A new relational solver for the Alloy Analyzer

Mudathir Mohamed, Baoluo Meng, Andrew Reynolds and Cesare Tinelli

†*Department of Computer Science*, *University of Iowa*, Iowa City, USA

***Abstract*—We present a new relation solver for Alloy Analyzer, the CVC4 Relational Solver (CRS), that translates Alloy models to SMT formulas over the theory of finite relations. It extends the Alloy Analyzer to prove properties on unbounded domains unlike Kodkod, the default solver of the Alloy Analyzer, which only works with bounded domains. CRS implements a new semantics over arithmetic operators on integer signatures which is simple, intuitive, and consistent with interpreting the applications of arithmetic operations as relational joins.**

I. Introduction

Alloy is a formal language for software specification based on relational logic that combines predicate calculus and relational calculus [1]. The *Alloy Analyzer* is a static analysis tool that translates Alloy specifications, or *models*, to a set of constraints in *predicate logic with relations* for the Kodkod solver [2]. Kodkod in turn translates its input to *propositional logic* formulas to be solved by off-the-self SAT solvers. While Kodkod is a very efficient solver, it needs concrete lower and upper bounds on the cardinality of each relational domain, called a *signature* in Alloy, thus requiring users to specify all bounds in advance, including a maximum bitwidth for bounded integers. As a result, model properties, or *assertions* in Alloy, can be verified only within the specified bounds.

Meng et al. [3] added the theory of finite relations to the SMT Solver CVC4, and used it to build an early prototype of CRS as an alternative to Kodkod. This prototype translates models in a restricted fragment of Alloy to SMT formulas without imposing artificial bound constraints on signatures.

We report here on our extension of CRS to support more Alloy features. These include modules, nested multiplicities, commands and integer signatures. We also propose a new semantics for arithmetic operators on integer signatures that we believe is more intuitive, and consistent with interpreting the application of arithmetic operations as relational joins.

CRS[1] is now integrated with Alloy Analyzer. It translates Alloy models to an extension of the SMT-LIB 2 format with *finite sets*, *tuples* and *relational operators* from the theory of finite relations. The solver transforms satisfying models found by CVC4 to Alloy model instances. It can return different instances when the constraints are satisfiable, and supports incremental solving when executing multiple commands.

II. Examples

The examples in Figures 1 and 2 demonstrate CRS features compared to Kodkod solver. In Figure 1, lines 1 and 2 define two top-level signatures (representing two disjoint domains) named A and B. Line 2 Also defines a field (or a relation) $f \subseteq A \times B$, the keyword **one** is a multiplicity constraint that restricts the relation f to be a function, and the keyword **disj** restricts the function f to be injective. Line 3 defines a fact (or an invariant) that restricts f to be surjective. Line 5 defines an assertion (a property) which is a formula we expect to logically follow from the model. Lines 8, 9 check the validity of the assertion when A and B have scope 3 (i.e., their cardinality is ≤ 3) and 5, respectively. Since Kodkod can disprove assertions only in bounded domains, it concludes there are no counter examples for the assertion within the given scopes. In contrast, CRS ignores the scope constraints by default and can prove in this case that the assertion is valid in unbounded scopes. It can also prove the validity in bounded scopes if the input option "*include scope*" is enabled.

```
1 sig B {}
2 sig A { f: disj one B } // f: A x B is injective
3 fact { f[A] = B }       // f is surjective
4 // no two distinct elements have the same image
5 assert assertion {
6   no disj x, y : A |  x != y and f[x] = f[y]
7 }
8 check assertion for 3 A, 3 B
9 check assertion for 5 A, 5 B
```

Fig. 1. Kodkok can prove valid assertions only in bounded scopes such scopes 3 and 5 for A and B in lines 10 and 11, respectively. CRS can prove valid assertions in an unbounded scope.

```
1  sig A, B, C in Int {}
2  fact {
3    A = 1 + 2       // A = {1}∪{2}
4    B = 4 + 5       // B = {4}∪{5}
5    C = plus[A, B]
6  }
7  one sig x, y in Int {}  // x, y are singletons
8  fact { plus[x, y] in C and minus[x, y] in C }
9  run {} for 6 Int
10 run {} for 3 Int
```

Fig. 2. In line 11, CRS returns $C = \{5, 6, 7\}, x = \{6\}, y = \{1\}$, whereas Kodkod returns $C = \{12\} = \{sum[A] + sum[B]\}, x = \{12\}, y = \{0\}$.

In Figure 2, line 1 defines three integer subsignatures A, B, and C (denoting possibly overlapping sets of integers). Lines 3 and 4 constraint A to be the set $\{1, 2\}$ and B the set $\{4, 5\}$. Note that, by design, Alloy does not provide ways to denote scalar values [1]. Each numeral denotes the singleton unary relation containing the number (e.g., 5 denotes the set $\{5\}$). Also note that the $+$ operator denotes set union.

[1]Available at https://github.com/CVC4/org.Alloytools.Alloy .

Line 5 restricts C to be the *addition* of sets A and B (explained below). Line 7 defines two integer singletons x, y, and line 8 constrains the sets resulting from the addition and the subtraction to be subsets of C.

Lines 9 and 10 define two *run commands* which attempt to find instances satisfying all constraints within integer scopes 6 and 3, respectively. Scope 6 in line 9 represents the maximum bitwidth for integers when expressed in two's complement (binary) notation [1]. This restricts integers to be in the range $[-32, +31]$. Similarly, scope 3 denotes the range $[-4, +3]$.

