Preventing Arithmetic Overflows in Alloy

Aleksandar Milicevic and Daniel Jackson

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{aleks, dnj}@csail.mit.edu

Abstract. In a bounded analysis, arithmetic operators become partial, and a different semantics becomes necessary. One approach, mimicking programming languages, is for overflow to result in wrap-around. Although easy to implement, wrap-around produces unexpected counterexamples that do not correspond to cases that would arise in the unbounded setting. This paper describes a new approach, implemented in the latest version of the Alloy Analyzer, in which instances that would involve overflow are suppressed, and consequently, spurious counterexamples are eliminated. The key idea is to interpret quantifiers so that bound variables range only over values that do not cause overflow.

1 Introduction

A popular approach to the analysis of undecidable logics artificially bounds the universe, making a finite search possible. In model checking, the bounds may be imposed by setting parameters at analysis time, or even hardcoded into the system description. The Alloy Analyzer [1] is a model finder for the Alloy language that follows this approach, with the user providing a 'scope' for an analysis command that sets the number of elements for each basic type.

Such an analysis is not sound with respect to proof; just because a counterexample is not found (in a given scope) does not mean that no counterexample exists (in a larger scope). But it is generally sound with respect to counterexamples: if a counterexample is found, the putative theorem does not hold.

The soundness of Alloy's counterexamples is a consequence of the fact that the interpretation of a formula in a particular scope is always a valid interpretation for the unbounded model. There is no special semantics for interpreting formulas in the bounded case. This is possible because the relational operators are closed, in the sense that if two relations draw their elements from a given universe of atoms, then any relation formed from them (for example, by union, intersection, composition, and so on) can be expressed with the same universe.

Arithmetic operators, in contrast, are not closed. For example, the sum of two integers drawn from a given range may fall outside that range. So the arithmetic operators, when interpreted in a bounded context, appear to be partial and not total functions, and call for special treatment. One might therefore consider applying the standard strategies that have been developed for handling logics of partial functions.

A common strategy is to make the operators total functions by selecting appropriate values when the function is applied out of domain. In some logics (e.g. [9]) the value is left undetermined, but this approach is not easily implemented in a search-based model finder. Alternatively, the value can be determined. In the previous version of the Alloy Analyzer, arithmetic operators were totalized in this way by giving them wrap-around semantics, so that the smallest negative integer is regarded as the successor of the largest positive integer. This matches the semantics in some programming languages (e.g., Java), and is relatively easy to implement. Unfortunately, however, it results in counterexamples that would not arise in the unbounded context, so the soundness of counterexamples is violated. This approach leads to considerable confusion amongst users [2], and imposes the burden of having to filter out the spurious cases.

Another common strategy is to introduce a notion of undefinedness — at the value, term or formula level — and extend the semantics of the operators accordingly. However this is done, its consequence will be that formulas expressing standard properties will not hold. The associativity of addition, for example, will be violated, because the definedness of the entire expression may depend on the order of summation. In logics that take this approach, the user is expected to insert explicit guards that ensure that desired properties do not rely on undefined values. In our setting, however, where the partiality arises not from any feature of the system being described, but from an artifact of the analysis, demanding that such guards be written would be unreasonable, and would violate Alloy's principle of separating description from analysis bounds.

This paper provides a different solution to the dilemma. Roughly speaking, counterexamples that would result in arithmetic overflow are excluded from the analysis, so that any counterexample that is presented to the user is guaranteed not to be spurious. This is achieved by redefining the semantics of quantifiers in the bounded setting so that the models of a formula are always models of the formula in the unbounded setting. This solution has been implemented in Alloy4.2 and can be activated via the "Forbid Overflows" option.

The rest of the paper is organized as follows. Section 2 illustrates some of the anomalies that arise from treating overflow as wraparound. Section 3 shows the problem in a more realistic context, by presenting an Alloy model of a minimum spanning tree algorithm that combines arithmetic and relational operators, and shows how a valid theorem can produce spurious counterexamples. Section 4 gives our new semantics, and Section 5 explains its implementation in boolean circuits. Finally, Section 7 presents related work on the topic of partial functions in logic, compares our approach with the existing ones, and discusses alternatives for solving the issue of overflows in Alloy.

2 Prototypical Overflow Anomalies

While a wraparound semantics for integer overflow is consistent and easily explained, its lack of correspondence to unbounded arithmetic produces a variety of anomalies. Most obviously, the expected properties of arithmetic do not necessar-

```
check {
  all a, b: Int |
    a > 0 && b > 0 => a.plus[b] > 0
} for 3 Int
```

counterexample

```
Int = {-4, -3, ..., 2, 3}
a = 3; b = 1;
a.plus[b] = - 4
```

(a) Sum of two positive integers is not necessarily positive.

```
check {
  all s: set univ |
    some s iff #s > 0
} for 4 but 3 Int
```

counterexamples

```
Int = {-4, -3, ..., 2, 3}
s = {S0, S1, S2, S3}
#s = -4
```

(b) Overflow anomaly involving cardinality of sets.

