.g., TotalIntensity()), and familiar and frequently used conditions to be defined as macros (e.g., OccupancyUndetectable()). *Functions* in RSML^{-e} are mathematical functions that are used to abstract complex calculations. A *macro* is simply a named AND/OR table that is used for frequently repeated conditions and is defined in a separate section of the document.

3. USING UNDEFINED VALUES TO MODEL UNCERTAINTY

To accurately model the state of a real-world system, we must consider the circumstances when a portion of the system has failed or is in an unknown state. When modeling an input variable from the environment, there are several circumstances in which the true value of the variable is unknown. For example, consider the following:

- At system startup, the software system has not yet received the current status of the input variable.
- At system shutdown, the value of a sensor reading may no longer be trustworthy.
- The sensor recording the input variable has failed.
- The system variable that the sensor is monitoring is outside the range of the sensor.
- The value of the input is *stale*, i.e., the time period since the data was received is greater than some maximum allowable period and the data is outdated.

To represent these scenarios within a formal model, we need some way to represent that the input variable no longer has a valid value. In several specification languages, including SpecTRM-RL [17], RSML^{-e} [21], and VDM [3], this can be accomplished by assigning the variable a special undefined value (*nil* in VDM). This value can always be used in equality, inequality, and type membership predicates. How undefined interacts with other operators is discussed in Section 5.

Matt Jaffe *et al.* have discussed the problems inaccurate models of unknown values may cause in critical control systems [13]. They define a collection of criteria specifiers must consider when developing models of critical software systems. Several criteria address the problems that occur when unknown environmental conditions are not handled correctly. Consider the following from [13]:

“The behavior of the software with respect to input received before startup, after shutdown, or when the computer is temporarily disconnected from the process (off-line) must be specified, or it must be determined that this information can be safely ignored.”

and

“Every state must have a behavior defined in case there is no input for a given period of time.”

Serious accidents have occurred because the designers of the software did not consider how the system would handle gaps in data caused by taking the software temporarily off-line. For example, an accident occurred in a batch chemical reactor when a computer was taken off-line to modify the software [15]. At the time the computer was shut down, it was counting the revolutions of a metering pump that was feeding the reactor. When it came back on-line, the software continued counting where it left off, eventually overcharging the reactor.

RSML^{-e} and SpecTRM-RL use a special undefined value as the default value upon startup or upon specific mode transitions (e.g., after temporary shutdown of the computer). This value is used to ensure consistency between the model of the process state and the real, physical, process state by forcing resynchronization of the model with the outside world after an interruption of processing [17]. There are two ways in which these languages could have specified a safe behavior for the chemical plant. If the software was notified by the system that it was being removed from control, then the specification should have explicitly set all process-monitoring environmental variables to undefined. Alternatively, if the control system was simply disconnected from the environment, a timeout condition could have been specified. In this case, if no input from the metering pump was received by the software within a certain period of time, the variable representing the revolutions should have been considered “stale” and set to undefined.

A second example from [13] illustrates another case where correct modeling of undefined values is critical. When values are received from the physical environment, Jaffe *et al.* recommends:

incoming values should have their values checked and there should be a specified response in the event of an out-of-range condition.

Variables that are out of bounds are appropriately modeled as having the undefined value to prevent inappropriate use in the computation of the new system state or outputs.

These situations, although easier to specify with an undefined value, do not *require* it. In the absence of an undefined value, there are other approaches to modeling missing data and failures. We can represent each environmental quantity as two variables, one that represents the value of the quantity, and the other that acts as a flag determining whether or not the model variable is meaningful. This is the approach used in SCR [10].

However, there are two problems with this approach. First, there is no semantic connection between the two variables. The specifier must ensure that wherever the *value-variable* is referenced, they have tested the *flag-variable* to ensure that the value-variable is meaningful. The second problem is that this solution may make the specification significantly larger since we, in effect, may double the number of input variables. One of our language goals is to provide a rich enough semantics that the features of the system to be controlled can be naturally and easily expressed.

