Preventing Arithmetic Overflows in Alloy

Aleksandar Milicevic[a,*], Daniel Jackson[a]

[a] *Massachusetts Institute of Technology*
Computer Science and Artificial Intelligence Laboratory,
32 Vassar St, Cambridge, MA 02139, USA

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.

Keywords: arithmetic overflows, partial functions, logic, first order, alloy

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 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 counterex-

*Corresponding author

Email addresses: aleks@csail.mit.edu (Aleksandar Milicevic), dnj@csail.mit.edu (Daniel Jackson)

URL: http://people.csail.mit.edu/aleks (Aleksandar Milicevic), http://people.csail.mit.edu/dnj (Daniel Jackson)

ample exists (in a larger scope). But it is generally sound with respect to counterexamples. That is, no spurious counterexamples are generated, so 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., [2]) 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 wraparound 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 among users, 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 it is by default turned on; it can be deactivated via the "Prevent Overflows" option.

The rest of the paper is organized as follows. Section 2 introduces the Alloy Analyzer. Section 3 illustrates some of the anomalies that arise from treating overflow as wraparound. Section 4 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 5 explains and formalizes our new semantics, which is the key contribution of this paper. Section 6 explains our implementation in boolean circuits and discusses how it ensures the desired semantics. Section 7 presents evaluation, showing (1) a case study where the analysis time is cut by 33% due to the reduced search space imposed by the new semantics, and (2) an exhaustive set of tests we applied to ensure our implementation meets the specification. Section 8 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. Section 9 concludes.

2. Alloy Background

Alloy [3] is a first-order relational modeling language. Alloy models lend themselves to fully automated bounded analysis—embodied in a tool called the Alloy Analyzer [1]. The expressiveness of the relational language is one of the characteristic features of Alloy which makes it particularly suitable for checking deep properties of structurally complex systems. To keep the logic decidable and the analysis fully automated, the Alloy Analyzer requires, however, that all domains be bounded.

The model finding part of the analysis is offloaded to Kodkod [4], a constraint solver for relational first-order logic. Kodkod works by translating a given relational formula (together with provided bounds for each relation) into an equivalent propositional boolean formula and using an of-the-shelf SAT solver to check its satisfiability.

In addition to pure relations, Kodkod also provides support for integers and arithmetic operations. Integers are an important part of Alloy, because they enable various program analysis tools that build on top of it; examples include tools for testing [5], verification [6], and specification execution [7].

Integers in Alloy must also be bounded. The user is required to explicitly specify the number of bits (*bitwidth*) to be used for their representation. In Alloy, signed integers are represented in a twos-complement system, restricting the analysis to using the integers from $\{-2^{bitwidth-1}, \ldots, 2^{bitwidth-1} - 1\}$. As explained above, this poses certain dilemmas about the semantics of arithmetic operations. The previous versions of Alloy (up to and including v4.1.2) implement wraparound semantics; in this paper we explain the new semantics implemented in Alloy 4.2, which excludes the models containing arithmetic overflows from the search space.

3. 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 necessarily hold: for example, that the sum of two positive integers is positive (Figure 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 cardinality greater than zero (Figure 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.

One might imagine that this problem could be eliminated by requiring that the scope of any analysis always assign a bitwidth to integers that can measure, without overflow, the cardinality of any signature. But this is not practical, since the cardinality operator by definition counts tuples, and can be applied to any relational expression — including one of higher arity (whose cardinality rises exponentially with the number of columns). An example of the use of the cardinality operator for non-set relations is the claim that a binary relation `p` has no more tuples than a binary relation `q` if `p` is a subrelation of `q` (Figure 1(c)).

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

```
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
```

counterexample

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

(b) Overflow anomaly involving cardinality of sets.

```
check {
  all p, q: univ -> univ |
    p in q => #p <= #q
} for 3 but 3 Int
```

counterexample

```
Int = {-4, -3, ..., 2, 3}
p = {}
q = {S0->S0, S0->S1, S1->S0, S1->S1}
#p = 0; #q = -4
```

(c) Overflow anomaly involving cardinality of relations.

Figure 1: Prototypical overflow anomalies in the previous version of Alloy.

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.

4. Motivating Example

Consider checking Prim's algorithm [8, §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.

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 proce-

dures for arithmetic are not expressive enough. In this paper, we are not arguing that such an approach cannot work, and indeed, experts in these other approaches may find a suitable encoding of the problem that makes it tractable. But, either way, 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.

Figure 2 shows an Alloy representation of the problem. The sets (signatures in Alloy) `Node` and `Edge` (lines 3–10) represent the nodes and edges of a graph. Each edge has a weight (line 5) and connects a set of nodes (line 6); weights are non-negative and edges connect exactly two nodes (line 9).