Fig. 1. Prototypical overflow anomalies in the previous version of Alloy.

ily hold: for example, that the sum of two positive integers is positive (Fig. 1.a). More surprisingly, expected properties of the cardinality operator may not hold. For example, the Alloy formula `some s` is defined to be true when the set `s` contains some elements. One would expect this to be equivalent to stating that the set has a non-zero cardinality (Fig. 1.b). And yet this property will not hold if the cardinality expression `#s` overflows, since it may wrap around, so that a set with enough elements is assigned a negative cardinality.

Of course, in practice, Alloy is more often used for analyzing software designs than for exploring mathematical theorems, and so properties of this kind are rarely stated explicitly. But such properties are often relied upon implicitly, and consequently, when they fail to hold, the spurious counterexamples that are produced are even harder to comprehend. Such a case arises in the the example discussed in the next section, where a test for an undirected graph being treelike is expressed by saying that there should be one fewer edge than nodes. Clearly, when using such a formulation, the user would rather not consider the effects of wraparound in counting nodes or edges.

3 Motivating Example

Consider checking Prim's algorithm [7, §23.2], a greedy algorithm that finds a minimum spanning tree (MST) for a connected graph with positive integral weights. Alloy is for the most part well-suited to this task, since it makes good use of Alloy's quantifiers and relational operators, including transitive closure. The need to sum integer weights, however, is potentially problematic, due to Alloy's bounded treatment of integers. [1]

[1] An alternative approach would be to use an analysis that includes arithmetic without imposing bounds. It is not clear, however, whether such an approach could be fully automated, since the logics that are sufficiently expressive to include both arithmetic and relational operators do not have decision procedures, and those (such as SMT) that do offer decision procedures for arithmetic are not expressive enough. In this paper, we are not arguing that such an approach cannot work. But, either way,

```
open util/ordering[Time]

sig Time {}

sig Node {covered: set Time}

sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 and #nodes = 2
}

pred cutting (e: Edge, t: Time) {
  (some e.nodes & covered.t) and (some e.nodes & (Node - covered.t))
}

pred step (t, t': Time) {
  -- stutter if done, else choose a minimal edge from a covered to an uncovered node
  covered.t = Node =>
         chosen.t' = chosen.t and covered.t' = covered.t
  else some e: Edge {
         cutting[e,t] and (no e2: Edge | cutting[e2,t] and e2.weight < e.weight)
         chosen.t' = chosen.t + e
         covered.t' = covered.t + e.nodes}
}

fact prim {
  -- initially just one node marked
  one covered.first and no chosen.first
  -- steps according to algorithm
  all t: Time - last | step[t, t.next]
  -- run is complete
  covered.last = Node
}

pred spanningTree (edges: set Edge) {
  -- empty if only 1 node and 0 edges, otherwise covers set of nodes
  (one Node and no Edge) => no edges else edges.nodes = Node
  -- connected and a tree
  #edges = (#Node).minus[1]
  let adj = {a, b: Node | some e: edges | a + b in e.nodes} |
    Node -> Node in *adj
}

correct: check { spanningTree [chosen.last] } for 5 but 10 Edge, 5 Int

smallest: check {
  no edges: set Edge {
    spanningTree[edges]
    (sum e: edges | e.weight) < (sum e: chosen.last | e.weight)}
} for 5 but 10 Edge, 5 Int
```

Fig. 2. Alloy model for bounded verification of Prim's algorithm that finds a minimum spanning tree for a weighted connected graph.

Figure 2 shows an Alloy representation of the problem. The sets (signatures in Alloy) `Node` and `Edge` (lines 5–5 and 7–13) represent the nodes and edges of a graph. Each edge has a weight (line 8) and connects a set of nodes (line 9); weights are non-negative and edges connect exactly two nodes (line 12).

This model uses the *event-based idiom* [10, §6.2.4] to model sequential execution. The `Time` signature (line 3) is introduced to model discrete time instants, and fields `covered` (line 5) and `chosen` (line 10) track which nodes and edges have been covered and selected respectively at each time. Initially (line 31) an arbitrary node is covered and no edges have been chosen. In each subsequent time step (line 33), the state changes according to the algorithm. The algorithm terminates (line 35) when the set of all nodes has been covered.

At each step, a 'cutting edge' (that is, one that connects a covered and a non-covered node) is selected such that there is no other cutting edge with a smaller weight (line 24). The edge is marked as chosen (line 25), and its nodes as covered (line 26)[2]. If the node set has already been covered (line 21), instead no change is made (line 22), and the algorithm stutters.[3]

Correctness entails two properties, namely that: (1) at the end, the set of covered edges forms a spanning tree (line 47), and (2) there is no other spanning tree with lower total weight (lines 49–53). The auxiliary predicate (`spanningTree`, lines 38–45) defines whether a given set of edges forms a spanning tree, and states that, unless the graph has no edges and only one node, the edges cover all nodes of the graph (line 40), the number of given edges is one less than the number of nodes (line 42), and that all nodes are connected by the given set of edges (lines 43–44).

