Alloy*: A General-Purpose Higher-Order Relational Constraint Solver

Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang and Daniel Jackson
Massachusetts Institute of Technology, Cambridge, MA 02139, USA
Email: {aleks,jnear,eskang,dnj}@csail.mit.edu

Abstract—**The last decade has seen a dramatic growth in the use of constraint solvers as a computational mechanism, not only for analysis of software, but also at runtime. Solvers are available for a variety of logics but are generally restricted to first-order formulas. Some tasks, however, most notably those involving synthesis, are inherently higher order; these are typically handled by embedding a first-order solver (such as a SAT or SMT solver) in a domain-specific algorithm.**

Using strategies similar to those used in such algorithms, we show how to extend a first-order solver (in this case Kodkod, a model finder for relational logic used as the engine of the Alloy Analyzer) so that it can handle quantifications over higher-order structures. The resulting solver is sufficiently general that it can be applied to a range of problems; it is higher order, so that it can be applied directly, without embedding in another algorithm; and it performs well enough to be competitive with specialized tools. Just as the identification of first-order solvers as reusable backends advanced the performance of specialized tools and simplified their architecture, factoring out higher-order solvers may bring similar benefits to a new class of tools.

I. Introduction

As constraint solvers become more capable, they are increasingly being applied to problems previously regarded as intractable. Program synthesis, for example, requires the solver to find a single program that computes the correct output for all possible inputs. This "$\exists\forall$" quantifier pattern is a particularly difficult instance of higher-order quantification, and no existing general-purpose constraint solver can reliably provide solutions for problems of this form.

Instead, tools that rely on higher-order quantification use ad hoc methods to adapt existing solvers to the problem. A popular technique for the program synthesis problem is called CEGIS (counterexample guided inductive synthesis) [1], and involves using a first-order solver in a loop: first, to find a candidate program, and second, to verify that it satisfies the specification for all inputs. If the verification step fails, the resulting counterexample is transformed into a constraint that is used in generating the next candidate.

In this paper, we present Alloy*, a general-purpose, higher-order, bounded constraint solver based on the Alloy Analyzer [2]. Alloy is a specification language combining first-order logic with relational algebra; the Alloy Analyzer performs bounded analysis of Alloy specifications. Alloy* admits higher-order quantifier patterns, and uses a general implementation of the CEGIS loop to perform bounded analysis. It retains the syntax of Alloy, and changes the semantics only by expanding the set of specifications that can be analyzed, making it easy for existing Alloy users to adopt.

Alloy* handles higher-order quantifiers in a generic and model-agnostic way, meaning that it allows higher-order quantifiers to appear anywhere where allowed by the Alloy syntax, and does not require any special idiom to be followed. Alloy* first creates a *solving strategy* by decomposing an arbitrary formula (possibly containing nested higher-order quantifiers) into a tree of subformulas and assigning a decision procedure to each of them. Each such tree is either (1) a higher-order "$\exists\forall$" pattern, (2) a disjunction where at least one disjunct is higher-order, or (3) a first-order formula. To solve the "$\exists\forall$" nodes, Alloy* applies CEGIS; for the disjunction leaves, Alloy* solves each disjunct separately; and for first-order formulas, Alloy* uses Kodkod [3].

To our knowledge, Alloy* is the first general-purpose constraint solver capable of solving formulas with higher-order quantification. Existing solvers either do not admit such quantifiers, or fail to produce a solution in most cases. Alloy*, by contrast, is both sound and complete for the given bounds, and still efficient for many practical purposes.

We have evaluated Alloy* on a variety of case studies taken from the work of other researchers. In the first, we used Alloy* to solve classical higher-order NP-complete graph problems like `max-clique`, and found it to scale well for uses in modeling, bounded verification, and fast prototyping. In the second, we encoded all of the SyGuS [4] program synthesis benchmarks that do not require bit vectors, and found that, while state-of-the-art purpose-built synthesizers are typically faster, Alloy* beats all the reference synthesizers provided by the competition organizers.

The contributions of this paper include:

- A framework for extending a first-order solver to the higher-order case, consisting of the design of datatypes and a general algorithm comprising syntactic transformations (skolemization, conversion to negation normal form, etc.) and an incremental solving strategy;
- A collection of case study applications demonstrating the feasibility of the approach in different domains (including synthesis of code, execution and bounded verification of higher-order NP-hard algorithms), and showing encouraging performance on standard benchmarks;
- The release of a freely available implementation for others to use, comprising an extension of Alloy [5].

```
1  some sig Node  {val: one Int}
2  // between every two nodes there is an edge
3  pred clique[edges: Node -> Node, clq: set Node] {
4    all disj n1, n2: clq | n1 -> n2 in edges }
5  // no other clique with more nodes
6  pred maxClique[edges: Node -> Node, clq: set Node] {
7    clique[edges, clq]
8    no clq2: set Node | clq2!=clq and clique[edges,clq2] and #clq2>#clq }
9  // symmetric and irreflexive
10 pred edgeProps[edges: Node -> Node] {
11   (~edges in edges) and (no edges & iden) }
12 // max number of edges in a (k+1)-free graph with n nodes is (k-1)n^2/2k
13 check Turan {
14   all edges: Node -> Node | edgeProps[edges] implies
15     some mClq: set Node {
16       maxClique[edges, mClq]
17       let n = #Node, k = #mClq, e = (#edges).div[2] |
18         e <= k.minus[1].mul[n].mul[n].div[2].div[k] }
19 } for 7 but 0..294 Int
```