Another approach is to use a special sentinel value at the edge of the range of the type (e.g., MAX_INT) to represent that the variable is undefined. However, this approach has the same problems that existed in the two-variable approach; it is possible to accidentally use the variable in arithmetic expressions without checking that the value of the variable is equal to the sentinel value (i.e., the variable has the value undefined). This usage may cause the specification to exhibit nonsensical behavior when executed and/or statically analyzed. It can also cause serious problems if the specification is used in a new environment, because the sentinel value may now be in the valid range of values for the new environment.

4. UNDEFINED IN RSML^{-e}.

In RSML^{-e}, `undefined` is a special value that can be assumed by variables and expressions in a specification language. It signifies that the value of the variable or expression is unknown. It is used in three situations within a specification.

Explicit Description: `undefined` can be explicitly assigned to a variable, or as the value returned by a function in RSML^{-e}. This usage of `undefined`, corresponds to the undefined value discussed in Section 3. For example, if we detect an error in an incoming message, if data is “stale”, etc., the specification can assign `undefined` to the variable.

Child State Variables: Hierarchical composition of state variables allows us to view the state of the RSML^{-e} specification as a hierarchical type. Often, in control systems, there is information that is only relevant if the system is in a particular state or mode. Child state variables provide a succinct way of representing this kind of information. Thus, in the next state computation, we first evaluate the top-level state variables. Then, we can “drill down” and further elaborate the current state by evaluating the child state variables.

Child state variables only contain relevant information in certain system states. We define a child variable to have a defined value (i.e. be *relevant*) when its parent variable equals a predefined *parentValue*. If the parent variable is not equal to the child’s *parentValue*, the value of the child variable is set to the special value `undefined`.

Expression Out of Range: RSML^{-e} maintains the assignment histories of all variables, and supports a set of expressions that allow a specifier to access previous values of a variable. If the specification asks for a previous value before the variable has been assigned enough times, `undefined` is returned. In other words, `undefined` is returned if the expression attempts to read past the beginning of the assignment history for a variable.

`undefined` unifies several seemingly distinct concepts within an RSML^{-e} specification. However, the *meaning* of these concepts is the same: the variable or expression has an unknown value. Without `undefined`, it would be more difficult to describe hierarchical state variables and anomalous conditions in the environment.

5. FORMALIZING UNDEFINED

Although `undefined` allows us to say many useful things easily in a specification, it is no longer clear whether the mathematical operators are total functions. For example, the meaning of $x + y$, where x, y are possibly undefined, could be described several different ways:

- As a partial function: (`undefined` is outside the domain of the operator)
- As a total function, returning `undefined` when provided an `undefined` argument.
- As an underspecified function, in which the result of the operator is defined to be a total function, but the value of the function is not specified when one of the arguments is `undefined`.

Existing specification languages use each of the techniques enumerated above to handle undefined values and the problems with partial functions that follow. The pros and cons of the techniques are summarized and illustrated below.

5.1 VDM: Logic of Partial Functions

VDM [3] allows type unions, so that it is possible to create new types that are, for example, the union of all integers plus some enumerated error values. In this way, VDM supports a more robust conception of `undefined` values than is described earlier in this paper. For example, given a pressure sensor, it is possible to distinguish between failure modes:

$$\text{pressureSensorType} = \text{nat} \mid \langle \text{random} \rangle \mid \langle \text{stuckHigh} \rangle \mid \langle \text{stuckLow} \rangle$$

When the sensor is operating normally, a variable of this type yields an natural number. If the sensor is returning random values because of a sensor failure, *random* is used.

Otherwise, if the sensor is stuck returning high or low values outside the rated range of the sensor, *stuckHigh* and *stuckLow* are used.

VDM also has optional types, which add the value *nil* to the type description. These optional types are built as type unions between the “basic” type, and the type $\langle nil \rangle$, which is the type containing only the value *nil*. This notion is almost exactly the same as the notion of `undefined` used in RSML^{-e}.

Preconditions in VDM define when functions are applicable. Functions should have a defined value in VDM whenever their precondition holds. If this is true, then no run-time errors can occur within the body of the function. However, as VDM uses a first-order predicate calculus for its preconditions, they are in general undecidable. Therefore, one must prove, perhaps with the aid of a theorem prover, that a function is complete with respect to its precondition, and that the function’s precondition is satisfiable.

VDM uses the logic of partial functions (LPF) [14] to handle partial functions. It uses a three-valued logic, containing true, false, and * (UNDEFINED). When a function is applied outside of its domain, the special value * is returned. The logic of VDM has composition rules for each of the logical operators (\wedge, \vee, \neg) when used with * arguments.