If we run the previous version of the Alloy Analyzer to check these two properties, the `smallest` check fails. In each of the reported counterexamples, the expression `sum e: edges | e.weight` (representing the sum of weights in the alternative tree, line 52) overflows and wraps around, and thus appears (incorrectly) to have a lower total weight than the tree constructed.[4] In the latest version of the Alloy Analyzer that incorporates the approach described in this paper, the check, as expected, yields no counterexamples for a scope of up to 5 nodes, up to 10 edges and integers ranging from -16 to 15.

4 Approach

Our goal is to give a semantics to formulas whose arithmetic expressions might involve out-of-domain applications, such as the addition of two integers that ide-

exploring ways to mitigate the effects of bounding arithmetic has immediate benefit for users of Alloy, and may prove useful for other tools that impose ad hoc bounds.

[2] For a field `f` modeling a time-dependent state component, the expression `f.t` represents the value of `f` at time `t`.

[3] An implementation would, of course, terminate rather than stuttering. Ensuring that traces can be extended to a fixed length allows better symmetry breaking to be employed, dramatically improving performance.

[4] One might think that this overflow could be avoided by adding guards, for example that the total computed weight in the alternative tree is not negative. This does not work, since the sum can wrap around all the way back into positive territory.

ally would require a value that cannot be represented. In contrast to traditional approaches to the treatment of partial functions, the out-of-domain applications arise here not from any intrinsic property of the system being modeled, but rather from a limitation of the analysis.[5] Consequently, whereas it would be appropriate in more traditional settings to produce a counterexample when an out-of-bounds application occurs, in this setting, we aim to mask such counterexamples, since they do not indicate problems with the model per se.

First, a standard three-valued logic [13] is adopted, in which elementary formulas involving out-of-bounds arithmetic applications are given the third logical value of 'undefined' ($\perp$), and undefinedness is propagated through the logical connectives in the expected way (so that, for example, 'false and undefined' evaluates to false). But the semantics of quantifiers diverges from the standard treatment: the meaning of a quantified formula is adjusted so that the bound variable ranges only over values that would yield a body that evaluates to true or false. Thus bindings that would result in an undefined quantification are masked, and quantified formulas are never undefined. Since every top level formula in an Alloy model is quantified[6] this means that counterexamples (and, in the case of simulation, instances) never involve undefined terms.

This semantics cannot be implemented directly, since the analysis does not explicitly enumerate values of bound variables, but instead uses a translation to boolean satisfiability (SAT) [21]. A scheme is therefore needed in which the formula is translated compositionally to a SAT formula. To achieve this, a boolean formula is created to represent whether or not an arithmetic expression is undefined. This is then propagated to elementary subformulas in an unconventional way which ensures the high-level semantics of quantifiers given above.

To understand this intuitively, it may help to think of all the quantifiers being eliminated by explicit unrolling, and the entire formula being put in disjunctive normal form, as a collection of clauses, each consisting of a conjunction of elementary subformulas. The goal is to ensure that when an arithmetic term is undefined, the clause containing it evaluates to false and is effectively dropped.

We therefore have given two semantics: the high level semantics that the user needs to understand, and the low level semantics that justifies the analysis. This lower level semantics is then implemented by a translation to boolean circuits.

4.1 User-Level Semantics

As explained above, the key idea of our approach is to change the semantics of quantifiers so that the quantification domain is restricted to those values for which the body of the quantifier is defined (determined by the **`def`** function):

[5] Note that this discussions concern only the partial function applications arising from arithmetic operators; partial functions over uninterpreted types are treated differently in Alloy, and counterexamples involving their application are never masked.

[6] The fields and signatures of an Alloy model are always implicitly bound in an outermost existential quantifier, which is eliminated in analysis by skolemization.

$$[\![\texttt{all } x\texttt{: Int | } p(x)]\!] = \forall x \in \texttt{Int} \bullet \textbf{def}[\![p(x)]\!] \implies p(x)$$
$$[\![\texttt{some } x\texttt{: Int | } p(x)]\!] = \exists x \in \texttt{Int} \bullet \textbf{def}[\![p(x)]\!] \wedge p(x)$$

Integer expressions (i.e. those employing Alloy's arithmetic operators) are undefined if any argument is undefined or the evaluation results in overflow:

$$\textbf{def}[\![\alpha(i_1, \ldots, i_n)]\!] = (i_1 \neq \bot) \wedge \cdots \wedge (i_n \neq \bot) \wedge \neg([\![\alpha(i_1, \ldots, i_n)]\!] \text{ overflows})$$

Integer predicates are boolean formulas that relate one or more integer expressions. In Alloy, the only integer predicates are the integer comparison operators. They are also undefined if any argument is undefined:

$$\textbf{def}[\![\rho(i_1, \ldots, i_n)]\!] = (i_1 \neq \bot) \wedge \cdots \wedge (i_n \neq \bot)$$

A formula is defined if it evaluates to either true or false when three-valued logic truth tables of propositional operators are used (e.g. [13, Table A.1]):

$$\textbf{def}[\![\texttt{and}(p,q)]\!] = (p \wedge_3 q) \neq \bot \qquad \textbf{def}[\![\texttt{implies}(p,q)]\!] = (p \Rightarrow_3 q) \neq \bot$$
$$\textbf{def}[\![\texttt{or}(p,q)]\!] = (p \vee_3 q) \neq \bot \qquad \textbf{def}[\![\texttt{not}(p)]\!] = (\neg_3 p) \neq \bot$$

Finally, quantifiers are always defined:

$$\textbf{def}[\![\texttt{all } x \mid p(x)]\!] = \texttt{true} \qquad \textbf{def}[\![\texttt{some } x \mid p(x)]\!] = \texttt{true}$$

Note that the semantics of the rest of the Alloy logic (in particular, of the relational operators) remains unchanged.

