Many-Valued Logic, Partiality, and Abstraction in Formal Specification Languages

REINER HÄHNLE, *Chalmers University of Technology, School of Computer Science and Engineering, 41 296 Gothenburg, Sweden, E-mail: reiner@chalmers.se*

Abstract

The purpose of this article is to clarify the role that many-valued logic can or should play in formal specification of software systems for modeling partiality. We analyse a representative set of specification languages. Our findings suggest that many-valued logic is less useful for modeling those aspects of partiality, for which it is traditionally intended: modeling non-termination and error values. On the other hand, many-valued logic is emerging as a mainstream tool in abstraction of formal analyses of various kinds, and we suggest that specification languages feature many-valued abstraction logics.

Keywords: Formal specification, many-valued logic, partial functions, abstraction

1 Introduction

The purpose of this article is to clarify the role that many-valued logic can or should play in formal specification of software systems,[1] which is often mentioned as one of its major applications.

Many-valued, or, to be more precise, three- and four-valued logics have long been used to model partiality in formal specification languages [4, 8]. For instance, one of the oldest and most important three-valued logics, Kleene's strong connectives [43], was introduced to model non-terminating recursive functions. Modeling partiality in formal specification and verification is often mentioned in texts on many-valued logic as a justification to use multiple truth values [35, 48, 52], although rarely discussed in detail. Many well-known formal specification languages, including VDM-SL [39], RSL [30, 31], *Z* [40], and the recent OCL [59], use many-valued logic or undefined values to model partiality. There is also a suggestion to include multiple truth values in Hoare logic [11]. On the other hand, some specification languages do not use undefined values, notably Larch [34], JML [45], B [1], and Alloy [33].

There is a considerable amount of literature on the subject of partiality in formal specification. Attempts to classify various approaches to partiality can be found, for example, in [3, 25, 27, 33].

In this paper we make the following contributions: first, in Section 2 we discern three fundamentally different aspects of partiality phenomena. Following that, in Section 3 we summarize the different kinds of many-valued logic that are used for modeling various kinds of partiality. In the course of this discussion we exhibit a

[1] We exclude from our discussion the use of many-valued logic for modeling partiality in mathematics, linguistics, and philosophy.

loss of semantic precision while going from propositional connectives to first-order quantifiers and we make an explicit connection between the analysis of logic programs done in the 1980s and three-valued logics with least fixed point operators. Section 4 contains an overview of how many-valued logic is utilized in a fairly representative set of specification languages. Our analysis entails that many-valuedness seems not the right tool for modeling partiality in a concise way, at least not in the present state of affairs. There are no new technical results in this section, but we are not aware of a similarly complete discussion in the literature. In Section 5 we review two-valued approaches and argue that at least for the specification of structured software they are superior to many-valued modeling. Our contribution here is a clarified semantics of underspecification. We also demonstrate that in behavioral specification this approach makes guards unnecessary even in most partial definitions. Finally, in Section 6 we argue for inclusion of abstraction in specification languages as the natural counterpart to refinement in cases where precision is traded in for lower computational complexity. In this context one can motivate many-valued logics as a compositional approximation of concrete or nondeterministic behavior without loss of precision. Section 7 concludes the paper.

2 Three Aspects of Partiality

A closer look at the usage of partiality that one encounters in software specification reveals that there are at least *three* different partiality phenomena. It is important to analyse and distinguish them carefully:

Non-termination A subcomputation needed for the evaluation of an expression does not terminate.

Error value A computation has an erroneous result, because it was called with an illegal value. Prototypical examples are `1/0` and `top([])`. An illegal value is not intended to occur, but if it does, one has to handle it. Trying to evaluate `1/0` typically will terminate, but there is no reasonable result value. In the context of program execution, a runtime error is generated or an exception is thrown.

Nondeterminism A different situation arises with nondeterminism. In contrast to error values, indeterminate values typically are *intentional*. An expression like `pop([])` could be an error, but it could just as well be *loosely specified*: it has a defined value (for example, `[]`), but it is left to an implementation to fix that value. (This suggests that error values can also be handled by underspecification, see Section 5.1.)

Nondeterminism is closely connected with *abstraction* [21, 22] and *refinement* [1]. An abstraction of a program or a machine can often be seen as an nondeterministic concrete execution. Many-valued logic has recently been used to improve the precision of abstractions [16, 54, 38], see Section 6 below.

3 Many-Valued Logics in Formalization of Partiality

We begin with a brief review of the kind of many-valued logics that are used in the literature for the purpose of modeling partiality. The basic idea in using three-valued logic is that atomic propositions may yield a truth value *undefined*, which we denote

with the symbol $\perp$ in this paper. Three-valued logic now provides the meaning of formulas whose atomic propositions are three-valued.

3.1 Propositional Logic

Three-valued propositional connectives are represented by disjunction in Table 1. Other types of connectives can be obtained by duality, residuation, etc. All connectives behave classically on classical truth values. Strong Kleene disjunction [43] can be defined as the join of the *truth order lattice* $F < \perp < T$. It is optimistic in the sense that if any of its arguments determines the result of the operation, then the other arguments are discarded, whether undefined or not. Kleene's motivation was to model possibly non-terminating recursive functions, where one dovetails the simultaneous computation of both arguments.

Strong Kleene

$\vee$	F	$\perp$	T
F	F	$\perp$	T
$\perp$	$\perp$	$\perp$	T
T	T	T	T

Bochvar

$\vee$	F	$\perp$	T
F	F	$\perp$	T
$\perp$	$\perp$	$\perp$	$\perp$
T	T	$\perp$	T

McCarthy

$\vee$	F	$\perp$	T
F	F	$\perp$	T
$\perp$	$\perp$	$\perp$	$\perp$
T	T	T	T

Belnap

$\vee$	F	$\perp$	T
F	F	F	T
$\perp$	F	$\perp$	T
T	T	T	T

TABLE 1. Three-valued matrices for disjunctions used for modeling partiality.

Bochvar's connectives [10] (also known as Weak Kleene) yield undefined as soon as any subexpression is undefined. This principle of evaluation is called *strict*. It is pessimistic in the sense that an undefined value that occurs anywhere in a computation compromises the result, no matter what other subcomputations may give.

A *sequential* disjunction introduced by McCarthy [49] mixes weak and strong Kleene and represents the idea that if the truth value can be determined after evaluation of the first argument, then the result is computed without looking at the second argument. Many programming languages contain operators that exhibit this kind of behavior.

Finally, the result of Belnap's [6] disjunction when computed over at least one non-classical argument is the join of the *information order semi-lattice*, where $\perp < T$, $\perp < F$. The idea here is to propagate defined values as soon as they appear in a subcomputation. In the following we denote the least upper bound in this lattice with $\sqcup$.

All four connectives have rather different justifications that seem to depend on the application at hand. The situation becomes even more complex, however, when we look at first-order logic.