This model uses the *event-based idiom* [3, §6.2.4] to model sequential execution. The `Time` signature (line 2) is introduced to model discrete time instants, and fields `covered` (line 3) and `chosen` (line 7) track which nodes and edges have been covered and selected respectively at each time. Initially (line 25) an arbitrary node is covered and no edges have been chosen. In each subsequent time step (line 27), the state changes according to the algorithm. The algorithm terminates (line 29) 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 19). The edge is marked as chosen (line 20), and its nodes as covered (line 21)[1]. If the node set has already been covered (line 16), instead no change is made (line 17), and the algorithm stutters. An implementation would, of course, terminate rather than stuttering. In Alloy, however, ensuring that traces can be extended to a fixed length allows the Alloy Analyzer to employ a better symmetry breaking strategy, dramatically improving performance.

Correctness entails two properties, namely that: (1) at the end, the set of covered edges forms a spanning tree (line 39), and (2) there is no other spanning tree with lower total weight (lines 40–44). The auxiliary predicate (`spanningTree`, lines 31–38) 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 33), the number of given edges is one less than the number of nodes (line 35), and that all nodes are connected by the given set of edges (lines 36–37).

If we run the previous version of the Alloy Analyzer (v4.1.2) to check these two properties, the `smallest` check fails. In each of the reported coun-

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

```
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
```

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

terexamples, the expression `sum e: edges | e.weight` (representing the sum of weights in the alternative tree, line 43) overflows and wraps around, and thus appears (incorrectly) to have a lower total weight than the tree constructed. 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. In the latest version of the Alloy Analyzer that incorporates the approach described in this paper (v4.2), 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.

5. 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 ideally 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.[2] 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 [9] 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 is *not undefined* (i.e., evaluates to true or false)[3]. Thus bindings that would result in an undefined quantification are masked (never presented to

[2]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.

[3]One might wonder at this point how an automated solver for this logic can possibly know in advance which bindings will not yield an overflow (without explicitly enumerating and checking every single combination); indeed, our compilation to SAT does not modify the ranges of the bound variables, rather, it uses a clever translation (as explained in Section 5.2) that make the associated bindings irrelevant whenever an overflow occurs.

the user), and quantified formulas are never undefined. Since every top level formula in an Alloy model is quantified (the fields and signatures of an Alloy model are always implicitly bound in an outermost existential quantifier) this means that counterexamples (and, in the case of simulation, instances) never involve undefined terms.

This semantics cannot be implemented directly, since the Alloy Analyzer does not explicitly enumerate values of bound variables, but instead uses a translation to boolean satisfiability (SAT) [10]. 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 that ensures the high-level semantics of quantifiers given above.

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

5.1. 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 **dfn** predicate). For the universal quantifier, that means that the body must be satisfied for all bindings for which the body does not overflow; similarly for the existential quantifier, there must exist at least one binding for which the body does not overflow and evaluates to true:

$$\begin{aligned}
[\![\texttt{all } r\texttt{: } \mathcal{R} \texttt{ | } \phi(r)]\!] &\equiv \forall r \in \mathcal{R} \setminus \{i \mid r \to i \text{ causes overflow in } \phi(r)\} \bullet [\![\phi(r)]\!] \\
[\![\texttt{some } r\texttt{: } \mathcal{R} \texttt{ | } \phi(r)]\!] &\equiv \exists r \in \mathcal{R} \setminus \{i \mid r \to i \text{ causes overflow in } \phi(r)\} \bullet [\![\phi(r)]\!]
\end{aligned}$$

An important subtlety to note about this definition is that it is more strict than simply saying that, for a given binding, some subexpression in the body of the quantifier is undefined (e.g., because of an arithmetic overflow); additionally, it is crucial to ensure that the undefinedness is *caused* by this particular binding and not something else (e.g., an addition of two integer constants that overflows). Formally defining this causation relation at this level would only clutter this semantics and defeat its main purpose—namely to be intuitive and easy to understand. Instead, we formalize here how formulas are evaluated (denoted with [[]] brackets) and what it means for a formula/expression to be undefined (embodied in the **dfn** function); the implementation-level semantics (Section 5.2), of course, provides a complete

formalization. A concrete example of applying the user-level semantics to evaluate quantifiers can be found in Section 5.3.

We have already given the quantifier evaluation semantics; all other formulas are either undefined or evaluate to the same value they do in the standard Alloy Analyzer (denoted as $\mathcal{A}[\phi]$, which is always defined):

$$[\![\phi]\!] \equiv \begin{cases} \bot, & \text{if } \neg\mathbf{dfn}[\phi] \\ \mathcal{A}[\phi], & \text{otherwise} \end{cases}$$

Quantifiers are always defined. This simply follows from the idea to restrict quantification domains to bindings for which the quantifier body is defined—if every instantiation of the body is defined, the quantifier as a whole must also be defined:

$$\mathbf{dfn}[\texttt{all}\ r : \mathcal{R} \mid \phi(r)] \equiv \texttt{true} \qquad \mathbf{dfn}[\texttt{some}\ r : \mathcal{R} \mid \phi(r)] \equiv \texttt{true}$$