4.2 Implementation-Level Semantics

A direct implementation of the user-level semantics in Alloy would entail a three-valued logic, and the translation to SAT would thus require 2 bits for a single boolean variable (to represent the 3 possible values), a substantial change to the existing Alloy engine. Furthermore, such a change would likely adversely affect the analysis performance of models that do not use integer arithmetic. In this section, we show how the same semantics can be achieved using the existing Alloy engine, merely by adjusting the translation of elementary integer functions and integer predicates.

To make all formulas denote (and thus to avoid the need for a third boolean value), a truth value must be assigned to an integer predicate even when some of its arguments are undefined. A common approach [8,17] is to assign the value `false`. For example, the sentence e_1`<` e_2 will be true *iff* both e_1 and e_2 are defined and e_1 is less than e_2 (and similarly for e_1`>=` e_2):

$$[\![\ \texttt{lt}(e_1,e_2)\ ]\!] = e_1 < e_2 \ \wedge\ e_1{\downarrow} \ \wedge\ e_2{\downarrow} \qquad [\![\ \texttt{gte}(e_1,e_2)\ ]\!] = e_1 \geq e_2 \ \wedge\ e_1{\downarrow} \ \wedge\ e_2{\downarrow}$$

(using the syntactic shortcuts $e{\downarrow} \equiv e \neq \bot$, and $e{\uparrow} \equiv e = \bot$).

Negation presents a challenge. Following the high-level semantics, negation of an integer predicate (e.g., `!(`e_1`<` e_2`)`) is still undefined if any argument is undefined. Therefore, under the low-level semantics, `!(`e_1`<` e_2`)` must also, despite the negation, evaluate to false if either e_1 or e_2 is undefined (and thus have

(a) Semantic Domains

```
Formula      = BoolConst | IntPred(IntExpr, ..., IntExpr) |
               BoolPred(Formula, ..., Formula) |
               QuantFormula(VarDecl, Formula)
IntExpr      = IntConst | IntVar | IntFunc(IntExpr, ..., IntExpr)
BoolConst    = true | false
IntConst     = ⊥ | 0 | -1 | 1 | -2 | 2 | ...
QuantFormula = all | some
BoolPred     = not₁ | and₂ | or₂ | implies₂ | iff₂
IntPred      = eq₂ | neq₂ | gt₂ | gte₂ | lt₂ | lte₂
IntFunc      = neg₁ | plus₂ | minus₂ | times₂ | div₂ | mod₂ |
               shl₂ | shr₂ | sha₂ | bitand₂ | bitor₂ | bitxor₂
Store        = {var: IntVar; val: IntConst; quant: QuantFormula;
                polarity: BoolConst; parent: Store}
```

(b) Symbols

$\bot \in$ `IntConst`	(undefined integer)	$b_i \in$ `BoolConst`	(boolean constants)
$i_i \in$ `IntConst`	(integer constants)	$p_i \in$ `Formula`	(boolean formulas)
$e_i \in$ `IntExpr`	(integer expressions)	$\beta_i \in$ `BoolPred`	(boolean predicates)
$\rho_i \in$ `IntPred`	(integer predicates)	$x_i \in$ `IntVar`	(integer variables)
$\alpha_i \in$ `IntFunc`	(arithmetic functions)	$q_i \in$ `QuantFormula`	(quantified formula)

(c) Stores

σ : `Store` (environment of nested quantifiers and variable bindings)

Fig. 3. Overview of semantic domains, symbols, and stores to be used. Subscripts in function and predicate names indicate their arities.

aeval : `IntExpr` → `Store` → `IntConst`

$$
\begin{aligned}
&\textbf{aeval}[\![i]\!]\sigma &&= i \\
&\textbf{aeval}[\![x]\!]\{x_\sigma, i_\sigma, q, b, \sigma_\mathsf{p}\} &&= \textbf{if } x_\sigma = x \textbf{ then } i_\sigma \textbf{ else } \textbf{aeval}[\![x]\!]\sigma_\mathsf{p} \\
&\textbf{aeval}[\![\alpha(i_1, \ldots, i_n)]\!]\sigma &&= \begin{cases} \bot & \textbf{if } i_i = \bot \text{ or ... or } i_n = \bot \\ \bot & \textbf{if } \alpha(i_1, \ldots, i_n) \text{ overflows} \\ \alpha(i_1, \ldots, i_n) & \text{otherwise} \end{cases} \\
&\textbf{aeval}[\![\alpha(e_1, \ldots, e_n)]\!]\sigma &&= \textbf{aeval}[\![\alpha(\textbf{aeval}[\![e_1]\!]\sigma, \ldots, \textbf{aeval}[\![e_n]\!]\sigma)]\!]\sigma
\end{aligned}
$$

