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 that 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.

Keywords
Partial functions, undefined values, formal specifications

1 INTRODUCTION
Executable specification languages, such as Statecharts [4], SCR [7, 6], RSML [12], and SpecTRM-RU [13], are becoming more widely used in the specification of safety-critical 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 [17].

There were four goals that provided a framework to evaluate different approaches to represent unknown values:

- The syntax for representing unknown values should be clear and concise.
- The semantics of unknown values must be intuitive and easy to understand.
- The semantics should support static checks to ensure unknown values are not misused within a specification.
- The semantics must lend themselves to other static analyses by standard tools such as model checkers and theorem provers.

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

The contribution of this paper is twofold: first, as far as we are aware, no paper specifically describes and analyzes different approaches to model unknown monitored variables from the environment. Second, we provide an approach to modeling unknown values in RSML−e and describe a straightforward type check for RSML−e specifications that ensures that unknown values are not misused.
the next section, we will provide a short description of the RSML$^c$ language. In Section 2, we will describe different uses for undefined values in specifications. In Section 5, we describe the different ways in which undefined values are used in RSML$^c$. 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 4, and describe our approach for RSML$^c$ in Section ???. Finally, we describe an algorithm for ensuring that undefined values are used correctly within RSML$^c$ specifications in Section 6.

2 JUSTIFICATION
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.

Matt Jaffe et al. have described specific requirements for software specifications that are related to unknown values, and accidents that have occurred because of the misuse of unknown variables. Consider the following requirements from [9]:

"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."

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

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

These requirements describe conditions in which the behavior of the specification must be modified because the true value of an environmental variable is no longer known.

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 [11]. 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.

In order to correctly specify the behavior of control systems when a variable has an unknown value, two things are required. First, we need a mechanism for representing instances when the true value of an environmental variable is unknown. Second, we need a test to ensure that unknown variables are not misused within a specification. The formal definition of 'misused' is dependent on the specification language; informally, however, the goal is that any calculation or relational operator that uses unknown values should be skipped or ignored.

3 APPROACHES FOR MODELLING UNKNOWN VARIABLES
We considered three approaches for specifying when a variable is unknown.

Sentinel Value (SV)
In this approach, the specifier creates an value outside the expected range but within the expressible range of a variable to represent unknown.

This approach is often used in imperative language programs, so it has the advantage of being familiar to programmers, and it does not require changes to the type system of the language. However, there are two serious drawbacks to this approach. First, because unknown variables are not a facility of the language, it is difficult to perform static checks to ensure that unknown values are not misused. Second, portions of specifications are often re-used in different environments. The sentinel value may be outside the expected range of values in the original environment, but not in a new environment.

Orthogonal Value (OV)
This approach extends each of the supported types with a special unknown value, representing unknown. This value is outside the normal expressible range of the variable type, so there is no risk of the unknown value being
a possible valid value of the variable. Also, because the unknown value is part of the language, it is straightforward to build static checks to make sure that potentially unknown variables are not misused. However, the behavior of unknown values with relational and arithmetic operators must be considered.

Flag Variable (FV)
In this approach, each potentially unknown variable is represented as a pair of values, with the first value representing the variable value, and the second value representing whether or not the variable is defined. This can be represented in the specific language either by a tuple or a pair of variables.

This approach has the benefit that the operators do not have to be redefined as in OV. However, this solution requires more effort by the specification writer. There is no automatic semantic connection between the flag and value variables, so this must be manually provided by the writer. Another problem is that this solution may make the specification significantly larger since it requires the effect, may double the number of input variables.

For the rest of the paper, we will describe the OV (or OV) approach, as it is more concise than FV and requires no effort by the specifier to analyze for misuse, unlike either FV or SV.

4 ISSUES IN FORMALIZATION
The unknown value is outside the set of integers and real values, so the arithmetic/relation operators, as they are normally defined, simply do not apply to unknown, just as the divide operator does not apply to a zero divisor. Therefore, we can either choose to redefine the arithmetic/relation operators so that they are total w.r.t. unknown, or we must discuss logical frameworks that allow partial functions.

For example, the meaning of $x + y$, where $x$, $y$ are possibly unknown, could be described as any of the following:

1. As a total function, returning unknown when provided an unknown argument.

2. As a partial function: (unknown is outside the domain of the operator)

3. 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 unknown.