Integer expressions (i.e., those using Alloy's arithmetic operators) are defined all arguments are defined and the evaluation does not result in overflow:

$$\mathbf{dfn}[\alpha(i_1, \ldots, i_n)] \equiv \mathbf{dfn}[i_1] \wedge \cdots \wedge \mathbf{dfn}[i_n] \wedge \neg(\alpha[i_1, \ldots, i_n] \text{ overflows})$$

Other expressions supported in Alloy include: (1) relational algebra operators (e.g., union, intersection, etc.), (2) operators that take a relation and produce an integer (e.g., the cardinality operator), and (3) operators that take an integer and produce a relation (e.g., the int-to-expression cast operator). They are all defined if all arguments are defined:

$$\mathbf{dfn}[\psi(r_1, \ldots, r_n)] \equiv \mathbf{dfn}[r_1] \wedge \cdots \wedge \mathbf{dfn}[r_n]$$

Predicates are boolean formulas that relate one or more (either integer or relational) expressions. In Alloy, predicates that relate integer expressions correspond directly to integer comparison operators (e.g., less than, greater than, equal to, etc.), and predicates that relate relational expressions correspond to standard boolean operators in relational algebra (e.g., subset, equality, etc.). Predicates are also defined if all arguments are defined:

$$\mathbf{dfn}[\phi(r_1, \ldots, r_n)] \equiv \mathbf{dfn}[r_1] \wedge \cdots \wedge \mathbf{dfn}[r_n]$$

A constant is defined unless it is equal to $\bot$:

$$\mathbf{dfn}[c] \equiv c \neq \bot$$

A formula is defined if it evaluates to either true or false when three-valued logic truth tables (e.g., [9, Table A.1]) of propositional operators are used (denoted here as $\wedge_3$, $\vee_3$, $\neg_3$, $\Rightarrow_3$, and $\Leftrightarrow_3$). Before the three-valued propositional operators can be applied, the operands must first be evaluated to determine their definedness:

$$\begin{array}{lll}
\textbf{dfn}\,[\texttt{and}(p,q)] & \equiv & (\llbracket p \rrbracket \wedge_3 \llbracket q \rrbracket) \neq \bot \\
\textbf{dfn}\,[\texttt{or}(p,q)] & \equiv & (\llbracket p \rrbracket \vee_3 \llbracket q \rrbracket) \neq \bot \\
\textbf{dfn}\,[\texttt{implies}(p,q)] & \equiv & (\llbracket p \rrbracket \Rightarrow_3 \llbracket q \rrbracket) \neq \bot \\
\textbf{dfn}\,[\texttt{iff}(p,q)] & \equiv & (\llbracket p \rrbracket \Leftrightarrow_3 \llbracket q \rrbracket) \neq \bot \\
\textbf{dfn}\,[\texttt{not}(p)] & \equiv & (\neg_3 \llbracket p \rrbracket) \neq \bot
\end{array}$$

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

5.2. 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 evaluation of elementary integer functions and integer predicates.

We call this semantics "implementation-level", not because it shows how boolean formulas and relational expressions are translated (rewritten) to propositional formulas (to be solved by a SAT solver), but because it is directly implementable on top of Kodkod, the solver used by the Alloy Analyzer. In this section, thus, we show the mathematical semantics of evaluating formulas to boolean constants in the presence of arithmetic overflows; in Section 6 we explain how we modified Kodkod to achieve this semantics.

> **Syntax notes**. For semantic function definitions, we use the expression **fun_name**$[args...]\,\sigma$ whenever the content of the store is irrelevant; otherwise, we either write **fun_name**$[args...](x,\ i,\ q,\ b,\ \sigma_{\texttt{p}})$ (which automatically assigns concrete variable names to store fields), or use the dot notation to access store fields by name (e.g., $\sigma.\texttt{polarity}$). We use the same square brackets to explicitly designate cases where a built-in function or predicate (e.g., α, ρ, β) is to be applied to a number of constant arguments to produce a concrete (constant) result.

To make all formulas denote (and thus 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. The key idea behind our approach is that in such cases a logic value can be assigned to make the subformula irrelevant in the context of the entire (enclosing) formula (i.e., the Alloy model as a whole). This is different from common approaches (e.g., [11, 12]) which in those cases simply 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):

$$\begin{aligned}[\![\ \ \texttt{lt}(e_1,e_2)\ ]\!] &\equiv [\![e_1]\!] < [\![e_2]\!] \ \wedge\ \textbf{dfn}[e_1] \ \wedge\ \textbf{dfn}[e_2] \\ [\![\ \texttt{gte}(e_1,e_2)\ ]\!] &\equiv [\![e_1]\!] \geq [\![e_2]\!] \ \wedge\ \textbf{dfn}[e_1] \ \wedge\ \textbf{dfn}[e_2]\end{aligned}$$