Fig. 4. Evaluation of arithmetic operations (**aeval**). If any operand of an arithmetic operation is undefined, the result is undefined too.

beval : `Formula` → `Store` → `BoolConst`

$$
\begin{aligned}
&\textbf{beval}[\![b]\!]\sigma &&= b \\
&\textbf{beval}[\![\rho(e_1, e_2)]\!]\sigma &&= \textbf{ieval}[\![\rho(e_1, e_2)]\!]\sigma \\
&\textbf{beval}[\![\mathtt{not}(p)]\!]\{x, i, q, b, \sigma_\mathsf{p}\} &&= \neg\, \textbf{beval}[\![p]\!]\{x, i, q, \neg b, \sigma_\mathsf{p}\} \\
&\textbf{beval}[\![\beta(p_1, \ldots, p_2)]\!]\sigma &&= \beta(\textbf{beval}[\![p_1]\!]\sigma, \ldots, \textbf{beval}[\![p_2]\!]\sigma) \\
&\textbf{beval}[\![\mathtt{all}\ x\mathtt{:\ Int\ |}\ p]\!]\sigma &&= \bigwedge_{i \in \text{ Int}} \textbf{beval}[\![p]\!]\{x, i, \mathtt{all}, \mathtt{true}, \sigma\} \\
&\textbf{beval}[\![\mathtt{some}\ x\mathtt{:\ Int\ |}\ p]\!]\sigma &&= \bigvee_{i \in \text{ Int}} \textbf{beval}[\![p]\!]\{x, i, \mathtt{some}, \mathtt{true}, \sigma\}
\end{aligned}
$$

Fig. 5. Evaluation of boolean formulas. The new semantics (together with the **ieval** function, Fig. 6) ensures that quantifiers quantify over only those values that do not cause any overflows.

$$
\begin{array}{l}
\textbf{ieval} : \texttt{IntPred} \to \texttt{Store} \to \texttt{BoolConst} \\
\hline
\quad \textbf{ieval}[\![\rho(e_1,\, e_2)]\!]\sigma \quad = \textbf{let}\ b\ = \rho(\textbf{aeval}[\![e_1]\!]\sigma, \textbf{aeval}[\![e_2]\!]\sigma)\textbf{in} \\
\qquad\qquad\qquad\qquad\quad\ \ \textbf{ensureDef}[\![b, \{e_1, e_2\}]\!]\sigma \\
\\
\textbf{ensureDef} : \texttt{BoolConst} \to \{\texttt{IntExpr}\} \to \texttt{Store} \to \texttt{BoolConst} \\
\hline
\quad \textbf{ensureDef}[\![b,\, e_{\texttt{in}}]\!]\{x,\, i,\, q,\, b_{\texttt{pol}},\, \sigma_{\texttt{p}}\} = \textbf{let}\ e_{\text{univ}} = \{\, e \mid e \in e_{\texttt{in}} \wedge \textbf{isUnivQuant}[\![e]\!]\sigma\}\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ e_{\text{ext}}\ = e_{\texttt{in}} \setminus e_{\text{univ}}\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ b_{\texttt{def}}\ = (e_{\text{ext}} = \varnothing) \vee \bigwedge_{e \in e_{\text{ext}}} (\textbf{aeval}[\![e]\!]\sigma \neq \bot)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ b_{\texttt{undef}} = (e_{\text{univ}} \neq \varnothing) \wedge \bigvee_{e \in e_{\text{univ}}} (\textbf{aeval}[\![e]\!]\sigma = \bot)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ b_{\texttt{pol}}\ \ \textbf{then}\ (b \vee b_{\texttt{undef}}) \wedge b_{\texttt{def}} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{else}\ (b \vee \neg b_{\texttt{def}}) \wedge \neg b_{\texttt{undef}} \\
\textbf{isUnivQuant} : \texttt{IntExpr} \to \texttt{Store} \to \texttt{BoolConst} \\
\hline
\quad \textbf{isUnivQuant}[\![e]\!]\{\} \quad = \texttt{false} \\
\quad \textbf{isUnivQuant}[\![e]\!]\{x,\, i,\, q,\, b,\, \sigma_{\texttt{p}}\} \quad = \textbf{if}\ x \in \textbf{vars}[\![e]\!]\ \ \textbf{then}\ q\ = \texttt{all} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{else isUnivQuant}[\![e]\!]\sigma_{\texttt{p}} \\
\textbf{vars} : \texttt{IntExpr} \to \{\texttt{IntVar}\} \\
\hline
\quad \textbf{vars}[\![i]\!] \quad = \varnothing \\
\quad \textbf{vars}[\![x]\!] \quad = \{x\} \\
\quad \textbf{vars}[\![\alpha(e_1,\, \ldots,\, e_n)]\!] \quad = \textbf{vars}[\![e_1]\!] \cup \ldots \cup \textbf{vars}[\![e_n]\!]
\end{array}
$$

Fig. 6. Evaluation of integer predicates. If any argument of an integer predicate is undefined, the result is `true` if the expression is in a universally quantified context, otherwise it is `false`.