3.2 First-Order Logic

Most specification languages are (at least) first-order, so that one can talk about objects and their relations. It is possible to model many properties in a purely relational way, but it is cumbersome to write, for example, $x = r \wedge \mathrm{div}(m, n, r)$ instead of simply $x = (m/n)$. Therefore, most specification languages allow *complex terms*.[2]

[2] One notable exception is Alloy [33], which trades off expressivity for simplicity and so avoids the problems discussed in this subsection.

While relations simply may not hold or may be empty, and there is no conceivable use of partially defined constants, complex terms that represent applications of partial functions may be undefined. Standard first-order logic interpretations are total on all functions. In order to model partial functions, one has to reconcile the semantics of functions and predicates, independently of whether three-valued logic is used on the propositional level or not. Therefore, almost all specification languages employ an undefined value for terms. We use the same symbol $\bot$ as before.[3]

Most languages define term evaluation to be *strict*, that is, the value of a term is $\bot$ iff the value of any subterm is $\bot$. This is even true for several languages that contain non-strict three-valued connectives (we come back to this issue later). A tricky issue is the semantics of the equality predicate, notably, when both arguments are undefined. Again, this is discussed in the examples below.

It is not much disputed to exclude $\bot$ from the *domain* of first-order models during evaluation of quantifiers. But in the case of three-valued logics, one still has to define the semantics of quantifiers when the formula in the scope evaluates to undefined at one or more locations. Unfortunately, it seems difficult to do this in a precise and simple way. In classical logic it is possible to see a universal (existential) quantifier as an infinitary conjunction (disjunction). If one generalizes this construction, then both Bochvar and strong Kleene connectives give rise to three-valued quantifiers. The first, which we call *strict* quantifiers, have the drawback that a formula such as $\forall m.(m \neq 0 \rightarrow \exists n.(m = 1/n))$ evaluates to $\bot$, simply because the existential formula is undefined at $n = 0$. On the other hand, quantifiers based on strong Kleene logics may cause higher effort in formal proofs of statements involving them. For example the shortest proof theoretic characterization of $\mathrm{val}(\forall x.\phi(x)) = \bot$ is "$\mathrm{val}(\phi(c)) = \bot$, where c is a skolem term, and $\mathrm{val}(\phi(t)) \in \{F, \bot\}$ for any ground term t" [36], which yields a duplication of ϕ as compared to the classical case.

An even more precise modeling would lift the idea behind McCarthy's connectives to the first-order level and allow one to specify an evaluation policy that matches the underlying application. One specifies an order on the semantic domain that corresponds, for example, to a given loop or iterator traversal. Then, however, we leave classical first-order logic and enter the realm of generalized quantifiers [60]. This seems not to be realized in any specification language.

3.3 Computational Logic

In practice, even first-order logic is not sufficient to model interesting properties of programs, because it is not expressive enough to characterize inductive data structures. To achieve this, perhaps the most straightforward means is to extend first-order logic with *least fixed points* [23]. There are many ways to set this up syntactically. We use λ abstraction to define predicates p from the domain D of a fixed point variable n into $\{F, T\}$ via least fixed points. To define $p : D \rightarrow \{F, T\}$, we form an expression of the form $\lambda n : D.\phi(p)$, where p occurs positively in the scope ϕ. The least fixed point formula for p is then written as $p \Leftarrow \lambda n : D.\phi(p)$. Evaluation is done as usual by approximation with ordinal powers, starting with the empty relation as the interpre-

[3] In many-valued specification languages (for example, RSL, OCL) $\bot$ denotes the same value in terms and formulas alike. This is explained by the fact that software specification languages stand closer to the programming language tradition than to the logic tradition. In the former, there is no distinction between formulas and terms, one has simply expressions with different type.

tation of p. The value of a fixed point formula is computed as the join of the values at each approximation stage in the Boolean truth lattice $F < T$.

Example 3.1
A predicate that characterizes even numbers might be defined as follows:

$$\begin{array}{lll} \text{even}: & int \to \{F,T\} & \\ \text{even} & \Leftarrow \quad \lambda n : int\,(& n = 0 \vee \\ & & n \neq 1 \wedge \exists v : int.(n = v{+}2 \wedge \text{even}(v))) \end{array} \tag{3.1}$$

Evaluating this definition gives $\text{val}(\text{even}(2)) = T$, $\text{val}(\text{even}(1)) = F$, $\text{val}(\text{even}(0)) = T$ as expected for the arguments that can be computed with a finite approximation, but it also gives $\text{val}(\text{even}(-1)) = F$. Hence, non-terminating arguments are conflated with arguments on which the predicate does not hold. ∎

In order to remedy this situation, one can use three-valued logic. Recursively defined predicates are declared three-valued, in the example even: $int \to \{F, \bot, T\}$. The definitions of predicates are left unchanged, but evaluated with three-valued semantics.

This three-valued semantics should be designed in such a way that the behavior on terminating arguments is unchanged, but the semantic value $\bot$ serves as an indicator for non-termination. In the example, we would like to have $\text{val}(\text{even}(-1)) = \bot$. In order to guide the search for an appropriate semantics we use the observation that recursive definitions of predicates such as even can be written in Prolog as follows:

```
even(0).
even(n) :- n -\- 1,
           v is n - 2,
           even(v).
```

Now, the least fixed point of the Prolog program coincides with the semantics of the fixed point formula (3.1). Three-valued semantics of Prolog with exactly the properties that we want has been studied in the 1980s [29] Hence, if one transfers the definitions of three-valued operators from Prolog to first-order logic with least fixed points, then one obtains strong Kleene semantics for propositional connectives and for quantifiers, and Belnap semantics for combining the results of the recursive calls. The idea behind the latter is to fix the truth value of an expression as soon as the evaluation reaches a defined value.

Unlike in the classical case, approximation with ordinal powers is based on predicates that are $\bot$ by default. In the example, $\text{val}(\text{even}(x)) = \bot$ for all x. More precisely, if κ is an ordinal, then $\Leftarrow^\kappa \phi$ denotes κ times unfolding of ϕ and is defined as follows (where V is the fixed point variable in ϕ):

$$\text{val}(\Leftarrow^\kappa \phi) = \begin{cases} \bot & \kappa = 0 \\ \text{val}(\phi\{V \mapsto \text{val}(\Leftarrow^\beta \phi)\}) & \kappa = \beta + 1 \\ \bigsqcup_{\beta<\kappa} \text{val}(\Leftarrow^\beta \phi) & \kappa \text{ limit ordinal} \end{cases}$$

Based on this we define fixed point semantics as $\text{val}(\Leftarrow \phi) = \bigsqcup_{\beta \in \mathcal{O}} \text{val}(\Leftarrow^\beta \phi)$, where $\mathcal{O}$ is the set of all ordinals. This semantics gives the desired result. In Fig. 1 the beginning of the (infinite) semantic evaluation tree of `even(-1)` is shown, which illustrates how only $\bot$ values are propagated to the top-level based on the Belnap/Kleene semantics.