Negation presents a challenge. Following the user-level semantics, negation of an integer predicate (e.g., `!(`e_1`<`e_2`)`) is still undefined if any argument is undefined. Therefore, under the implementation-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 exactly the same semantics as $e_1 \geq e_2$). To achieve this behavior, the *polarity* [13] of each expression must be known (which is, loosely speaking, the number of enclosing negations). Evaluation of a binary integer predicate can be then formulated (ignoring the stack of enclosing quantifiers for the moment) as:

$$[\![\rho(e_1,e_2)]\!] \equiv \begin{cases} \rho[[\![e_1]\!],[\![e_2]\!]] \ \wedge\ (\textbf{dfn}[e_1] \ \wedge\ \textbf{dfn}[e_2]), & \text{if polarity is positive;} \\ \rho[[\![e_1]\!],[\![e_2]\!]] \ \vee\ \neg(\textbf{dfn}[e_1] \ \wedge\ \textbf{dfn}[e_2]) & \text{otherwise.} \end{cases}$$

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 Section 5.4).

In the presence of quantifiers, to achieve the goal of restricting quantification domains to values that do not cause overflows, the key idea is to assign truth values to formulas that overflow such that the *associated bindings* to quantification variables *become irrelevant*. For example, consider the following formula:

```
some x: Int | x > 0 and x+1 < 0
```

Kodkod unrolls this existential quantifier to a disjunction with as many clauses as there are integers in the given scope. Assuming that the bitwidth

is set to 4 (integers ranging from -8 to 7), the clause in which `x` is bound to 7 will overflow. To make a clause in a disjunction of clauses irrelevant, the truth value `false` must be assigned to it; in this context, therefore, we define `x+1 < 0` to be false when `x` is bound to 7.

Now consider an example involving a universal quantifier:

```
all x: Int | x+1 > x
```

Universal quantifiers get unrolled to a conjunction of clauses; making a binding irrelevant in this case means assigning the value `true` to the associated clause. Assuming the same scope for integers, when `x` is bound to 7, we define `x+1 > x` in this context to be true.

The semantics is formally defined in Figures 3–6. Expressions and formulas are interpreted in the context of a store (defined in Figure 3(a)) which for each variable (`var`) bound in an enclosing quantifier holds: (1) the value of the variable in the particular binding (`val`), (2) whether the quantifier is universal or existential (`quant`), and (3) its current polarity (`polarity`). Here we only focus on handling integers, as the semantics of the relational operators remains the same.

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 (depending on the current polarity, the quantifier is adjusted according to De Morgan's laws). For the evaluation of top-level formulas, an empty existential environment is presented.

The crucial differences lie in the evaluation of integer predicates (**`ieval`**). 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 **`ensureDfn`**, which determines the truth value based on the current polarity and the stack of enclosing quantifiers.

As explained before, to make bindings resulting in overflow irrelevant, it is enough to make predicates containing existentially quantified variables evaluate to `false` and predicates containing universally quantified variables evaluate to `true`. Therefore, all expressions with universally quantified variables are identified first ($e_{\texttt{univ}}$) and a definedness condition for them ($b_{\texttt{undef}}$)

(a) Syntactic 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

$\perp$ ∈ `IntConst`	(undefined value)	b_i ∈ `BoolConst`	(boolean constants)
i_i ∈ `IntConst`	(integer constants)	p_i ∈ `Formula`	(formulas)
e_i ∈ `IntExpr`	(integer expressions)	β_i ∈ `BoolPred`	(boolean predicates)
ρ_i ∈ `IntPred`	(integer predicates)	x_i ∈ `IntVar`	(integer variables)
α_i ∈ `IntFunc`	(arithmetic functions)	q_i ∈ `QuantFormula`	(quantified formula)

(c) Stores

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

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

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

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

Figure 4: Evaluation of arithmetic operations (**aeval**). If any operand of an arithmetic operation is undefined, the result is undefined too. The $\alpha[i_1, \ldots, i_n]$ syntax applies the α integer predicate to given integer constants; the result is another integer constant.

beval : Formula → Store → BoolConst	
$\textbf{beval}[b]\,\sigma$	$\equiv\ b$
$\textbf{beval}[\rho(e_1, e_2)]\,\sigma$	$\equiv\ \textbf{ieval}[\rho(e_1, e_2)]\,\sigma$
$\textbf{beval}[\texttt{not}(p)](x,\ i,\ q,\ b,\ \sigma_\texttt{p})$	$\equiv\ \neg\ \textbf{beval}[p](x,\ i,\ q,\ \neg b,\ \sigma_\texttt{p})$
$\textbf{beval}[\beta(p_1,\ \ldots,\ p_2)]\,\sigma$	$\equiv\ \beta\,[\textbf{beval}\,[p_1]\,\sigma,\ \ldots,\ \textbf{beval}\,[p_2]\,\sigma]$
$\textbf{beval}[\texttt{all}\ x\texttt{: Int | }\ p]\,\sigma$	$\equiv\ \texttt{let}\ q\ \texttt{=}\ (\sigma\texttt{.polarity})\ \texttt{?}\ \texttt{all}\ \texttt{:}\ \texttt{some}\ \textbf{in}$
	$\equiv\ \bigwedge_{i\in\text{Int}}\ \textbf{beval}\,[p]\,(x, i, q, \sigma\texttt{.polarity}, \sigma)$
$\textbf{beval}[\texttt{some}\ x\texttt{: Int | }\ p]\,\sigma$	$\equiv\ \texttt{let}\ q\ \texttt{=}\ (\sigma\texttt{.polarity})\ \texttt{?}\ \texttt{some}\ \texttt{:}\ \texttt{all}\ \textbf{in}$
	$\equiv\ \bigvee_{i\in\text{Int}}\ \textbf{beval}\,[p]\,(x, i, q, \sigma\texttt{.polarity}, \sigma)$