exactly the same semantics as $e_1 \geq e_2$). To achieve this behavior, the *polarity* [11] of each expression must be known. Polarity is easily determined by the structure of the enclosing negations. Evaluation of a binary integer predicate can be then formulated as (ignoring the stack of enclosing quantifiers for the moment):

$$
\begin{array}{l}
[\![\rho(e_1, e_2)]\!] = \textbf{if}\ \text{polarity is positive}\ \ \textbf{then}\ \rho([\![e_1]\!], [\![e_2]\!]) \wedge [\![e_1]\!]{\downarrow} \wedge [\![e_2]\!]{\downarrow} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{else}\ \rho([\![e_1]\!], [\![e_2]\!]) \vee \neg([\![e_1]\!]{\downarrow} \wedge [\![e_2]\!]{\downarrow})
\end{array}
$$

The polarity approach is not *compositional*, since the meaning of the negation of a formula is not simply the logical negation of the meaning of that formula. For that reason, this approach violates the law of the excluded middle, which, fortunately, will not be problematic, since the violation would only be observable for variable bindings that result in overflow and such bindings are excluded by the semantics (see Sec. 4.4).

The semantics are formally defined in Figs. 3–6. Expressions and formulas are interpreted in the context of a store that holds, for each variable bound in an enclosing quantifier: (a) the value of the variable in the particular binding, (b) whether the quantifier is universal or existential, and (c) its current polarity.

Evaluation of integer expressions (**aeval**) and boolean formulas (**beval**) has the same effect as evaluation in the user-level semantics; it is elaborated differently here simply to account for the need to pass the store. Every time a negation is seen, the inner formula is interpreted in a store in which the polarity is negated. Quantifiers are unfolded, with the body interpreted in a new nested

store with polarity set to `true`. For the evaluation of top-level formulas, an empty existential environment is presented.

Evaluation of integer predicates (**ieval**) is where the crucial differences lie. Whereas in the user-level semantics predicates evaluate to true, false and undefined, in this implementation semantics predicates evaluate only to true or false. When a predicate would have been undefined in the user-level semantics, its meaning will be either true or false, chosen in such a way as to ensure that the associated binding becomes irrelevant. This choice is represented by the auxiliary function **ensureDef**, which determines the truth value based on the current polarity and the stack of enclosing quantifiers.

For the existential case, the goal is to ensure that a predicate evaluates to false when any argument is undefined. However, when such a predicate contains a universally quantified variable, to achieve the desired semantics of the universal quantifier (which is to ignore cases where the body is undefined), it is enough to simply make the predicate evaluate to true instead. Therefore, all expressions with universally quantified variables are identified first (e_{univ}) and a definedness condition for them ($b_{\texttt{undef}}$) is computed as a disjunction of either being undefined. For all other arguments (e_{ext}) the definedness condition ($b_{\texttt{def}}$) is a conjunction of all being defined (as before). Finally, based on the value of the polarity flag ($b_{\texttt{pol}}$), the two conditions are attached to the base result (b).

4.3 Correspondence Between the Two Semantics

To show that our low-level semantics correctly implements the high-level user semantics, it is enough to establish a correspondence between the two definitions of quantifiers (the low-level semantics only introduced a change to the semantics of quantifiers). Following directly from the two definitions, this is equivalent to proving that whenever an expression $p(x)$ is undefined by the laws of three-valued logic (i.e., $\textbf{def}[\![p(x)]\!]$ is false), if x is *universally* quantified then $\textbf{beval}[\![p(x)]\!]$ evaluates to true, else it evaluates to false.

This hypothesis would traditionally be proved by a structural induction on expressions. Instead of giving a complete proof (which would exceed the scope of this paper), we explain several interesting base cases instead.

As said earlier, the low-level evaluation of integer predicates is where the crucial differences lie. Let us therefore consider the case when $p(x)$ is an integer predicate, $\rho(e_1(x), e_2(x))$. Furthermore, let us assume that $e_1(x)$ is undefined, which makes $p(x)$ undefined as well. In this context, polarity is positive, and the value of $\textbf{beval}[\![\rho(e_1(x), e_2(x))]\!]$ becomes the value of **ensureDef**. There are two cases to consider: (1) if x is *universally* quantified, e_{univ} contains both e_1 and e_2, $b_{\texttt{undef}}$ becomes true, $b_{\texttt{def}}$ is true by default, so the result is also true regardless of the base value b; (2) if x is *existentially* quantified, e_{ext} contains both e_1 and e_2, $b_{\texttt{def}}$ becomes false, $b_{\texttt{undef}}$ is false by default, so the result is also false, as expected.

Let us now assume that $p(x)$ is a negation of an integer predicate, $p(x) = \neg\rho(e_1(x), e_2(x))$, and that $e_1(x)$ is again undefined. Despite the negation, $p(x)$ is *still* undefined, so the low-level evaluation should behave exactly as in the previous case. The result of $\textbf{beval}[\![p(x)]\!]$ now becomes a negation of the value

returned by **ensureDef**, which, in contrast, now evaluates in a context where the polarity is negative. Following exactly the same derivation as before, it can be shown that **ensureDef** now returns false for the universal case, and true for the existential case (because of the negative polarity), so the end result of $\mathbf{beval}[\![p(x)]\!]$ remains the same, as expected.