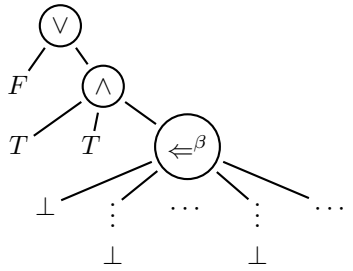

FIG. 1. Evaluation tree of `even(-1)`.

In this subsection we showed that it is possible to use a three-valued fixed point logic to characterize the situation when a subcomputation needed for formula evaluation does not terminate. The main problem with this approach is that the semantics is much more difficult to comprehend than the classical "false unless defined" approach, even though it is more accurate than the latter. We have not seen it realized in any specification language. Even in logic programming, despite its technical advantages, three-valued semantics has not caught on.

4 Many-valued Logic and Partiality in Formal Specification Languages

In Section 3 we described how many-valued logic is used to model various aspects of partiality in software specification. Pure logic, however, is rarely directly used in formal specification, because its syntax is not rich enough. Typically, logic-based specification languages feature built-in concepts for types, standard data structures (such as numbers, collections, or strings), local definitions, procedures, etc., as well as syntactic sugar. Even specification languages that have a strong relation to logic are often based on a set-theoretic semantics. For example, OCL can easily be mapped into a slight extension of first-order logic [5], the UML standard is based on a set-theoretic semantics [53]. This notwithstanding, several major specification languages feature three-valued logic, and *all* specification languages have to address the various aspects of partiality. In the rest of this section we analyze the treatment of partiality in a representative set of specification languages which have in common that they all feature undefined values and partial functions.

4.1 The Z Notation and its Derivates

The *Z* notation has been developed by the Oxford Programming Group since the mid 1970s, based on work by Jean-Raymond Abrial. A definitive version appeared in 1992 [57], and in 2002 *Z* was standardized by the ISO [40]. Two other important specification languages were strongly influenced by *Z*: the languages *B* [1], created also by Abrial, and Alloy [33]. The *B* language is not a "general purpose" specification language, but part of an integrated formal development concept. Alloy does not strive to be as expressive as other specification languages, but is intended to be simple and efficient. In particular, it is based on the relational fragment of first-order logic.

Function application is modeled by relational join (in the sense of database theory, that is, composition of relations), so the value of functions on undefined arguments is simply the empty set. The complications described in Section 3.2 need not be addressed then.

All three languages were used in substantial industrial and academic projects. The *Z* notation is a large language. It possesses a formal semantics, and the work of the (mostly academic) *Z* community had considerable impact on the field of formal methods.

Z does not use three-valued logic, rather, formulas have classical two-valued semantics. There is an undefined value $\perp$ for terms, which are evaluated according to strict semantics. Predicates containing an undefined subterm have in any case a defined (classical) truth value, but this value is unknown [57, p40f]. It is not quite clear whether this means that the semantics of *Z* is not compositional on expressions that may contain undefined subterms, or whether evaluation is simply impossible as soon as a critical expression is encountered, which would put it closer to underspecification (discussed in Section 5.1). There is a predicate *dom* available that allows one to ensure definedness of an expression provided that the undefined values can be characterized in *Z*. If a formula is supposed to have a unique value, then explicit qualification of critical expressions is necessary. The disadvantage is that it can bloat the specification. It also tends to overemphasize exceptional behavior.

*4.2 Common Algebraic Specification Language (*CASL*)*

The Common Algebraic Specification Language (CASL) is an informal standard of the algebraic specification community. Developed since the mid 1990s, a definitive version appeared recently [7, 19]. The main target of CASL is specification of abstract software designs and their requirements. It has relatively little impact outside of the algebraic specification community. CASL is based on clear mathematical concepts and comes with a formal semantics [19].

Like *Z*, CASL has classical formula semantics and admits an undefined value for terms, which are evaluated according to strict semantics. There are, however, two important differences. The first is that atomic propositions containing an undefined value are always evaluated to F. The second is an exception to that rule, namely, the equality predicate which yields T in the case when both arguments are undefined [7, p48]. This is called *strong equality* and a consequence of the design decision in CASL to model partial recursive functions as closely as possible.

This choice of semantics can have unintuitive consequences. The expression $1/0 = 0$ evaluates to F as expected, but it is less clear whether $1/0 = 2/0$ should evaluate to T. In general, equations such as $1/n = 2/n$ do hold for this single value of n, where both sides are undefined, even if they differ semantically for all other values.

Since propositions containing an undefined value evaluate to F, care must be taken with predicate definitions. For example, if $<$ is intended to be a total order, then the totality axiom for $<$ must be restricted to defined arguments. This can be seen, for example, with $1/0 \geq 2/0$ and $1/0 < 2/0$ which evaluate both to F, because they contain undefined arguments. Strict evaluation stipulates then T for $\neg(1/0 < 2/0)$. If the totality axiom is not relativized by restricting the arguments of $<$ to defined values, then one can immediately derive a contradiction. As a consequence, in order

to avoid unintended overspecification one needs to restrict *all* predicate definitions to defined values.

4.3 VDM-SL and RAISE Specification Language (RSL)

The Specification Language of the Vienna Definition Method (VDM-SL), developed since the early 1970s, was the first major effort to design a specification language being as systematic and usable as a programming language, and as such was very influential. Important versions of VDM-SL were issued in [8, 24], and in 1996 it was standardized by the ISO [39]. A formal semantics is not part of the definition.

The Specification Language of the Rigorous Approach to Industrial Software Engineering (RAISE) project is called RSL and can be seen as an attempt to address certain shortcomings and limitations in VDM-SL. Development started around 1987, and a definitive version appeared in [30, 31]. Unlike VDM-SL, RSL has a formal semantics [50], although the manuals feature only an axiomatic semantics in the form of a collection of *proof rules* that implicitly determine the derivable specifications.

Together, VDM-SL and RSL constitute over 30 years of experience in designing formal specification languages. Both methods have been extensively applied in industrial case studies.

For our analysis it is interesting that both languages have an undefined value for terms *and* formulas. In the following we concentrate on RSL. There, the undefined value is called **chaos**. The value **chaos** is outside of the normal type system, for example, it is not a value of the set **Bool**, but the proof rules implicitly specify how expressions that contain **chaos** values are to be evaluated. Propositional connectives are evaluated according to McCarthy semantics, and terms are evaluated according to strict semantics. In some cases, it is not easy to find the relevant proof rule, for example, evaluation of quantifiers on **chaos** is not obvious from [30].

Like in Z, undefined values are used in RSL to model non-termination as well as error values, but there is also a third use: unbounded nondeterminism, such as choosing an arbitrary element from an infinite set, results in **chaos** as well. In contrast to this, bounded nondeterminism lets the result of an evaluation depend on the evaluation strategy of an nondeterministic operator and results in a non-compositional semantics.