One problem with this formulation is the loss of the law of the excluded middle. This means that $(x \vee \neg x)$ is not necessarily true. Also, the associativity of equivalence (\equiv) is lost [1]. The IFAD implementation of a VDM specification [5] also behaves differently than the theoretical model. The IFAD implementation of a VDM specification does not actually use the * value, but instead relies on short-circuit evaluation of Boolean operators:

$$(x \notin \text{dom } f \vee f(x) = 5)$$

returns true if x is outside the domain of f , while

$$(f(x) = 5 \vee x \notin \text{dom } f)$$

causes a runtime error.

5.2 NP: Undefined Terms are false

Another approach, used in the *NP* checker [12] and advocated by David Parnas [18], is to assign the value false to any elementary formula containing an undefined function. This method simplifies implementation of tools that allow partial functions.

However, there are several problems with this approach. Many of the conventional laws of arithmetic no longer hold. For example, the law of trichotomy of arithmetic:

$$\forall x \bullet x = 0 \vee x > 0 \vee x < 0$$

no longer holds, because $x = 0$, $x > 0$, and $x < 0$ all evaluate to false, if x is an undefined term. Similarly, since $=$ is an elementary function,

$$f(x) = f(x)$$

is not true if $f(x)$ is undefined. One must also decide where to draw the boundary for elementary functions. For exam-

ple, is \neq an elementary function? If so, then

$$(f(x) \neq f(y)) \Leftrightarrow \neg (f(x) = f(y))$$

is no longer valid.

5.3 LARCH: Underspecified Functions

This approach, advocated by David Gries and Fred Schneider, avoids the use of an undefined value in logic by insisting that all functions be total. From [4]:

All operations and functions are assumed to be defined for all values of their operands – they are total operations and functions. However, the value assigned to an expression need not be uniquely specified in all cases.

In this case, functions are defined using implication, with an antecedent describing the domain over which the function is specified. This antecedent, a predicate over the operands of the function, fills nearly the same role as the function precondition in VDM. Given a function $f(a \times b \times c)$, using underspecification, the definition of f with precondition P would look like:

$$\begin{aligned} \forall a, b, c \bullet \\ (P \Rightarrow f(a, b, c) = \\ \quad \{\text{function definition for defined range}\}) \wedge \\ (\neg P \Rightarrow \exists d \bullet f(a, b, c) = d) \end{aligned}$$

With VDM’s preconditions, the definition looks like this:

$$\begin{aligned} \forall a, b, c \bullet \\ (P \Rightarrow f(a, b, c) = \\ \quad \{\text{function definition for defined range}\}) \wedge \\ (\neg P \Rightarrow (a, b, c) \notin \text{dom } f) \end{aligned}$$

The difference between the two definitions is that in the former, f is a total function that is not completely specified. In the second example, f is a partial function that is completely specified.

Standard mathematical logic does not have to be significantly altered to accommodate underspecified functions. The law of the excluded middle still holds, as do the standard laws of arithmetic. Unfortunately, underspecified functions are not well suited to an executable semantics. This is easiest to explain by example. Suppose we had a predicate:

$$f(a) > 100 \wedge a > 0$$

and suppose that the value of f is specified whenever $a > 0$. With the underspecified function approach, we always must assign *some* value to $f(a)$, even if a is outside the specified domain of f . For this particular predicate, that is acceptable; if $a \leq 0$, then it does not matter what we assign to $f(a)$, because it cannot affect the outcome of the predicate since the second part of the disjunct, $a > 0$, is false. However, suppose instead that f was specified whenever $a > 50$. If a was, for example, equal to 45, we would face a problem. Whatever value we assign to $f(a)$ will affect the value of the predicate, so we really should halt execution, rather

than allowing this random value to propagate through the evaluation of the specification.

The problem is that it is difficult, in general, to know whether the value of an underspecified function will affect the value of a predicate. If we generate a value, or prompt the user for a value, it is possible that the specification will behave in an unpredictable and erroneous manner, instead of halting. We could use a marker to denote an underspecified value, and test whether this marker is actually required to determine the value of the predicate. However, by doing so, we are essentially using a three-valued logic, with the marker playing the role of undefined. With a three-valued logic, we can assign undefined to the function, and allow it to propagate through a predicate. We only need to halt execution if the predicate as a whole is undefined.