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

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

Figure 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`. The $\rho[i_1,\ i_2]$ syntax applies the ρ predicate to given integer constants; the result is a boolean constant.

is computed as a disjunction of either being undefined. For all other arguments ($e_{\texttt{ext}}$) 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).

Following the formalization in Figure 6, evaluating an integer predicate ρ boils down to evaluating its arguments (e_1, and e_2), applying ρ to obtained integer values, and appending the definedness condition to the previous result. Since ρ is a built-in integer comparison predicate, it can be applied concretely to two given constants to obtain a concrete boolean constant (b) corresponding to the result of the comparison. If any of the arguments evaluates to $\perp$, the definedness condition appended in the next step will ensure that the concrete value of b becomes irrelevant.

The definedness condition function (**ensureDfn**) takes a boolean constant (b), a set of integer expressions ($e_{\texttt{in}}$), and a store. If all received expressions are defined, the result is b; otherwise the result is computed so that the associated binding in the enclosing quantifiers becomes irrelevant. Concretely, the **ensureDfn** function splits $e_{\texttt{in}}$ into two sets, $e_{\texttt{univ}}$ and $e_{\texttt{ext}}$, first containing expressions with universally quantified integer variables, and the second all others (which are implicitly considered as existentially quantified). The conditions associated with the existentially and universally quantified expressions ($b_{\texttt{def}}$ and $b_{\texttt{undef}}$) state that none of the expressions are undefined, or at least one is undefined, respectively. If the polarity is positive (the easiest way to think about it is the case when there are no enclosing negations), the final result is computed as $(b \vee b_{\texttt{undef}}) \wedge b_{\texttt{def}}$. Intuitively, if any existentially quantified expression is undefined ($b_{\texttt{def}}$ equal to `false`), the result must be `false` (since `false` is the value that makes a binding irrelevant inside an existential quantifier); if any universally quantified expression is undefined ($b_{\texttt{undef}}$ equal to `true`), the result must be `true` (since `true` makes a binding irrelevant inside a universal quantifier); otherwise, the result is exactly equal to the original value b. When the polarity is negative, the same reasoning applies, except that the conditions need to be flipped, so $b_{\texttt{def}}$ becomes $\neg b_{\texttt{undef}}$, and $b_{\texttt{undef}}$ becomes $\neg b_{\texttt{def}}$.

The helper functions used in the definition of **ensureDfn**, **isUnivQuant** and **vars** are straightforward. An expression is universally quantified if any of its variables (computed by a simple top-down algorithm embodied in the **vars** function) is quantified over by a universal quantifier (information about the enclosing quantifiers is directly accessible from the store).

Finally, the user-level semantics also allows relational expressions to be undefined, for example, when the int-to-expression cast operator is applied to an undefined integer expression. Under the user-level semantics, whenever a

boolean predicate is applied to a number of relational expressions, the result is undefined if at least one of its arguments is undefined; here, however, a truth value must be assigned for all cases. The way this is done is analogous to evaluating integer predicates (**ieval**), which we already formally defined.

5.3. 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., **dfn**$[p(x)]$ is `false`), if x is *universally* quantified then **beval**$[p(x)]$ evaluates to `true`, else it evaluates to `false`.

This hypothesis could be proved by a structural induction on expressions. Instead of giving a complete proof, we explain several interesting 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 **beval**$[\rho(e_1(x), e_2(x))]$ becomes the value of **ensureDfn**. There are two cases to consider: (1) if x is *universally* quantified, $e_{\texttt{univ}}$ 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, $e_{\texttt{ext}}$ 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 **beval**$[p(x)]$ now becomes a negation of the value returned by **ensureDfn**, 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 **ensureDfn** now returns `false` for the universal case, and `true` for the existential case (because of the negative polarity), so the end result of **beval**$[p(x)]$ remains the same, as expected.