The first choice, while intuitively appealing, is not acceptable, because it can lead to counter-intuitive behavior from specifications. Supposing that $x$ and $y$ are unknown, the following expressions would be true:

\[
x = 1 + x \]
\[
(x = y) \land (x = y + 1)
\]

The second and third choices are based in two different approaches for handling partial functions within a logical framework.

The debate on how to handle partial functions within logic is older than the field of computer science, and it has spawned an immense volume of research. In order to focus the discussion in this paper, we consider only two criteria for evaluating the different formulations. First, whether or not the logic have 'tricks' that can make specifications containing unknown values error-prone to write and reason about. Second, whether it is difficult to perform static analyses relating to unknown. We provide a brief overview of three approaches advocated by leaders of the software engineering community.

**Logic of Partial Functions (LPF)**
The logic of partial functions, advocated by Jones [10], and used in VDM [2] adds a third truth-value, to logic that signifies that the truth-value of the predicate is unknown. When a function is applied outside of its domain, the special value * is returned. This logic has composition rules for each of the logical operators ($\land$, $\lor$, $\neg$) when used with * arguments, as follows:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \land B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

In cases when one argument is false, we can guarantee that AND-expressions are false. Similarly, the logical-OR operator is true if one of its arguments is true:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \lor B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>*</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

This formulation allows a straightforward and intuitive formulation of unknown values. Arithmetic and relational operators can be left as partial functions. The value of a partial function outside of its domain is * , so the value of any predicate that uses unknown values in an essential way is *. It is straightforward to formulate tests that check whether unknown is misused in this logic (see Section 6). Unfortunately, it is more difficult to reason in a three-valued, rather than two-valued logic. The law of the excluded middle is lost, mean-
ing that \((x \lor \neg x)\) is not necessarily true. Also, the associativity of equivalence \((\equiv)\) is lost [1].

**Undefined Atomic Formulas are False (UAFF)** Another approach, used in the NP checker [8] and advocated by David Parnas [14], is to assign the value false to any atomic formula containing an undefined function. This method simplifies implementation of tools that allow partial functions.

However, there are several problems using this approach with an **unknown** value. First, there are equivalent-looking constructs that have different meanings within a specification. For example, suppose that a function is defined:

\[
    f(x) = \begin{cases} 
    0 & \text{if } x < 100 \\ 
    1 & \text{if } x \geq 100 
    \end{cases}
\]

Assuming that \(x\) can be **unknown**, this function is partial, because both cases are false when \(x = \text{unknown}\). However, this equivalent-looking function is correct:

\[
    f(x) = \begin{cases} 
    0 & \text{if } x < 100 \\ 
    1 & \text{if } (x < 100) 
    \end{cases}
\]

In the case where \(x\) is **unknown**, the atomic formula \(x < 100\) will be false, so \(\neg(x < 100)\) will be true.

With UAFF, slight syntactic differences change the meaning of the specification. Specifications may also be 'artificially' complete. There is a good chance that in the latter formulation of \(f(x)\), the specifier did not consider the case where \(x\) is **unknown**, and allowed unexpected behavior to creep into the specification.

Many of the conventional laws of arithmetic no longer hold in UAFF. For example, the law of trichotomy of arithmetic:

\[
    \forall x \cdot x = 0 \lor x > 0 \lor x < 0
\]

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.

[COME BACK TO THIS...] Finally, it is not possible to specify a semantic test to discover whether **unknown** values are being misused.

**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 [3]:

\[
    \text{All operations and functions are assumed to be defined for all values of their operands -- they are total operations and functions. However, the value assigned to an expression need not be uniquely specified in all cases.}
\]

In this case, functions are defined using implication, with an antecedent describing the domain over which the function is specified. Under this approach, arithmetic and relational operators would be modified so that they are specified only when presented with numeric arguments.

The value of the operators when presented with **unknown** arguments would be left unspecified. 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. However, it is difficult to directly formulate some useful specification properties using this logic, most notably specification completeness, since every function is by definition complete! At a metalevel, however, it is still possible to check these properties by testing whether all possible models of a given function are the same. If so, then the function must be completely specified.

5 **Supporting Unknown Variables**

Next we would to describe how **unknown** values are supported by the RSML-\(\varepsilon\) language. To do so, we first provide a brief overview of the RSML-\(\varepsilon\) language.