4.4 Object Constraint Language (OCL)

The Object Constraint Language (OCL) is part of the Unified Modeling Language (UML). As such it has been standardized by the Object Management Group (OMG). Development of OCL started around 1995 and it was included for the first time in the UML 1.1 standard [51], the latest version being 2.0 (partly described in [59]). Since version 2.0, a formal (set theoretic) semantics is part of the OCL standard. The language is strongly typed.

The main motivation for OCL was the need to make the semantics of UML metamodels more precise, and the first application was to write constraints for UML metamodels that would guarantee well-formed UML models as instances of the metamodel. The recent adoption of Model Driven Architecture (MDA) by the OMG gives OCL a boost, because it requires a language that is more expressive than the diagram-

matic parts of UML. Potentially, OCL has higher visibility in mainstream software development than any other formal specification language before.

Compared to other specification languages, OCL is small. In fact, it was a design goal that the language is easy to read and to learn even for people not familiar with formal notations. The concrete syntax allows postfix/dot notation for function application as it is familiar from object-oriented programming languages. This convention is even in place for operations on collections and sets of objects, where an arrow `->` is used instead of a dot. For example, the expression

```
Set{1,2} -> any(n|P(n))
```

with type integer selects any number from $\{1, 2\}$ such that the boolean predicate P is satisfied. Assume that $P(n)$ is `n>3`, then according to OCL semantics, the result is undefined.

Any complex OCL expression can contain undefined values, so the semantics uses undefined values throughout (like in other specification languages, the difference between formulas and terms in OCL is merely their type). The rules are as follows:

- All operators with the exception of certain boolean ones are evaluated according to strict semantics. In particular, equality is undefined whenever one of its arguments is undefined, therefore "`undefined = undefined`" yields `undefined`.
- Boolean connectives `or`, `not`, `and`, `implies` are evaluated according to strong Kleene semantics, but not `xor`, which is strict like `=`.
- There is an `if-then-else` operator that is evaluated sequentially (first on the condition). Hence, "`if True then True else undefined`" gives `True`, on the other hand "`if undefined then True else True`" gives `undefined`.
- The semantics of quantifiers is not directly defined in [59] and leaves room for two interpretations: since quantifiers are not listed as exceptions to strict evaluation, they might be considered as strict; in this case, the value of the expression "`Set{0,1} -> exists(n|1/n=1)`" is undefined, which seems quite odd. Another interpretation rests on the possibility to define quantifiers such as `exists(n|p(n))` via the `iterate` construct:

  ```
  iterate(n; result: Boolean = False | result or p(n))
  ```

  This would result in a strong Kleene quantifier semantics and the expression above then yields `True`.

Like other specification languages, OCL behaves non-compositionally in the presence of nondeterminism.[4] For example, the result of "`(Set0,1 -> any(n|True)) < 1`" can be either `True` or `False`.

In summary, OCL's three-valued semantics is a mix of strict, strong Kleene, and McCarthy semantics. It is not always clearly motivated or even defined (the whole discussion of undefined values in [59] comprises half a page). The result of an evaluation is not straightforward to interpret and can be surprising. One can observe that practitioners who write OCL constraints tend to ignore the many-valued aspects.

[4] In OCL, all nondeterminism is bound, because only finite objects can be constructed as arguments.

4.5 Analysis

Let us now attempt an analysis of the preceding case studies. We start by noting:

All formal specification languages that we looked at so far are semantically evaluated with an explicit undefined value. The main difference was whether this holds for all expressions or only for terms. We claim that in both cases, *explicit undefined values create more problems than they solve.* Let us look at the two possible setups in turn:

Semantics has undefined values for all expressions: we note that three-valued logic *sometimes* simplifies the formulation of properties, but the particular logics that are used *always* complicate the semantics, at least for the languages we have looked at. In order to explain the validity status of a specification, the user needs to know non-standard formula semantics. The OCL semantics is even more complicated than the solution in RSL. As shown in Section 3.2, modeling with first-order many-valued quantifiers has limitations. In all languages that we investigated, definitions of three-valued constructs at the very least are in need of clarification and better motivation. With the current state of affairs, usability is compromised and, based on our experience with OCL, authors of specifications tend to ignore many-valuedness. As one practitioner remarks: "All but the most expert readers will be ill-served by formal expositions which make use of devious tricks" [3].

Classical formula semantics[5],but strict semantics for undefined terms: the problems of formulas with non-standard semantics are avoided at the price that one must coerce undefined-valued term semantics into two-valued atomic propositions. This can be done either with a non-compositional semantics (as in *Z*) or with a false-by-default rule (as in CASL). We think that certain aspects of partiality indeed have a non-compositional nature, but would suggest a different setup detailed in Section 5.2. The false-by-default rule creates many practical problems, in particular, in connection with strong equality (see Section 4.2). It also involves a serious loss of precision when moving from the term to the formula level.

In both versions, definedness predicates are needed in practice frequently to avoid overspecification, and this can bloat specifications.

A further problem of strict term semantics: one tends to use terms for specifying operators that are close to implementational features, but these are often not evaluated strictly, as for example, JAVA's conditional Boolean operators (`&&`, etc). This would suggest sequential rather than strict semantics for terms.

Either version does not achieve the level of precision that is required in practice. This is because in all languages that we investigated, the undefined value is used undiscerningly for two or even all three different aspects of partiality listed in Section 2. But in real applications the source of undefinedness matters very much. Of course, one can introduce different undefined values, but then the semantics becomes even more complex. In addition, none of the languages we checked accommodated many-valuedness in inductive definitions as suggested in Section 3.3.

In conclusion, many-valuedness as found in contemporary specification languages, whether only for terms or also for formulas, seems not the right tool for modeling

[5] Note that in this setup there is a semantical difference between formulas and terms.

partiality in a simple, yet precise way.[6] Whether it is possible to come up with a useful three-valued semantics that is simple to work with, admittedly is an open question. Given that so much time and effort have been spent without convincing results, one may have doubts.

5 Modeling Partiality with Two-Valued Semantics

We criticized many-valued approaches to modeling partiality, but it is clear that partiality must be principally addressed. Next to many-valued logics, partiality has been modeled with intuitionistic logic, non-monotonic logic, plus a number of dedicated logics [9]. These approaches are mainly targeted at applications in philosophy, linguistics, and mathematics. Besides, the resulting logics are even more difficult to use than many-valued logics. Therefore, we turn to an alternative that is very close to classical logic.

5.1 Underspecification

The *Larch* family of specification languages [34] avoids using many-valued semantics, as does the proposal by Gries & Schneider [32] (who trace the idea back to [20]). The trick is to use *underspecification*, that is, whenever an "undefined" term such as `1/0` occurs, it is given a definite, but unknown value from its domain.[7] All functions are total just as in classical logic.