5.4 Z: several options

There have been several different semantics suggested for Z [2]. However, the standard semantics maintains that if a partial function is used outside its domain, the result is some “unknown” value that is a member of the expected type of the function. This definition, although intuitively appealing, gives rise to the problem that if x is outside the range of f , it is possible that $f(x) \neq f(x)$. This approach is similar to the the approach of Larch (Section 5.3), but it does not require that the value returned by a function outside its domain obey any functional properties.

6. RSML^{-e}: SIMULATION VS. PROOF SEMANTICS

The goal of an RSML^{-e} specification effort is to create a complete and consistent specification that is correct with respect to the user’s needs. However, we desire to simulate the system in question from a very early stage, when a specification is rife with inconsistencies and incompleteness. In [21], the denotational semantics of RSML^{-e} are described using Z. Although Z has a given semantics for partially-defined functions, we can model different semantics for RSML^{-e} by adding Boolean values as a “meta-type” that describe how predicates and partial functions are evaluated.

RSML^{-e} extends each of the user-visible types in the language with `undefined`, so we must consider how expressions and operators work with this value. RSML^{-e} explicitly does *not* include a user-visible Boolean type because of this reason. If such a type was included, we would be forced to use a three-valued logic for RSML^{-e} semantics.

6.1 Simulation semantics

For simulation of incomplete/inconsistent specifications, RSML^{-e} uses a three-valued logic, because it is more robust than the other choices. By allowing Boolean-undefined values, it is not necessary to halt if a term is undefined (see 5.3). Because RSML^{-e} does not allow recursion, there is no risk of non-termination when evaluating undefined atomic formulas. Therefore, we can allow `undefined` atomic formulas at arbitrary positions in a predicate.

As we already have a value that represents that a value is unknown (`undefined`), we use it as the third value for our

logic. In this way, the value `undefined` fills the role of both *nil* and UNDEFINED (*) in VDM.

In almost every case, when operators are used on an expression containing `undefined`, the result is `undefined`. The only exceptions are the operators $\{=, \neq, \wedge, \vee\}$. The $=$ expression will return true if both sides of the predicate are `undefined`, and false if one, but not both, sides of the predicate are `undefined`. The \neq expression does the opposite. The binary Boolean operators, when presented with at least one `undefined` argument, behave the same as in VDM:

A	B	$A \wedge B$
<code>undefined</code>	T	<code>undefined</code>
T	<code>undefined</code>	<code>undefined</code>
<code>undefined</code>	F	F
F	<code>undefined</code>	F
<code>undefined</code>	<code>undefined</code>	<code>undefined</code>

In cases when one argument is false, we can guarantee that the expression is false. Otherwise, the result is `undefined`. Similarly, the logical-OR operator is true if one of it’s arguments is true:

A	B	$A \vee B$
<code>undefined</code>	T	T
T	<code>undefined</code>	T
<code>undefined</code>	F	<code>undefined</code>
F	<code>undefined</code>	<code>undefined</code>
<code>undefined</code>	<code>undefined</code>	<code>undefined</code>

Because variables of all types (integer, floating point, etc.) can be set to `undefined`, the value of certain expressions can be counter-intuitive. Supposing that x and y were undefined, the following expressions are all true under the formalism:

$$\begin{aligned} x &= 1 + x \\ (x = y \wedge x = y + 1) \\ x &= \neg x \end{aligned}$$

This problem is especially troublesome because a specifier may not realize that a variable can assume the value `undefined`. Any expression in which `undefined` could be used with an operator other than $\{=, \neq, \wedge, \vee\}$ is considered *u-dangerous*. Therefore, expressions that can yield the value `undefined` should be *u-guarded*. A specification is *u-guarded* given two conditions:

1. The only way a variable can be assigned `undefined` is because it is the child of another variable, or it was assigned the literal expression `undefined` (i.e. No variable can implicitly be assigned the value `undefined`).
2. No predicate can be decided by an *undefined-dangerous* sub-expression.

A static check, defined in Section 7 defines the procedure for checking a specification for correct use of `undefined`.