**Overview of RSML-\(\varepsilon\)**

RSML-\(\varepsilon\) is based on the the RSML language developed by the Irvine Safety Research Group [12]. RSML-\(\varepsilon\) was developed as a requirements specification language specifically for embedded control systems. It is designed to easy to read and understand by engineers without sophisticated mathematical backgrounds [13], and is a hybrid graphical/tabular notation, similar to SCR.

RSML-\(\varepsilon\) specifications consist of variables, interface functions, and macros. **Variables** describe the internal state of the system. **Interfaces** describe how the specification interacts with the external environment. **Functions** and **Macros** are mechanisms for representing common expressions and predicates, respectively, in order to make specifications more concise. Each construct will be explained more clearly in the following paragraphs.

A static set of **variables** form the internal state of an RSML-\(\varepsilon\) specification. All variables are associated with a type. **Types** include subranges of integers and real numbers, as well as Booleans and enumerations. Each numeric type must also have a specification of its units of measurement. The state of the specification includes the complete **true**, or assignment history, of every vari-
able. There are two classes of variables in RSML-\textsuperscript{e}, \textit{input} and \textit{state} variables.

\textit{Input variables} represent inputs received from the external environment. These could correspond to sensor values, input from other software systems or the user interface of the system.

\textit{State variables} represent all variables whose value is computed by the RSML-\textsuperscript{e} specification. State variables can be grouped into mode and output variables, which differ by purpose but not semantics. They describe a mutually exclusive set of system behaviors \cite{13}, while outputs are computed in order to be sent to the external environment. State variables are similar to mode classes in SCR \cite{6}. Unlike SCR, however, it is possible to define hierarchical relationships between state variables. These hierarchal relationships allow a specifier to directly model hierarchical system modes.

\textit{Interfaces} form the boundary between the RSML-\textsuperscript{e} specification and the external environment. RSML-\textsuperscript{e} specifications communicate with the external environment through \textit{messages}, which are simply records of data, that are sent and received through unidirectional \textit{channels}. Interfaces define expected properties for communication, such as the expected minimum and maximum separation between messages on a channel. Also, interfaces regulate the assignment of inputs to the specification to the \textit{input variables}, described in the next paragraph. Using this feature, simple safety and liveness constraints can be imposed by the interfaces without considering the (potentially complex) next state computation involving the variables and associated definitions.

To illustrate the different constructs, we use a small example of an altitude switch, illustrated in Figure 1. The altitude switch is designed to turn on a device-of-interest (DOI) in an airplane when the altitude of the airplane drops below a certain threshold. There are a few complicating factors for the switch:

- the altitude switch can be manually inhibited or reset by the crew of the airplane.

- the switch contains fault detection logic to determine whether the switch and the DOI are behaving correctly.

For the altitude switch, the control software must communicate with the DOI, an altimeter, and a pilot's display that contains an inhibit and reset button as well as a "fault detected" indicator. The interfaces shield the rest of the specification from details about how the system is connected to the external environment. In this case, the reset and inhibit buttons are separately wired into the controller, along with an altimeter value and the DOI status.

The input variables: \textit{InhibitSignal}, \textit{Reset}, \textit{DOIStatus}, and \textit{Altitude} are assigned by the input interfaces as messages are received from the environment. The \textit{DOI status} variables are \textit{SystemMode}, which describes the overall mode of the specification, \textit{AltitudeStatus}, which describes where the plane is in relation to the altitude threshold, DOI, our model of the device of interest, and \textit{ASWOpModes}, the operational modes for the altitude switch. There are two output variables for the model: \textit{DOICommand} and \textit{FaultDetected}, which describe the command to be sent to the DOI and whether or not the switch has detected any faults. This specification uses hierarchical modes: DOI, AltitudeStatus, and ASWOpModes are children of SystemMode.

Assignment \textit{actions} in RSML-\textsuperscript{e} determine the value of state variables and interfaces. They are all organized as case functions, with a \textit{condition} describing when the case is relevant, and a \textit{value-assignment} describing the value of the function. For state variables, the value-assignment is an atomic value. For input interfaces, it is a list of assignments to input variables based on the contents of a received message. For output interfaces, it describes whether or not a message is sent to the exter-

![Figure 1: A graphical view of the components of the altitude switch.](image-url)
nal environment, and if so, the contents of the message. Mechanized procedures exist to ensure that the assignment functions are complete and consistent [5].