Let us now give a semantic characterization of validity in such a logic, which is more precise than the one in [32]. It is best to phrase it in the framework of first-order *interpreted reasoning*. In an interpreted first-order model (D, I) the interpretation of function and predicate symbols by I is not arbitrary, but underlies certain restrictions. Perhaps the most common example are equality models, where the equality predicate is interpreted as equality on D. It is common to have a mix of *interpreted symbols* (for arithmetic, set theory, etc.) and *free symbols*, whose semantics can optionally be restricted by adding axioms for them. The peculiarity of underspecification is that some function symbols are at the same time interpreted (on the part of the domain where they are defined) and free (on the part of the domain where they are not defined, that is, underspecified).

There are now several possibilities to define the semantics of first-order logic with underspecification that differ at the exact spot, where we choose to plug in the information on when a function is not defined. First, one can equip models with *partially interpreted* function symbols, for example, "/" is interpreted as division exactly on those argument values where it is defined and nothing is stipulated for other values. Then all other semantic notions, such as truth and validity, can be kept. While simple, this setup does not allow to characterize formulas that are true in a model up to underspecified arguments. To do this requires a little more work:

Assume that $I(f) : D_f \to R_f$ is underspecified on $U_f \subseteq D_f$.[8] We consider U_f to be

[6] We stress that our analysis does not necessarily apply to modeling partiality in natural language or general mathematics.

[7] This is exactly the opposite position of CASL [7, p48]: "For instance, the (value of the) term choose(empty) may be undefined. This is more natural than insisting that choose(empty) has to denote some arbitrary but fixed element of Elem."

[8] In general, U_f could be an undecidable set, but in typical applications it is decidable and can be easily charac-

part of the signature of f. For example, let $I(/)$ be a total function on int $\times$ int that is underspecified on int $\times \{0\} \subseteq$ int $\times$ int. A *choice function* for f is a total function $c_f : U_f \to R_f$. Let C be a tuple of choice functions c_f, one for each function f in the signature of ϕ. Now we define *truth* in a model I of a formula ϕ with respect to C: all definitions with the exception of term evaluation are standard and simply ignore C. The evaluation of terms in I with respect to C is as follows:

$$\mathrm{val}_{I,C}(f(t_1,\ldots,t_n)) = \begin{cases} I(f)(\mathrm{val}_{I,C}(t_1),\ldots,\mathrm{val}_{I,C}(t_n)) \\ \quad \text{if } \langle \mathrm{val}_{I,C}(t_1),\ldots,\mathrm{val}_{I,C}(t_n)\rangle \in (D_f - U_f) \\ c_f(\mathrm{val}_{I,C}(t_1),\ldots,\mathrm{val}_{I,C}(t_n)) \quad \text{otherwise} \end{cases}$$

A formula ϕ is *true* in a model I if it is true in I with respect to all possible tuples of choice functions C. All other semantic notions are as usual.

For example, $\exists n.(1/0 = n)$ is true in $I(/)$ from above, because for each possible choice function $c_/$ it is possible to find a witness element, namely $c_/(1,0)$. On the other hand, $\mathrm{val}_{I,\langle c_/\rangle}(1/0 = 0)$, $\mathrm{val}_{I,\langle c_/\rangle}(1/0 = 1)$, etc., give false for almost all choice functions $c_/$.[9]

Proof theoretically, this logic is very easy to handle. It is sufficient to protect rewrite rules for underspecified functions with appropriate domain restrictions. For example, the equation $(-x)/y = -(x/y)$ must be guarded against y being 0, because it is invalid in this case. One could write, for example, $\forall x, y.(y \neq 0 \to (-x)/y = -(x/y))$.

It is a major advantage of the underspecification approach to partiality that all classical inference patterns are valid. In particular, the law of excluded middle still holds. In addition, all axioms involving only total (that is, fully specified) functions and predicates are valid without any restriction (in contrast to the situation described at the end of 4.2). For example, the axiom of totality for an ordering predicate $\leq$ is exactly as usual, because it contains no underspecified functions.

Mechanical deduction for the resulting logic is as easy as for classical logic, which is very important for tool support. Underspecification has been implemented in several systems including [2, 3].

The main critique against underspecification is, ironically, that it can lead to *overspecification* in certain situations. It is completely intended that, for example, $\exists n.(1/0 = n)$ is valid (see above). This simply reflects the fact all functions are total, and it is harmless as such. One must, however, be more careful with recursive definitions. Jones [42] pointed out that, for example, the "even" predicate (3.1) has a defined (although unknown) value for even(-1) and, hence, even(-1) = even$(-3) = \cdots$. While this is still harmless, [47] gave a similar example that causes inconsistency by forcing the interpretation of a predicate at an underspecified argument to be smaller than any negative integer. In general, recursive definitions must be protected against undefined arguments in order to avoid unintended overspecification. For example, the recursive definition of the even predicate (3.1) could be patched as follows:

$$\begin{array}{lll} \text{even} & \Leftarrow & \lambda n : int\,(\quad n = 0 \vee \\ & & \qquad n \neq 1 \wedge n > 0 \wedge \exists v : int.(n = v{+}2 \wedge \text{even}(v))) \end{array}$$

terized by suitable axioms.

[9] The valuation of underspecified terms is very similar to valuation of formulas with free variables using variable assignments in the usual way.

But underspecification constitutes only one half of the strategy used in recent approaches to behavioral specification of programs.

5.2 Behavioral Specification

In behavioral specification one specifies the *behavior* of concrete or abstract programs by stating *pre-* and *postconditions* that characterize how they transform program states (sets of variables).

Behavioral specification allows to *protect* [47] postconditions by adding a precondition that excludes those states that cause non-termination or undefined values. We agree with [3] who states that "in our experience, most real-life specifications that do make essential use of undefined terms are just wrong—they do not say what their author intended." In other words, non-termination and error values are mainly *unintended* behavior of implementations. They should be handled as far as possible on that level. A program that raises a run-time exception for certain inputs should not be modeled with a specification containing partially defined expressions. Rather, the specification should characterize the inputs that cause normal, respectively, exceptional behavior and, in each case, specify the resulting behavior with totally defined expressions.

We clarify this idea using the Java Modeling Language (JML), developed since 1997 by Gary Leavens, but in the meantime a community effort. JML is strongly influenced by Larch [34], and is based on classical logic. Its syntax is derived from side-effect free Java expressions. A formal semantics is currently not available for JML. The development is still in flux (recent snapshots are provided in [45, 46]), but there is a large number of partial implementations in analysis and verification tools [14].

If undefined expressions occur, JML suggests underspecification to deal with them [45, p58f]. (In contrast to this, an implementation of a JML runtime assertion checker [17, Sect. 3.2] suggests a rather involved semantics on terms with explicit undefined values that evaluates the smallest containing Boolean expression either to F or to T, depending on the nature of the undefined value.) In JML, different clauses specify normal and exceptional behavior of programs. The correctness notion of JML includes that programs must terminate when called in a state that satisfies the precondition.