6.2 Static Analysis/Code Generation Semantics

Although a three-valued logic allows us to simulate specifications very early on, it is not convenient for the other purposes of RSML^{-e}. A logic based on underspecified functions is mathematically cleaner, which aids translation into theorem proving tools and model checkers, and has a semantics that is better suited to code generation.

When generating code for a specification, we must map the RSML^{-e} types into the types native to the target language. Most imperative languages share the same basic types: Booleans, integers, floating-point numbers, etc. To map the RSML^{-e} integer type, containing an `undefined` value, to an Ada or C integer, we define the maximum (or minimum) RSML^{-e} integer value to be one less (greater) than the value from the target language. The remaining value becomes the `undefined` value for the generated code. In this case, the result of performing an arithmetic operation with an `undefined` operand becomes an underspecified function.

In order to implement the *simulation semantics* in a generated program, it would be necessary to test whether either of the arguments to an arithmetic operator was `undefined`. If so, the result would be set to `undefined`; if not, it would perform the operator normally. This approach is not acceptable both because of execution speed and code complexity.

6.3 Equivalence Conditions

Despite the arguments provided above, supporting two different semantics for `undefined` values may seem unnecessary and burdensome to the specifier. However, it turns out that a correctly written specification that is *u-guarded* will behave equivalently in either case. The simulation semantics is more forgiving when a specification is incomplete and inconsistent, which is why we have adopted it for our simulation engine.

7. UNDEFINED CHECKS FOR RSML^{-e}

As discussed in Section 6, using `undefined` in expressions other than the safe operators (i.e., the set $\{=, \neq, \wedge, \vee\}$) can cause unexpected results. Both of the u-guarded conditions merit some explanation. First, to ensure that a variable is not implicitly assigned `undefined`, we check the predicates of all of its assignment cases. Suppose we have a case

$$x := 5y \text{ if } y = z + 10$$

where y is a variable that can potentially equal `undefined`. This case would be in error, because we could implicitly assign x `undefined` if y was `undefined`. To satisfy the first condition, we would have to add:

$$x := 5y \text{ if } (y \neq \text{undefined} \wedge y = z + 10)$$

A subexpression is guarded if whenever it could equal `undefined`, it cannot affect the value of the predicate. Using our earlier example:

$$y \neq \text{undefined} \wedge y = z + 10$$

suppose that z could equal `undefined`. Then, this predicate has an *u-dangerous* subexpression ($z + 10$). However, the predicate can be guarded by adding:

$$y \neq \text{undefined} \wedge z \neq \text{undefined} \wedge y = z + 10$$

Then, whenever z is `undefined`, the predicate is guaranteed to be false.

7.1 Computing U-Guardedness

The RSML^{-e} next-state relation is created by composing the assignment relations of the interfaces and the state variables. If we can show that each of the assignment relations is u-guarded, it follows that the next-state relation is u-guarded. As each assignment relation is defined by a set of cases, we can further partition the analysis to check each assignment case separately. Finally, as predicates in RSML^{-e} are in disjunctive normal form, we can analyze each disjunct separately.

The analysis of *each disjunct* proceeds in four steps:

- 1. Create a Canonical Form of the Disjunct:** To make the analysis easier, we move all logical NOTs as far as possible into each atomic formula in the disjunct. Thus,

$$\neg \neg \neg (x = y)$$

becomes

$$x \neq y$$

This step makes it easier to compare expressions later on in the procedure.

- 2. Find Potentially-Dangerous Expressions:** There are five types of expressions in RSML^{-e} that can equal `undefined`. These expressions, such as variable value expressions, are considered *potentially dangerous* when used with a *u-dangerous* operator.

To find all of the potentially dangerous expressions for a disjunct, we consider both the sub-expressions within the disjunct, and also the sub-expressions in the assignment expression for the assignment case. This creates the set PD .

- 3. Transform PD into D :** Because of the structure of the specification, many of the expressions in PD are not dangerous. For example, an input variable that does not have an initial value of `undefined` and which is never explicitly assigned the value `undefined` will never equal `undefined`. These expressions are removed from the set PD , creating the set of *dangerous* expressions D .