Another class of interesting examples is those with nested quantifiers. Consider the following formula:

```
run {
  all x: Int | some y: Int | y = 3 and (x = 3 implies plus[x,x] = plus[y,y])
} for 3 Int
```

Applying the user-level semantics, this formula evaluates to

$$\begin{aligned}&[\![\texttt{all x:Int | some y:Int | f[x,y]}]\!]\\ &\quad = \forall \texttt{x} \in \texttt{Int} \setminus \{\texttt{x}_{\texttt{0F}}\} \bullet [\![\texttt{some y:Int | f[x,y]}]\!]\\ &\quad = \forall \texttt{x} \in \texttt{Int} \setminus \{\texttt{x}_{\texttt{0F}}\} \bullet \exists \texttt{y} \in \texttt{Int} \setminus \{\texttt{y}_{\texttt{0F}}\} \bullet [\![\texttt{f[x,y]}]\!]\end{aligned}$$

where `f[x,y]` is `y = 3 and (x = 3 implies plus[x,x] = plus[y,y])`. The set of excluded bindings for variable `x`, $\{\texttt{x}_{\texttt{0F}}\}$, is a set of all integers for which `plus[x,x]` overflows (which is the only subexpression containing variable `x` that can possibly be undefined); similarly, $\{\texttt{y}_{\texttt{0F}}\}$ is a set of all integers for which `plus[y,y]` overflows. In both cases, the excluded bindings are equal to $\{-3, 2, 3\}$. Since the only binding that satisfies `f[x,y]` is $x \to 3$, $y \to 3$, the formula as a whole is unsatisfiable. Now following the implementation-level semantics (the definition of the **`ensureDfn`** function), the $e_{\texttt{ext}}$ set contains `plus[y,y]`; given the binding $x \to 3$, $y \to 3$, $b_{\texttt{def}}$ evaluates to false, and since the polarity is positive ($b_{\texttt{pol}}$ is true), **`ensureDfn`** returns false, thus, the formula as a whole is, again, unsatisfiable.

As an exercise, the reader may want to check that if the quantifiers in the previous formula swap places, the result does not change. When `plus[x,y] = plus[x,y]` is used instead of `plus[x,x] = plus[y,y]`, the order of quantification does matter: the formula

```
all x: Int | some y: Int | y = 3 and (x = 3 implies plus[x,y] = plus[x,y])
```

is not satisfiable, but

```
some y: Int | all x: Int | y = 3 and (x = 3 implies plus[x,y] = plus[x,y])
```

is, because the two arithmetic expressions are both treated as "existentially" quantified in the former case, and "universally" in the latter. This behavior is not simply an artifact of our formalization. Rather, it is by design, as our intention was to have `all x: Int | some y: Int | plus[x,y] > x` be false (indeed, in a bounded setting, for `x = MAXINT`, there is no integer that can be added to it to obtain an integer greater than `MAXINT`), and `some y: Int | all x: Int | plus[x,y] > x` be true (for, e.g., `y = 1`, every integer `x` when added to it can only produce a number greater than `x`).

5.4. 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 (within the bitwidth of 3) 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.

Similarly, if we ask the Alloy Analyzer to find all instances of `x` where `x` multiplied by two is either less or not less than zero, we will not get all integers from the domain, but only those that do not overflow when multiplied by two.

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

instances

```
x = -2, x = 0
x = -1, x = 1
```

In a sense, however, the violation of the law of the excluded middle 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.

6. Implementation in Circuits

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.

The Alloy Analyzer delegates the core task of finding satisfying models to Kodkod [4], a bounded constraint solver for relational first-order logic. Kodkod 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.

Even though the goal here is to translate the input formula into a digital circuit (instead of evaluating it to a boolean constant), the denotational semantics we defined in Section 5.1 still applies, simply because all logic operators used in our formalization are also available at the level of digital gates. We only had to modify Kodkod's translation of appropriate terms, directly following the formal 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 if and only if the operation overflows. We used textbook overflow circuits for all arithmetic operations supported by Kodkod, and the definition in Figure 4 to propagate this information from the operands to the operation result;
- *the way the store gets updated* so that it additionally keeps track of the polarity and the quantification stack (the store is defined in Figure 3, and how it gets updated in Figure 5);
- *the translation of boolean predicates* so that the original circuit representing the predicate result is extended to include the definedness conditions, exactly as defined in Figure 6;

7. Evaluation

The goal of our evaluation is twofold: (1) test the new semantics as embodied in code within the Alloy Analyzer 4.2 and make sure that all the anomalies presented in Section 3 are fixed, as well as spurious counterexamples caused by integer overflows in several other prototypical cases are eliminated, and (2) provide some evidence about potential effects (in terms of scalability of the analysis) the new semantics might have on existing models already using integer arithmetics.

7.1. Exhaustive Testing of the New Translation Scheme

To ensure the correctness of the new semantics, as well as the implementation of the new translation scheme, we ran a series of exhaustive tests (up to a finite bound) and verified that the results were as expected.