Example 5.1
The "even" predicate (3.1) specified as a Java method with JML might look like this:

```
/*@  public normal_behavior
  @     requires n>=0;
  @     ensures \result == (n div 2 == 0);
  @ also
  @  public exceptional_behavior
  @     requires n<0;
  @     signals (IllegalArgumentException); @*/
public boolean even(int n);
```

Non-termination is simply unintended, incorrect behavior and an implementation is expected to return an exception for arguments, where "even" is undefined. ∎

The behavioral style of specification is supported by *model oriented* specification languages such as Z and VDM-SL, but it can be useful to look at logical languages.

In program logics that permit actual correctness proofs, one can go one step further. From a logical point of view behavioral specification languages and Hoare logic can be seen as an instance of *dynamic logic* [37]. In dynamic logic, each program `p` gives rise to multi-modal formulas $[\mathtt{p}]\phi$ (respectively, $\langle\mathtt{p}\rangle\phi$) which state that `p` is partially (totally) correct with respect to a postcondition ϕ.

Such a logic was realized in the KeY tool [2] for the target programming language Java Card [58]. In this setting one may state partial correctness of "even" by

$$\forall n : \text{int}.\,([\texttt{even}(\text{n});]\,(\text{result} \leftrightarrow (\text{div}(n, 2) = 0))) \quad .$$

The meaning of this formula is: "for all inputs n, if `even`(n) terminates normally (without exception), then its result is 'true' iff $\text{div}(n, 2) = 0$ holds." The partial correctness modality makes a claim only for terminating runs. Whenever `even`(n) terminates n must be non-negative, so the expression in the postcondition is defined. In this way, the partial correctness modality automatically protects against underspecification in the postcondition.

Similarly, the operational semantics of programs embodied in the calculus used for program verification protect specifications in postconditions against undefined values. Assume, for example, that during a verification proof the subgoal formula $\langle i = 1/i;\rangle\phi$ is encountered. Its meaning is "the assignment `i = 1/i` terminates normally and afterwards the formula ϕ holds." Then the program logic's calculus rules for assignment and arithmetic operators generate two new subgoals (simplified):

1. $i \neq 0 \rightarrow \phi\{i \mapsto \text{div}(1, i)\}$
2. $i = 0 \rightarrow \langle\texttt{throw new ArithmeticException}();\rangle\phi$

The calculus rule that characterizes the operational semantics of Java's division operator creates a case distinction according to whether the second argument is null and throws an exception if this happens to be so. Whether the two subgoals can be proven depends on the postcondition ϕ and on whether the exception is caught, but in no case an undefined logical term is generated. In particular, the expression $\text{div}(1, i)$ is protected by the guard $i \neq 0$.

The only way that undefined terms can be introduced (if not already present in the postcondition) is via quantifier elimination and other substitution rules such as equality. In the few cases, where undefined terms occur, they are handled by underspecification.

The separation of programs and specifications by modalities or other program logic constructs has also the advantage that unintentional overspecification [47] is not likely to occur provided that the specification language is not too expressive.

6 Abstraction

In the previous section we advocated program logics and underspecification for modeling non-termination and error values, in particular, for verification purposes. We did not discuss nondeterminism, refinement, and abstraction so far. Most model oriented formal specification languages, including *Z*, *B*, and VDM-SL, have support for a (language-specific) refinement of specifications. In principle, one has two specifications, an abstraction A and an implementation I, plus some kind of conservation

relation that guarantees that the intended models of A are also models of I. Hence, the important relation is $A \models I$. The specification frameworks define necessary and usually mechanically checkable conditions for this relation to hold.

In abstract interpretation [21], however, the opposite direction of this relation is of interest. The starting point is a system I with a state space that is too large[10] for current verification technology. An *abstraction* a maps I into a system with less states in such a way that valid properties of $a(I)$ are still valid for I. The abstraction $a(I)$ is often expressed in a less expressive language than I is, but not necessarily. Each proper abstraction loses information in the sense that there are properties valid in I, but not in $a(I)$. A proper abstraction is thus always an *approximation*.

In model checking [18] one can only handle finite state spaces, so abstraction is unavoidable when I has infinitely many states, but even finite-state systems must often be abstracted in order to become feasible. Until recently, two-valued logics and models were used in abstract model checking, with simulation as the abstraction relation. The need for a third value arises when checking non-universal properties or, more generally, properties expressed in temporal logics closed under negation [12]. Simulation is not sufficient to express this.

In order to model partial state spaces one introduces a third truth value understood as "unknown" on the level of propositions (many-valued terms do not suffice). A typical example is the notion of *state-wise preservation* in model checking [22]: here, a system I is a finite Kripke structure with propositional signature Σ and states $s_I : \Sigma \to \{F, T\}$; properties about I are written in temporal logic. Denote with $\|\phi\|_I(s_I) \in \{F, T\}$ whether ϕ holds in s_I or not. A canonical abstraction a identifies those states in I that are indistinguishable, that is, they have the same valuation and are connected to the same predecessors and successors. This abstraction does not lose any valid properties, but it is very restrictive. A much more common situation is that two states s'_I, s''_I, are indistinguishable with exception of propositions $p_1, \ldots, p_n$, that is, $s'_I(p_i) \neq s''_I(p_i)$ for $1 \leq i \leq n$ and $s'_I(p) = s''_I(p)$ on any other p. If one identifies $\{s'_I, s''_I\}$ with s_A in the abstract structure A, then some defined value must be assigned to $s_A(p_i)$. This is where many-valued logic enters the picture. We may set $s_A(p_i) = \bot$ for $1 \leq i \leq n$, and evaluate $\|\phi\|_A(s_A)$ in strong Kleene logic. As a consequence, one has $\|\phi\|_A(s_A) = \bot$ whenever there are concrete s'_I, s''_I such that $\|\phi\|_I(s'_I) \neq \|\phi\|_I(s''_I)$.

The point is that many-valued logic offers a *compositional* approximation of concrete behavior without loss of precision. This situation is ubiquitous in formal verification, abstract model checking [12, 16], and static analysis [54]. We illustrate this with a brief example: consider a finite state machine model of an elevator button. Each state has boolean values for the variables *ButtonPressed*, *LightOn*, and *Reset*. The first two are self-explanatory, the third is true iff the button has been reset. In a typical specification of the button behaviour, once the button is pressed in a reset state, the lightOn-set-LightOff-reset sequence is identical whether the button is pressed again or not. Using three-valued logic, where $\bot$ is assigned to *ButtonPressed* preserves all essential safety properties as compared to the two-valued formulation, but results in a considerably smaller state space. This is, because several states that are identical up to the value of *ButtonPressed* can be conflated.