- 4. Creating the Guarded-Expression Set:** In this step, we find all expressions within the disjunct of the form:

$$x \neq \text{undefined}$$

or

$$\text{undefined} \neq x$$

where x is any arbitrary sub-expression. We use these expressions to create the set of guarded expressions

(GE) containing all of the *sub-expressions* (i.e., x) that are u -guarded.

A disjunct is *u-guarded* if the set of dangerous expressions is a subset of the guarded expressions. In other words, a disjunct is *u-guarded* if $D \subseteq GE$. An assignment case is *u-guarded* if all of its disjuncts are *u-guarded*, and so on. If all of the assignment relations are *u-guarded*, then the specification is *u-guarded*. This algorithm rejects all specifications that are not correctly guarded, but also does not accept a class of specifications that are correctly guarded. The third step, transforming PD into D , is currently not very sophisticated. It only takes into account structural factors, e.g., whether or not a variable is *ever* assigned **undefined**. We are investigating methods to improve our algorithm.

8. CONCLUSION

to summarize, in this paper we described a mechanism for representing unknown values in specifications, using a special value **undefined**. This approach allows us to directly model uncertainty about environmental variables in the variables within our model. We then compared this approach to other techniques to represent an environmental variable: (1) using two variables, one to record the value and one to record the validity of the value, and (2) using a sentinel value to represent that a value is not valid, and discussed why neither approach is satisfactory.

We then described different approaches for supporting undefined values in specification languages. This problem is a specific case of the general problem of representing partial functions. Each approach has some benefits and associated problems.

To take full advantage of the use of undefined values while at the same time circumventing the problems, we have chosen to support two different semantics. The first, a loose simulation semantics, can be used to simulate and execute specifications even when they are incomplete, inconsistent, and contain many undefined values and partial functions. It uses a three-valued logic to be more robust when portions of a specification are not completely specified. Because this semantics is not very clean, however, it is not as well suited for static analysis and code generation. Therefore, we also support a tight semantics for static-analysis and code-generation. This semantics uses a two-valued logic, making it easier to translate into an imperative programming language or input language for a verification tool such as a model checker or theorem prover. Finally, we defined a *u-guarded* check to ensure that a completed specification will have the same behavior using both semantics, and that we are not misusing undefined values in the specification. The semantics have been fully formalized [21] and our RSML^{-e} development environment NIMBUS [19] supports the looser semantics. Our correctness preserving code generation approach, on the other hand, supports the tighter semantics [20].

9. REFERENCES

- [1] E. Aaron and D. Gries. Formal justification of underspecification. *Information Processing Letters*, 64(3):115–121, November 14 1997.
- [2] R. Arthan. Undefinedness in Z: Issues for specification and proof. via the world-wide-web: <http://www.lemma-one.com/papers/27.doc>.
- [3] J. Fitzgerald and P. G. Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998.
- [4] D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. *Computer Science Today: Recent Trends and Developments (Lecture Notes in Computer Science)*, 1000:366–373, 1995.
- [5] T. V. T. Group. The IFAD VDM⁺⁺ toolbox user manual. Technical Report, IFAD-VDM-43. Available from IFAD, Forskerparken 10, 5230 Odense M, Denmark, September 1997.
- [6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.
- [7] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
- [8] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.
- [9] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, pages 363–377, June 1996.
- [10] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.
- [11] K. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, January 1980.
- [12] D. Jackson and C. A. Damon. Nitpick reference manual. Technical Report CMU-CS-96-109. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.
- [13] M. S. Jaffe, N. G. Leveson, M. P. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.
- [14] C. Jones and C. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.
- [15] T. Kletz. Wise after the event. *Contr. Instrum.*, 1988.

- [16] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, pages 684–706, September 1994.
- [17] N. G. Leveson, M. P. Heimdahl, and J. D. Reese. Designing specification languages for process control systems: Lessons learned and steps to the future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, September 1999.
- [18] D. Parnas. A logic for describing, not verifying, software. *Erkenntnis (Kluwer)*, 43(3):321–338, November 1995.
- [19] J. M. Thompson. NIMBUS: A framework for static analysis and simulation of system-level inter-component communication. Master's thesis, University of Minnesota, December 1999.
- [20] M. W. Whalen. Automatic code generation for safety-critical systems. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, 1999.
- [21] M. W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, March 2000.