4.4 The law of the excluded middle

We mentioned earlier that our non-compositional rule for negation breaks the law of the excluded middle. Usually, this is not a problem.

Consider checking the theorem that all integers when multiplied by two are either less than zero or not less than zero:

```
check { all x: Int | x.mul[2] < 0 or !(x.mul[2] < 0) } for 3 Int
```

If we run the Alloy Analyzer with overflow prevention turned on, this sentence is interpreted as "*for all integers x s.t. x times two does not overflow, x times two is either less than zero or not less than zero*", and thus no counterexample is found, which is consistent with classical logic.

In a sense, however, the violation of the law is visible if truth is associated with whether or not a check yields a counterexample at all. For example, a check of whether 4 plus 5 is *equal* to 6 plus 3 for the bitwidth of 4 (`Int = {-8, ..., 7}`) does not return a counterexample, but neither does a check of whether 4 plus 5 is *different* from 6 plus 3.

```
check { 4.plus[5]  = 6.plus[3] } for 4 Int -- no counterexample found
check { 4.plus[5] != 6.plus[3] } for 4 Int -- no counterexample found
```

Though this might at first appear confusing, it is consistent with our design goal: indeed, for a bitwidth of 4, there is no non-overflowing instance in which 4 plus 5 is either equal to or different from 6 plus 3.

5 Implementation in Circuits

The core task of finding satisfying models for a relational formula is delegated to Kodkod [20]. Kodkod is a bounded constraint solver for relational first-order logic. It works by translating a given relational formula (together with bounds) into an equivalent propositional formula and using an of-the-shelf SAT solver to check its satisfiability.

Detecting arithmetic overflows at the level of relational logic would be difficult, and probably inefficient. We therefore implemented our approach at the level of the translation to propositional logic, as an extension to Kodkod. Even though the goal here is to translate the input formula into a digital circuit (instead of evaluating it to a boolean constant), we only had to modify Kodkod's translation of appropriate terms directly following the denotational semantics presented in this paper. In summary, we changed:

- *the translation of arithmetic operations* to generate an additional one-bit overflow circuit which is set iff the operation overflows. We used textbook overflow circuits for all arithmetic operations supported by Kodkod;

– *the way the environment gets updated* so that it additionally keeps track of the polarity and the quantification stack;
– *the translation of integer comparison predicates* so that the original circuit representing the comparison result is extended to include the definedness conditions, exactly as defined above.

6 Evaluation

Finding suitable models for evaluating the new approach is difficult, because most Alloy models do not involve arithmetic, in part because of the problem of overflow that motivated this work.

To evaluate the approach of this paper, we took a previously published model of a flash filesystem [15] which uses arithmetic operations and whose analysis is non-trivial, and compared its execution under the old (Alloy4) and new (Alloy4.2) analysis schemes. This model involves both assertions (that certain properties hold) and simulations (that produce sample scenarios). First, we checked that there are no new spurious counterexamples, and that none of the expected valid scenarios are lost. This was not the focus of our evaluation, however, since the design of the analysis ensures it. Rather, our concern was that the addition of new clauses to the SAT formula generated by the Analyzer might increase translation and solving time.

The new translation always results in a larger SAT formula, because extra clauses are needed to rule out models that overflow. One might imagine that adding clauses would cause the solving time to increase. On the other hand, the additional clauses might result in a smaller search space, and thus potentially reduce the search time.

We ran all checks that were present in the "concrete" module of the model. The first 10 (`run1` through `run10`) are simulations (which all find an instance), and the remaining 6 (`check1` through `check6`) are checks, which, with the exception of `check5`, produce no counterexamples. For each check, we measured both the translation and solving time, as shown in Table 1. As expected, in some cases the analysis runs faster, and sometimes it takes longer. In total, with the overflow prevention turned on, the entire analysis finished in about 8 hours, as opposed to almost 12 hours that the same analysis took otherwise.

7 Related Work

The problem addressed in this paper is an instance of the more general problem of handling partial functions in logic. The most important difference, however, is that, in our case, the out-of-bound function applications arise due to deficiencies in the analysis, rather than from the inherent semantics of the logic. Requiring the user to introduce guards in the formal description itself to mitigate the effects of undefinedness is therefore not acceptable.

Despite this fundamental difference, our approach shares some features of several previously explored approaches.