[10] If I is denoted in a first-order language, the state space is generally infinite, but as the number of states of a system is exponential in the number of its variables, even finite systems are often too large in practice.

Regarding implementation of many-valued model checking, some authors advocate many-valued reasoning [15], while others suggest to translate many-valued to two-valued logic [44]. In either case, the modeling language is many-valued.

In the paper [38] a class of lattice-based logics is suggested for setting up abstraction with improved precision for static analysis and verification. Here we suggest a somewhat different class of logics, also motivated by abstraction.

The idea of state-wise preservation sketched above suggests to interpret abstract truth values as the *union* of concrete truth values:

Union of concrete values		Abstract value
$\{T\}$	$\longleftrightarrow$	T
$\{F,T\}$	$\longleftrightarrow$	$\bot$
$\{F\}$	$\longleftrightarrow$	F

If we order the abstract values as $\{F\} < \{F,T\} < \{T\}$, then the lattice meet and join give exactly the intended evaluation of conjunction and disjunction (strong Kleene logic).

Generalizing this situation we can view abstract truth values as representing a *collection* of concrete truth values. Collections are commonly implemented as sets, multisets, or sequences. Taking sets gives strong Kleene logic as has just been shown. Now let us go from sets to multisets of truth values. A concrete application scenario might be as follows: assume one makes *experiments*, where one can test for the presence of a negative or of a positive feature. An experiment may not be conclusive. Assume further that at most p affirmative and n negative experiments are performed. In the abstract space we want to compute the support for a logical combination of features. A truth value is now a multiset $m : \{F,T\} \to \mathbb{N}$, where $m(F) \in \{0,\ldots,n\}$ and $m(T) \in \{0,\ldots,p\}$. A truth value m_1 is smaller than m_2 iff m_2 has stronger support for truth than m_1 and less support for falsehood. This induces a lattice with the following join (disjunction):

$$(m_1 \sqcup m_2)(T) = \max\{m_1(T), m_2(T)\} \qquad (m_1 \sqcup m_2)(F) = \min\{m_1(F), m_2(F)\} \quad .$$

The lattice has $(p+1)(n+1)$ elements, the least element $\bot$ is $\bot(T) = 0, \bot(F) = n$ and the greatest element $\top$ is $\top(T) = p, \top(F) = 0$. For the special case $n = p = 1$ and non-empty multisets, the three-valued abstraction based on set union is obtained. The special case $n = p$ has been investigated under the name SHn-logic in the literature [41, 55]. It would also be interesting to follow the connection to the theory of bilattices [28].

A further refinement can be obtained by taking finite *sequences* of classical interpretations. Abstract truth values are then from $\{F,T\}^n$. The componentwise order gives rise to *product logics* and has a structure that is isomorphic to the n-valued set lattice. An abstract truth value can be interpreted as modeling a collection of *agents*. This logic was used in [26] for $n = 2$.

We conclude that a family of many-valued logics can be naturally motivated for applications in abstraction. These logics are based on certain distributive lattices. This is important for two reasons: first, it is obvious how to lift these logics to the first-order case; second, there are particularly efficient automated reasoning techniques available for propositional [56] and first-order logics [36] based on distributive lattices. There are also efficient model checking techniques for multi-valued temporal logics [13].

We add that it is possible to handle even *nondeterministic* behavior within abstraction. Instead of resorting to a non-compositional semantics as in OCL (see Section 4.4), one could approximate nondeterministic predicates with strong Kleene logic as pointed out above.

7 Conclusion

In this paper we analysed which role many-valued logic should play in formal specification of software, which is often mentioned as one of its major applications.

We gave an overview on how many-valued semantics is used in a representative selection of formal specification languages to model various aspects of partiality, where we clearly differentiate between non-termination, error values, and non-determinism/ abstraction. In all but one of the investigated specification languages, only non-termination or error values are addressed. In no case, different truth values are introduced for different purposes.

Multiple values are essentially used to emulate partial functions within classical logic, which is based on total functions. We exhibited numerous problems, including semantical complexity, unintuitive semantics, semantic inadequacy of quantifiers, lack of justification, danger of overspecification. At the same time, definedness predicates cannot be avoided.

We argued that at least for the specification of structured software, two-valued approaches based on underspecification of total functions are superior. To this end, we clarified the semantics of underspecification in the presence of interpreted function symbols. We also argued that in behavioral specification with program logics one can get rid of most partial definitions without explicit protection conditions. We stress that our analysis does not necessarily apply to modeling partiality in natural language or general mathematics.

Finally, we suggested that modern specification languages should not only support refinement, but also its counterpart abstraction, which includes nondeterminism. In contrast to partially defined functions it is absolutely crucial to have additional truth values on the level of *propositions* for modeling conflicting behavior of concrete systems. In this context one can motivate a family of lattice-based many-valued logics. Strong Kleene logic is obtained in the most simple case and all logics extend to the first-order case. We emphasize that many-valued logics offer a compositional approximation of concrete or nondeterministic behavior without loss of precision.

To summarize, we think that many-valued logic is largely obsolete for modeling those aspects of partiality, where it was traditionally thought be useful; on the other hand, many-valued logic is emerging as a mainstream tool in formal analyses of various kinds, and specification languages should feature many-valued abstraction logics.

Acknowledgments

I thank Agata Ciabattoni and Matthias Baaz for the invitation to the ESF Exploratory Workshop on *The Challenge of Semantics*, where these ideas were first presented. Discussions with Martin Giese, Steffen Schlager, and Peter Schmitt were very helpful. Andreas Roth supplied important references. Wolfgang Ahrendt and Martin Giese thoroughly read draft versions and made numerous suggestions for improvement. The

comments of the anonymous reviewers helped to make the paper much clearer.

References

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool: integrating object oriented design and formal verification. *Software and System Modeling*, 4(1):32–54, 2005.

[3] R. D. Arthan. Undefinedness in Z: Issues for specification and proof. In William Farmer, Manfred Kerber, and Michael Kohlhase, editors, *Proc. Mechanization of Partial Functions Workshop, affiliated to CADE-13, New Brunswick*, pages 3–12, 1996.

[4] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[5] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002.

[6] Nuel D. Belnap Jr. A useful four-valued logic. In J. Micheal Dunn and George Epstein, editors, *Modern uses of multiple-valued logic*, pages 8–37. Reidel, Dordrecht, 1977.

[7] Michel Bidoit and Peter D. Mosses. CASL *User Manual*, volume 2900 (IFIP Series) of *LNCS*. Springer-Verlag, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[8] Dines Bjørner. The Vienna development method (VDM): Software specification and program synthesis. In M. Paul, E. K. Blum, and S. Takasu, editors, *Proc. International Conference on Mathematical Studies of Information Processing, Kyoto, Japan*, volume 75 of *LNCS*, pages 326–359. Springer-Verlag, August 1978.