7.1.1. Basic Arithmetic Tests

The unit test in Figure 7 exhaustively checks whether overflows are detected in all supported binary arithmetic functions. It dynamically constructs all possible expressions in the form of

```
ret = {i op j},
```

where `i` and `j` are integers drawn from {`-16`, . . . , `15`}, and `op` is an arithmetic operator drawn from {`+`, `-`, `*`, `/`, `%`}. For each case, it first computes the expected result using Java built-in arithmetic operators (which certainly will not overflow in this small scope). If the computed result falls outside of the `[-16, 15]` range, an overflow is expected, that is, no satisfying instance is

```
public void testArithmeticOverflows() {
 int bw = 5, l = -(1 << (bw - 1)), h = (1 << (bw - 1));
 IntOperator[] ops = new IntOperator[]{PLUS, MINUS, MULTIPLY, DIVIDE, MODULO};
 for (IntOperator op: ops) for (int i=l; i<h; i++) for (int j=l; j<h; j++) {
  // f: ret = Int[(i op j)]
  Formula f = ret.eq(compose(op, constant(i), constant(j)).toExpression());
  int javaRes = -1;
  try { javaRes = exeJava(op, i, j); } catch(ArithmeticException e){ continue; }
  if (javaRes >= h || javaRes < l) {
   try { exeKodkod(f); fail("Overflow not detected"); } catch(NoSolution e){}
  } else {
   assertEquals("Wrong result", javaRes, exeKodkod(f));
  }
 }
}
```

Figure 7: A Kodkod unit test that exhaustively checks overflow detection for all formulas in the form of `ret = {i op j}`, where `i` and `j` are integers drawn from {-16, ..., 15}, and `op` is an arithmetic operator drawn from {`+`, `-`, `*`, `/`, `%`}. Helper method `exeJava` computes the expected result. Method `exeKodkod` runs Kodkod to solve a given formula, then evaluates `ret` against the solution and converts the result to integer (if the formula was unsatisfiable, throws the `NoSolution` exception).

expected to be found by Kodkod; otherwise, the value of `ret` returned by Kodkod is expected to be equal to the value obtained in Java.

A slightly modified version of this test uses the same ideas to check all expressions in the form of

```
ret = (some {i op j} => {i op j} else {-1}).
```

This test shows that a constraint can be written to check whether an integer expression overflows. The formula above uses that feature to assign a default value (-1) to `ret` whenever `i op j` overflows. One might (wrongly) expect this entire formula to be unsatisfiable when `{i op j}` overflows; recalling the semantics, however, applying the int-to-expression cast operator to an overflowing integer expression (`i op j`) results in an undefined relational expression, then applying the **`some`** boolean predicate to it yields `false` (since the polarity is positive), which finally selects the else branch (regardless of the evaluation of the then branch).

Yet another variation of the same test uses relations in place of integer constants `i` and `j`, so that it can ask Kodkod to enumerate all valid solutions to `ret = {i op j}`. It then checks that the result exactly matches the set of all non-overflowing solutions computed in Java.

7.1.2. Testing Tautologies

Table 1 shows the list of arithmetic tautologies that we checked for counterexamples using Kodkod.

decl	precondition	postcondition
`a, b: Int`	`a > 0 && b > 0`	`a + b > 0 && a + b > a && a + b > b`
`a, b: Int`	`a < 0 && b < 0`	`a + b < 0 && a + b < a && a + b < b`
`a, b: Int`	`a > 0 && b < 0`	`a - b > 0 && a - b > a && a - b > b`
`a, b: Int`	`a < 0 && b > 0`	`a - b < 0 && a - b < a && a - b < b`
`a, b: Int`	`a > 0 && b > 0`	`a * b > 0 && a * b >= a && a * b >= b`
`a, b: Int`	`a < 0 && b < 0`	`a * b > 0 && a * b >= -a && a * b >= -b`
`a, b: Int`	`a > 0 && b < 0`	`a * b < 0 && -(a * b) > a && -(a * b) > -b`
`a, b: Int`	`a < 0 && b > 0`	`a * b < 0 && -(a * b) > -a && -(a * b) > b`

Table 1: List of checked arithmetic tautologies.

For the purpose of exercising various polarity cases (that is, nestings of negations and quantifiers), for each row from Table 1 we ran the test on the following equivalent formulas.

```
all decl | pre => post                !(some decl | !(pre => post))
!!(all decl | pre => post)            !!!(some decl | !(pre => post))
all decl | !!(pre => post)            !(some decl | pre && !post)
all decl | !(pre && !post)            !(some decl | !!(pre && !post))
all decl | !pre || post               !(some decl | !(!pre || post))
```

The cardinality operator (`#`) returns the number of elements in a given relation. If that number is greater than the largest integer in the scope, the operation overflows, often causing anomalies especially difficult to debug. To ensure that our new semantics correctly prevents overflows in those cases, we checked the following tautologies:

```
all s:    set univ | #s >= 0
no  s:    set univ | #s < 0
all s:    set univ | (some s) iff #s > 0
all s, t: set univ | #(s + t) >= #s && #(s + t) >= #t
all s, t: set univ | s in t => #s <= #t
all s, t: set univ | (no s & t && some s) => #(s + t) > #t
```

As expected, none of the tautologies could be refuted given the integer bitwidth of 5.