	run1		run2		run3		run4		run5		run6		run7		run8		run9	
old	1.2	0.9	2.1	0.4	0.8	0.2	12.9	2.3	5.9	0.5	12.7	1.0	11.9	1.1	9.0	1.0	12.5	1.0
new	1.2	0.8	1.6	0.4	0.8	0.3	13.4	8.7	6.2	0.5	12.6	0.8	12.1	1.5	9.1	1.0	12.7	2.6
abs diff	0	0.1	0.5	0	0	-0.1	-0.5	-6.4	-0.3	0	0.1	0.2	-0.2	-0.4	-0.1	0	-0.2	-1.6
speedup	0	11.1	23.8	0	0	-50.0	-3.9	-278.3	-5.1	0	0.8	20.0	-1.7	-36.4	-1.1	0	-1.6	-160.0

	run10		check1		check2		check3		check4		check5		check6		**total**
old	25.7	14.8	20.0	39.6	12.1	2190.7	12.0	30673.3	12.5	3713.2	12.3	3.0	74.3	5782.6	42663.5
new	25.9	12.5	20.2	12.6	12.2	1670.4	12.2	16741.9	12.7	3526.9	12.5	1.3	73.9	7083.5	29304.5
abs diff	-0.2	2.3	-0.2	27	-0.1	520.3	-0.2	13931.4	-0.2	186.3	-0.2	1.7	0.4	-1300.9	13359.0
speedup	-0.8	15.5	-1.0	68.2	-0.8	23.8	-1.7	45.4	-1.6	5.0	-1.6	56.7	0.5	-22.5	31.3

Table 1. Analysis times of all checks found in the "concrete" module of a flash filesystem from [15]. All values are in seconds, except the values in the "speedup" row which are in percents. "old" stands for the previous version of Alloy, whereas "new" stands for the new version with overflow prevention turned on.

The *Logic of Partial Functions* (LPF) was proposed for reasoning about the development of programs [13, 14], and was adopted in VDM [12]. In this approach, not only integer predicates but also boolean formulas may be non-denoting, so truth tables extended to a three-valued logic are needed. This allows guards for definedness to be treated intuitively; thus, for example, even when "`x`" is equal to zero, formula `x!=0 => x/x=1`, evaluates to `true` in spite of `x/x=1` being undefined. Our approach uses this three-valued logic for determining whether the body of a quantified formula is undefined, but the meaning of the formula as a whole is treated differently – masking the binding that produces undefinedness rather than interpreting the quantification in the same three-valued logic.

Our implementation-level semantics adopts the *traditional approach to partial functions* (a term coined by Farmer [8]), in which all formulas must be denoting but functions may be partial. Farmer's approach, however, leaves open whether, given an undefined `a`, `!(a=a)` and `a!=a` have different meanings — an issue that in the standard setting is hard to resolve because of the competing concerns of compositionality and preserving complementarity of predicates. In our case, the non-compositional choice fits nicely with the user-level semantics.

Like the Alloy Analyzer, SMT [6] solvers can also be used for model finding. They all support unbounded integer arithmetic, so the problem of overflows does not arise. However, using Alloy over SMT-based tools has certain benefits, most notably the expressiveness of the Alloy relational language. There are higher-level languages that build on SMT technologies (e.g. Dafny [16]), but for a task similar to verifying Prim's algorithm, such tools are typically not fully automatic, and demand that the user provide intermediate lemmas.

Model-based languages such as B [3] and Z [18], being designed for specifying programs, make extensive use of partial functions. Both are based on set theory, and model functions as relations. Whereas in Alloy out-of-bounds applications of partial functions over uninterpreted types result in the empty set, in B such an application results in an unknown (but determined) value [19]. The initial specification of the Z notation [18] left the handling of partial functions open.

Several different approaches have been proposed (see [5] for a survey); in the end, it appears that the same approach as in B has evolved to be the norm [19]. In both Z and B, integers are unbounded, and so the problems of integer overflow

do not arise. On the other side, the tools for discharging proof obligations (e.g. Rodin [4]) are typically less automated than the Alloy Analyzer.

References

1. Alloy: A language and tool for relational models. `http://alloy.mit.edu/alloy`.
2. User posts about arithmetic overflows on Alloy community forum. `http://alloy.mit.edu/community/search/node/overflow`.
3. J. Abrial and A. Hoare. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.
4. J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6), 2010.
5. R. Arthan and L. Road. Undefinedness in Z: Issues for Specification and Proof. In *CADE-13 Workshop on Mechanization of Partial Functions*. Springer, 1996.
6. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010.
7. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
8. W. M. Farmer. Reasoning about partial functions with the aid of a computer. *Erkenntnis*, 43, 1995.
9. D. Gries and F. Schneider. *A logical approach to discrete math*. Texts and monographs in computer science. Springer-Verlag, 1993.
10. D. Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.
11. Jean-Yves and Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.
12. C. B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
13. C. B. Jones. Reasoning about partial functions in the formal development of programs. *Electron. Notes Theor. Comput. Sci.*, 145, January 2006.
14. C. B. Jones and M. J. Lovert. Semantic Models for a Logic of Partial Functions. *Int. J. Software and Informatics*, 5(1-2), 2011.
15. E. Kang and D. Jackson. Formal Modeling and Analysis of a Flash Filesystem in Alloy. In *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, ABZ '08, Berlin, Heidelberg, 2008. Springer-Verlag.
16. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*. Springer, 2010.
17. D. L. Parnas. Predicate logic for software engineering. *IEEE Trans. Softw. Eng.*, 19, September 1993.
18. J. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge tracts in theoretical computer science. Cambridge University Press, 1988.
19. B. Stoddart, S. Dunne, and A. Galloway. Undefined Expressions and Logic in Z and B. *Formal Methods in System Design*, 15, 1999.
20. E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT, 2008.
21. E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.