[9] Stephen Blamey. Partial logic. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 4, pages 261–353. Kluwer, Dordrecht, 2nd edition, 2002.

[10] D.A. Bochvar. On a three-valued logical calculus and its applications to the analysis of the paradoxes of the classical extended functional calculus (in russian). *Matématičeskij Sbornik*, 4:287–308, 1939. Translated by Merrie Bergman, History and Philosophy of Logic 2, (1981), 87–112.

[11] Viviana Bono and Manfred Kerber. Crash in program and logic. In Gethin Norman, Marta Kwiatkowska, and Dimitar Guelev, editors, *Proc. AVoCS: Automated Verification of Critical Systems*, School of Computer Science, The University of Birmingham, 2002. Tech Report TR CSR-02-6.

[12] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In Nicolas Halbwachs and Doron Peled, editors, *Proc. 11th International Computer Aided Verification Conference*, volume 1633 of *LNCS*, pages 274–287. Springer-Verlag, 1999.

[13] Glenn Bruns and Patrice Godefroid. Model checking with multi-valued logics. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland*, volume 3142 of *LNCS*, pages 281–293. Springer-Verlag, 2004.

[14] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005. Online First.

[15] Marsha Chechik, B. Devereux, and Steve Easterbrook. Implementing a multi-valued symbolic model checker. In *Proc. Fourth European Joint Conferences on Theory and Practice of Software (ETAPS): Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Genova, Italy*, LNCS. Springer-Verlag, 2001.

[16] Marsha Chechik, Steve Easterbrook, and Victor Petrovykh. Model-checking over multi-valued logics. In *Proc. Formal Methods Europe, Berlin, Germany*, LNCS. Springer-Verlag, 2001.

[17] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, April 2003. The author's Ph.D. dissertation. Available from `archives.cs.iastate.edu`.

[18] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[19] CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2900 (IFIP Series) of *LNCS*. Springer-Verlag, 2004.

[20] Robert L. Constable and Michael J. O'Donnell. *A Programming Logic, with an Introduction to the PL/CV Verifier*. Winthrop Publishers, 1978.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Language, Los Angeles*, pages 238–252. ACM Press, New York, January 1977.

[22] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[23] Anuj Dawar and Yuri Gurevich. Fixed point logics. *The Bulletin of Symbolic Logic*, 8(1):65–88, 2002.

[24] John Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[25] Marie Duží. Do we have to deal with partiality? *Miscellania Logica*, Tom V:45–76, 2003.

[26] Steve Easterbrook and Marsha Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *Proc. 23rd International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press, May 2001.

[27] William M. Farmer. A partial functions version of Church's simple theory of types. *The Journal of Symbolic Logic*, 55(3):1269–1291, September 1990.

[28] Melvin Fitting. Bilattices are nice things. In *Proc. PhiLog Conference on Self-Reference, Copenhagen*. The Danish Network for Philosophical Logic and Its Applications, 2002.

[29] Melvin C. Fitting. A Kripke-Kleene semantics for logic programming. *Journal of Logic Programming*, 4:295–312, 1985.

[30] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The Raise Development Method*. Prentice Hall, London, 1995.

[31] Chris George, Peter Haff, Klaus Havelund, Anne E. Haxthausen, Robert Milne, Claus Bendix Nielson, Søren Prehn, and Kim Ritter Wagner. *The Raise Specification Language*. Prentice Hall, New York, 1992.

[32] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, New York, NY, 1995.

[33] Software Design Group. *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*, February 2002. Draft.

[34] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and Jeanette M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, 1993.

[35] Reiner Hähnle. *Automated Deduction in Multiple-Valued Logics*, volume 10 of *International Series of Monographs on Computer Science*. Oxford University Press, 1994.

[36] Reiner Hähnle. Commodious axiomatization of quantifiers in multiple-valued logic. *Studia Logica*, 61(1):101–121, 1998. Special Issue on Many-Valued Logics, their Proof Theory and Algebras.

[37] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, October 2000.

[38] Michael Huth and Shekhar Pradhan. Consistent partial model checking. In *Proc. Workshop Domains VI, September 16–19, 2002, Birmingham*, volume 73, to appear, 2004.

[39] International Organisation for Standardization. *Information technology—Programming languages, their environments and system software interfaces—Vienna Development Method—Specification Language—Part 1: Base language*, December 1996. ISO/IEC 13817-1.

[40] International Organisation for Standardization. *Information technology—Z Formal Specification Notation—Syntax, Type System and Semantics*, 2000. ISO/IEC 13568:2002.

[41] Luisa Iturrioz. Operators on symmetrical Heyting algebras. In T. Traczyk, editor, *Universal Algebra and Applications*, volume 9 of *Banach Center Publications*, pages 289–303. PWN–Polish Scientific Publishers, 1982.

[42] Cliff B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, April 1995.

[43] S.C. Kleene. On a notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.

[44] Beata Konikowska and Wojciech Penczek. Model-checking for multi-valued computation tree logics. In Melvin Fitting and Ewa Orłowska, editors, *Beyond Two: Theory and Applications of Multiple-Valued Logic*, volume 114 of *Studies in Fuzziness and Soft Computing*, pages 193–210. Physica-Verlag, 2003.

[45] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Iowa State University, Department of Computer Science, 2003. Revised June 2004.

[46] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry. *JML Reference Manual*, November 2004. Draft revision 1.98.

[47] Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10(1):59–75, 1998.

[48] Grzegorz Malinowski. *Many-Valued Logics*, volume 25 of *Oxford Logic Guides*. Oxford University Press, 1993.

[49] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–69. North Holland, 1963.

[50] Robert Milne. Semantic foundations of RSL. Technical Report RAISE/STC/REM/11, STC/STL, Harlow, UK, March 1990.

[51] Object Modeling Group. *Object Constraint Language Specification, version 1.1*, September 1997.

[52] Giovanni Panti. Multi-valued logics. In Dov Gabbay and Philippe Smets, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 1: Quantified Representation of Uncertainty and Imprecision, chapter 2, pages 25–74. Kluwer, Dordrecht, October 1998.

[53] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[54] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[55] Viorica Sofronie-Stokkermans. Priestley duality for SHn-algebras and applications to the study of Kripke-style models for SHn-logics. *Multiple-Valued Logic. An International Journal*, 5(4):281–305, 2000.

[56] Viorica Sofronie-Stokkermans. Automated theorem proving by resolution for finitely-valued logics based on distributive lattices with operators. *Multiple-Valued Logic*, 6(3–4):289–344, 2001.

[57] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[58] Sun Microsystems, Inc., Palo Alto/CA. *Java Card 2.0 Language Subset and Virtual Machine Specification*, October 1997.

[59] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, August 2003.

[60] Dag Westerståhl. Quantifiers in formal and natural languages. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. IV: Topics in the Philosophy of Language*, chapter 1, pages 1–131. Reidel, Dordrecht, 1989.

Received June 13, 2005.