The assignment functions are placed into a partial order based on data dependencies and the hierarchical structure of the state machine. An object is data-dependent on any other object that it names in its assignment function. If a state variable is a child variable of another state variable, then the child is also dependent on its parent variable. The value of each object can be computed after the items on which it is data-dependent have been computed. An example of the data dependencies of the altitude switch are shown in in Figure 2.

Figure 2: The data dependencies of the altitude switch.

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 [12] (see Figure 3). 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.”

![Truth Table for Conditions](image)

Figure 3: An example condition

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

**Uses of unknown in RSML-\(\text{c}\)**

In RSML-\(\text{c}\), unknown is a special value that can be assumed by variables in the specification language. unknown values are internal to the specification; it is not legal to receive or send unknown through an interface. There are two reasons for this choice. First, unknown is supposed to represent a situation where the specification does not have knowledge about the environment, which is subtly different than knowing that the state of a particular variable is unknown. Second, more practically, when simulating/executing specifications, we want to communicate with components that may not support an unknown value.

unknown can be explicitly assigned to a variable, or as the value returned by a function in RSML-\(\text{c}\). For example, if we detect an error in an incoming message, if data is “stale”, etc., the specification can assign unknown to the variable. In the altitude switch, the control software expects regular updates from the altimeter at 100 ms intervals. If the AltitudeInterface does not receive an update within two seconds from the altimeter, it sets the value of Altitude to unknown.

unknown is also used to support hierarchical composition of state variables. In hierarchical composition, child state variables are only relevant (i.e., defined) if their parent variable equals a specified parent value. Otherwise, the child state variables are assigned unknown. In the altitude switch, AltitudeStatus, DOI, and ASWOpMode are assigned unknown if SystemMode is not equal to Operational.

[NOTE: Add Examples from Altitude Switch]

The complete formalization of RSML-\(\text{c}\) in [17]. In this formalization, we have used a three-valued logic, basically LPF, to describe the behavior of specifications. However, as discussed in Section 4, there are drawbacks to each logic that handles partial functions. Furthermore, we wish RSML-\(\text{c}\) specifications to be analyzable in tools such as theorem provers and model checkers, which may use alternate logics. Ideally, we would like our specifications to behave the same regardless of the underlying logic. In Section 6, we discuss checks that ensure that unknown values are not misused in RSML-\(\text{c}\) specifications. As an ancillary benefit, these checks ensure the behavior of specifications w.r.t. unknown values is the same across the three logics described earlier.

**6 UNDEFINED CHECKS FOR RSML-\(\text{c}\)**

In this section, we discuss a-guardedness checks, which are a form of type-checking performed to ensure unknown values are not misused in a RSML-\(\text{c}\) speci-
First, we define unsafe operations. RSML-ε supports the standard arithmetic, relational, and logic operators: \{+,-,\times,/,\land,\lor,\land,\lor,\iff,\equiv,\neq\}. We can safely test \texttt{unknown} values for equality and inequality, but all other operators do not yield a defined (or at least known, in the case of UF-approach) value. Therefore any operation with an \texttt{unknown} operand and an operator in the set \{+,-,\times,/,\land,\lor,\iff,\equiv,\neq\} is unsafe.

There are three criteria that must be satisfied to show that a specification is \textit{u-guarded}.

1. No \texttt{unknown} values are emitted to the environment.
2. No conditions make essential use of an unsafe operation.
3. If a condition in a function is satisfied, the value-assignment for the function does not contain an unsafe operation.

The first criteria ensures that \texttt{unknown} values in RSML-ε are \textit{internal} to the specification. The second condition bounds the safe use of potentially \texttt{unknown} values in conditions. A condition makes essential use of an unsafe operation if that expression affects the value of the condition. In a three-valued logic, essential use means that an unsafe operation in a condition \(c\) makes the condition’s value to be equal to \(\bot\). The third condition ensures that \texttt{unknown} values will not cause functions to become undefined.

**Computing U-Guardedness**

As described in Section 5, the value of all objects are computed via assignment functions. These functions are organized into cases, and the conditions of these cases are in disjunctive normal form. We perform most of our analysis on the disjuncts. By ensuring that each disjunct does not make essential use of \texttt{unknown}, we can ensure that criteria 2 is satisfied. By adding extra restrictions onto all disjuncts for a particular case, we can ensure that criteria 3 is satisfied.

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 Variables:** Any expression that uses a variable with an operator other than \{\iff,\neq\} is potentially dangerous. To find all potentially dangerous variables for a disjunct, we consider all of the variables in the disjunct, and we consider all variables used in the value-assignment portion the case that contains this disjunct, creating the set \(PD\).