7.2. Effects on Models with Integer Arithmetic

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 [14], 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 11 (`run1` through `run11`) are simulations (which all find an instance), and the remaining 5 (`check1` through `check5`) 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 2. 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.

	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
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
x	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		run11		check1		check2		check3		check4		check5		**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
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
x	-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 2: Analysis times of all checks found in the "concrete" module of a flash filesystem from [14]. All values are in seconds, except the values in the "x" 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; "diff" is the difference between "old" and "new"; whereas "x" is the speedup in percents (in both cases negative values mean the version with overflow prevention turned on is faster).

8. 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.

The *Logic of Partial Functions* (LPF) was proposed for reasoning about the development of programs [9, 15], and was adopted in VDM [16]. 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 [11]), 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 [17] 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 [18]), 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 [19] and Z [20], 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 value [21] (consequently, propositions containing unknown values cannot be proved). The initial specification of the Z notation [20] left the handling of partial functions open.

Several different approaches have been proposed (see [22] for a survey); in the end, it appears that the same approach as in B has evolved to be the norm [21]. 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 [23]) are typically less automated than the Alloy Analyzer.

9. Conclusion

We have presented a new first-order logic that provides a special treatment of quantifiers to eliminate models yielding out-of-domain applications of partial functions. The main motivation for the new logic was making an automated analysis based on it sound with respect to counterexamples, even in the presence of partial functions. In this paper, we focused on applying this approach to integer functions (which in a bounded setting become partial), with the goal of excluding the models containing arithmetic overflows. We have extended the Alloy Analyzer—a bounded model finder based on a relational first-order logic—accordingly, and thus eliminated its only source of spurious counterexamples. Despite being focused on arithmetic overflows, our approach is more general and readily applicable to a broader class of partial functions.

Acknowledgments

This material is based upon work partially supported by the National Science Foundation under Grant No. CCF-1138967. We would like to thank Marc Frappier for carefully checking our formal semantics and suggesting numerous improvements. We also thank the anonymous reviewers for their thoughtful comments on the drafts of this paper.

[1] Alloy: A language and tool for relational models, http://alloy.mit.edu/alloy.

[2] D. Gries, F. Schneider, A logical approach to discrete math, Texts and monographs in computer science, Springer-Verlag, 1993.

[3] D. Jackson, Software Abstractions: Logic, language, and analysis, MIT Press, 2006.

[4] E. Torlak, A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications, Ph.D. thesis, MIT (2008).

[5] D. Marinov, S. Khurshid, TestEra: A Novel Framework for Automated Testing of Java Programs, in: ASE'01, 2001.

[6] G. Dennis, A relational framework for bounded program verification, Ph.D. thesis, MIT (2009).

[7] A. Milicevic, D. Rayside, K. Yessenov, D. Jackson, Unifying execution of imperative and declarative code, in: ICSE, 2011.

[8] T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, Introduction to Algorithms, 2nd Edition, McGraw-Hill Higher Education, 2001.

[9] C. B. Jones, Reasoning about partial functions in the formal development of programs, Electron. Notes Theor. Comput. Sci. 145.

[10] E. Torlak, D. Jackson, Kodkod: A relational model finder, in: O. Grumberg, M. Huth (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Vol. 4424 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2007.

[11] W. M. Farmer, Reasoning about partial functions with the aid of a computer, Erkenntnis 43.

[12] D. L. Parnas, Predicate logic for software engineering, IEEE Trans. Softw. Eng. 19.

[13] Jean-Yves, Girard, Linear logic, Theoretical Computer Science 50 (1).

[14] E. Kang, 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, Springer-Verlag, Berlin, Heidelberg, 2008.

[15] C. B. Jones, M. J. Lovert, Semantic Models for a Logic of Partial Functions, Int. J. Software and Informatics 5 (1-2).

[16] C. B. Jones, Systematic software development using VDM (2nd ed.), Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[17] C. Barrett, A. Stump, C. Tinelli, The SMT-LIB Standard: Version 2.0, Tech. rep., Department of Computer Science, The University of Iowa (2010).

[18] K. R. M. Leino, Dafny: An automatic program verifier for functional correctness, in: LPAR-16, Vol. 6355 of LNCS, Springer, 2010.

[19] J. Abrial, A. Hoare, The B-Book: Assigning Programs to Meanings, Cambridge University Press, 2005.

[20] J. Spivey, Understanding Z: a specification language and its formal semantics, Cambridge tracts in theoretical computer science, Cambridge University Press, 1988.

[21] B. Stoddart, S. Dunne, A. Galloway, Undefined Expressions and Logic in Z and B, Formal Methods in System Design 15.

[22] R. Arthan, L. Road, Undefinedness in Z: Issues for Specification and Proof, in: CADE-13 Workshop on Mechanization of Partial Functions, Springer, 1996.

[23] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in Event-B, STTT 12 (6).