Kodkod applies the builtin *sum* function to the arguments of $plus$ before computing the result [1]. The sum function takes a set of integers as an argument and returns their sum [1]. In this example, $C = plus[A, B]$ is the same as $C = plus[sum[A], sum[B]]$. Since $sum[A] = 3$ and $sum[B] = 9$, then Kodkod returns $C = \{12\}$ for scope 6. For scope 3, 12 is outside the bitwidth range. So Kodkod finds no instance if the "*Prevent Overflows*" option is enabled, or returns a result modulo 2^w for the prescribed bitwidth w if it is disabled. In the latter, $C = \{-4\} =_8 \{12\}$ is returned.

CRS does not use *sum* for the *plus* operation. Instead, it follows the semantics described in Table I. In this semantics, *plus* is interpreted as a finite subset of the ternary relation corresponding to integer addition. In our example, CRS returns a model instance where $plus = \{(1, 4, 5), (1, 5, 6), (2, 4, 6), (2, 5, 7)\}$ and $C = \{5, 6, 7\}$. Since by default CRS imposes no bounds on integers, both commands at line 9 and 10 return the same instance. When the "*include scope*" option is enabled, CRS adds (in essence) the the constraints below to the commands in lines 9 and 10, respectively where *intuniv* is a builtin signature described in the next section. In that case, it returns $C = \{5, 6, 7\}$ for scope 6, and returns unsat for scope 3.

```
run {all z: intuniv | z>=-32 and z<=31} for 6 Int
run {all z: intuniv | z>=- 4 and z<= 3} for 3 Int
```

III. Translation to SMT

Alloy has a rich type systems that includes subtypes and union and intersection types. Roughly speaking, every signature is also a type. Non-integer signatures are all subtypes of a root type of (uninterpreted) *atoms*. The *Int* signature is a builtin signature denoting the mathematical integers. The language provides the builtin unary signature *univ* which, for each Alloy model, is the union of *Int* and all the top-level signatures in the model. CRS conforms to the Alloy type system with the exception of prohibiting the construction of expressions that mix integer and non-integer elements. For example, the following predicate is considered ill-typed by CRS, whereas it is well-typed (and always true) in standard Alloy.

```
pred p {univ & Int = Int} // Int is a subset of univ
```

This facilitates the translation to CVC4, whose type systems has only simple types, and does not appear to be a major restriction in common usage [4]. Another notable, albeit mostly nominal, difference is that CRS treats all user-defined signatures as finite sets. CRS translates all non-integer signatures as finite sets of elements of a single uninterpreted type called *Atom*. Signature hierarchies are translated using *subset* and *disjoint union* constraints. Integer signatures are translated as finite sets of elements of an uninterpreted type *UInt* equipped with an injective mapping to the builtin type $\mathbb{Z}$. The rationale behind using *UInt* instead of $\mathbb{Z}$ is to avoid universal quantification over the infinite set $\mathbb{Z}$. CRS interprets *univ* as the universe set for the type *Atom* and introduces the new builtin signature *intuniv* to denote the finite universe set for *UInt*. Based on this translation, CRS supports quantified expressions over *intuniv* but not over *Int*.

TABLE I
SEMANTICS OF FUNCTIONS PLUS, DIV, AND REM (MINUS AND MUL ARE DEFINED ANALOGOUSLY).

Operation	Join syntax	Meaning
$plus[A, B]$	$B.A.plus$	$\{z \mid \exists\, x \in A, y \in B \,.\, x + y = z\}$
$div[A, B]$	$B.A.div$	$\{z \mid \exists\, x \in A, y \in B \,.\, x/y = z\}$
$rem[A, B]$	$B.A.rem$	$\{z \mid \exists\, x \in A, y \in B \,.\, x \textbf{ mod } y = z\}$

IV. Semantics of Integer Arithmetic

CRS interprets the builtin arithmetic operations *plus, minus, mul, div,* and *rem* differently from standard Alloy, and more consistently with the use of function application in Alloy as syntactic sugar for relational joins. Specifically, it uses the semantics in Table I where *plus, minus, mul, div,* and *rem* are interpreted as ternary relations in $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.

For comparison operators, which apply to two *sets* of integers, Alloy prescribes the application of the *sum* function to the arguments before the comparison. In contrast, the semantics of a comparison operator $op \in \{<, \leq, >, \geq\}$ in CRS is based on singletons as follows:

$$[\![A \; op \; B]\!] = \exists\, x, y \in \mathbb{Z}.\; A = \{x\} \wedge B = \{y\} \wedge (x \; op \; y)$$

CRS evaluates $A < B$ in Figure 2 as *false* since both A and B are not singletons, while Kodkod evaluates it as $3 < 9$.

V. Conclusion and Future Work

Currently, Kodkod has superior performance in the bounded case, especially in models with large relations and transitive closures. We are working on improving CRS's performance by enhancing the encoding to SMT, and further optimizing the relational solver in CVC4.

CRS supports most Alloy constructs. For cardinality constraints, it accepts only the comparison of a relation's cardinality to a numeral. Supporting general cardinality constraints is a longer term goal because it will require a corresponding extension of CVC4's relational solver.

References

[1] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[2] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Proceedings of TACAS 2007*, ser. LNCS, vol. 4424. Springer, 2007.

[3] B. Meng, A. Reynolds, C. Tinelli, and C. Barrett, "Relational constraint solving in SMT," in *Proceedings of CADE-26*, ser. LNCS, vol. 10395. Springer, 2017.

[4] D. Jackson, Jun. 2018, personal communication.