We do not have to consider expressions that try to use the literal value \texttt{unknown} in an unsafe operation (e.g. \(5 + \texttt{unknown}\)), as it is considered a syntax error.

3. **Transform \(PD\) into \(D\):** Because of the structure of the specification, many of the variables in \(PD\) are not dangerous. For example, any input or top-level state variable that is not explicitly assigned \texttt{unknown} will never equal \texttt{unknown}. State variables are \textit{top-level} if they are not children of other state variables.

4. **Creating the Guarded-Variable Set:** In this step, we look for expressions of the form:

\[ x \neq \texttt{unknown} \]

or

\[ \texttt{unknown} \neq x \]

where \(x\) is variable. Any variable in one of these expressions is considered \textit{guarded}. For any disjunct, we add all guarded variables to the set \(G\).

A disjunct is \textit{u-guarded} if the set of dangerous variables is a subset of the guarded variables. In other words, a disjunct is \textit{u-guarded} if \(D \subseteq G\). An assignment case is \textit{u-guarded} if all of its disjuncts are \textit{u-guarded}, and so on.

Criteria 1 is the only one that is not tested by disjuncts. To satisfy criteria 1, we place an additional restriction on the value-assignments for output interfaces.

5. **Output Interface Assignment Function Check**

An output interface is \textit{u-guarded} only if the literal value \texttt{unknown} is not used in any value-assignments for the output interface.

A specification is \textit{u-guarded} if all assignment functions within the specification are \textit{u-guarded}.

[EXAMPLE OF ALTITUDE GUARDS IN ALTITUDE SWITCH]

**Proving the u-guardedness Check**

To make sure that the \textit{u-guardedness} check is useful, we need to make sure that it meets our criteria. We start by showing that no unsafe operation in a condition causes the condition’s value to equal \(\bot\).
Theorem 1  Every u-guarded disjunct containing an unsafe operation is false.

Proof: Suppose that a disjunct disjoint is u-guarded, and an arbitrary variable \( x \) is used in an unsafe operation in the disjunct.

Since \( x \) is a variable used in the disjunct, it will be in \( PD \). \( x \) is unknown either because it was explicitly assigned UNDEFINED, or because it is a child state variable. In either case, it will be a member of the set \( D \) for this disjunct. Since disjunct is u-guarded, \( x \) must also be a member of the set \( G \). This means that there exists an expression \( x \neq \text{undefined} \) for the disjunct. Since \( x \) equals unknown\(, this expression must be false, and therefore, the disjunct is false.\)

Corollary 1  No unsafe operation in a condition causes the condition's value to equal *.

Suppose a condition is equal to * because of an unsafe operation. Then one of the disjuncts containing the operation must be equal to *. But Theorem 1 states that this is not possible.

Criteria three states “If a condition of a case is satisfied, the value-assignment for the case does not contain an unsafe operation”. This is equivalent to the following theorem.

Theorem 2 If the value-assignment for a case contains an unsafe operation, then the condition for the case is false.

Proof: Suppose the condition of a case is satisfied, and that the value-assignment for the case contains an unsafe operation using variable \( x \). Then a disjunct disjoint from the case condition must be true. Since all variables used in the value-assignment are members of \( PD \) for all disjuncts, \( x \) must be a member of \( PD \) for disjunct disjoint. From here on, the proof follows exactly the same logic as for Theorem 1.

Theorem 3  No unknown values are emitted to the environment by a u-guarded specification.

Proof: RSML-un specifications send messages to the external environment through output interfaces, which are assigned by assignment functions. If an unknown value was emitted to the environment, it would either be implicit, because of an unsafe operation in a value-assignment for the output interface, or explicit, because the literal value unknown was used in a value-assignment. By Theorem 2, we know that a condition for a case is false if the value-assignment for the case contains an unsafe operation, so it is not possible that we would have an implicit unknown value sent to the environment. By step 5, we know that the literal value unknown can be used in a value-assignment for the output interface.

7  CONCLUSION

to summarize, in this paper we described a mechanism for representing unknown values in specifications, using a special value unknown. 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 [17] and our RSML-un development environment NIMBUS [15] supports the looser semantics. Our correctness preserving code generation approach, on the other hand, supports the tighter semantics [16].

REFERENCES