Figure 1. Automatic checking of *Turan's* theorem in Alloy*.

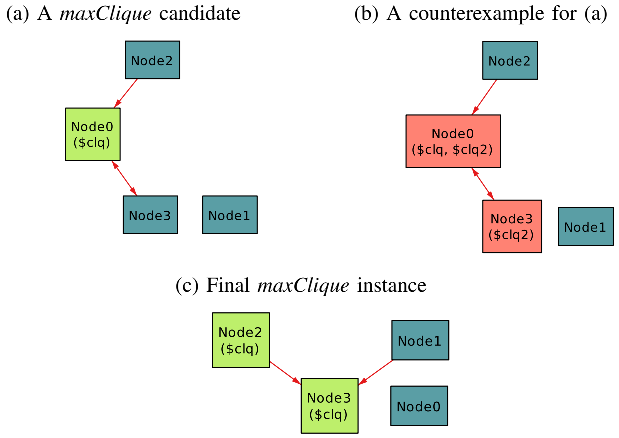

Figure 2. An automatically generated instance satisfying *maxClique*.

II. Example: Classical Graph Algorithms

Classical graph algorithms have become prototypical Alloy examples, showcasing both the expressiveness of the Alloy language and the power of the Alloy Analyzer. Many complex problems can be specified declaratively in only a few lines of Alloy, and then in a matter of seconds fully automatically animated (for graphs of small size) by the Alloy Analyzer. This ability to succinctly specify and quickly solve problems like these—algorithms that would be difficult and time consuming to implement imperatively using traditional programming languages—has found its use in many applications, including program verification [6], [7], software testing [8], [9], fast prototyping [10], [11], teaching [12], etc.

For a whole category of interesting problems, however, the current Alloy engine is not powerful enough. Those are the higher-order problems, for which the specification has to quantify over relations rather than scalars. Many well-known graph algorithms fall into this category, including finding maximal cliques, max cuts, minimum vertex covers, and various coloring problems. In this section, we show such graph algorithms can be specified and analyzed using the new engine implemented in Alloy*.

Suppose we want to check Turán's theorem, one of the fundamental results in graph theory [13]. Turán's theorem states that a $(k+1)$-free graph with n nodes can maximally have $\frac{(k-1)n^2}{2k}$ edges. A graph is $(k+1)$-free if it contains no clique with $k+1$ nodes (a clique is a subset of nodes in which every two nodes are connected by an edge).

Figure 1 shows how Turán's theorem might be formally specified in Alloy. A signature is defined to represent the nodes of the graph (line 1). Next, the clique property is embodied in a predicate (lines 3–4): for a given `edge` relation and a set of nodes `clq`, every two different nodes in `clq` are connected by an edge; the `maxClique` predicate (lines 6–8) additionally asserts that no other clique contains more nodes.

Having defined maximal cliques in Alloy, we can proceed to formalize Turán's theorem. The `Turan` command (lines 19–25) asserts that for all possible `edge` relations that are symmetric and irreflexive (line 20), if the max-clique in that graph has k nodes (`k=#mClq`), the number of selected edges (`e=(#edges).div[2]`) must be at most $\frac{(k-1)n^2}{2k}$. (The number of tuples in `edges` is divided by 2 because the graph in the setup of the theorem in undirected.)

Running the `Turan` command was previously not possible. Although the specification, as given in Figure 1, is allowed by the Alloy language, trying to execute it causes the Analyzer to immediately return an error: "Analysis cannot be performed since it requires higher-order quantification that could not be skolemized". In Alloy*, in contrast, this check can be automatically performed to confirm that indeed no counterexample can be found within the specified scope. The scope we used (7 nodes, ints from 0 to 294) allows for all possible undirected graphs with up to 7 nodes. The upper bound for ints was chosen to ensure that the formula for computing the maximal number of edges ($\frac{(k-1)n^2}{2k}$) never overflows for $n \leq 7$ (which implies $k \leq 7$). The check completes in about 45 seconds.

To explain the analysis problems that higher-order quantifiers pose to the standard Alloy Analyzer, and how those problems are tackled in Alloy*, we look at a simpler task: finding an instance of a graph with a subgraph satisfying the `maxClique` predicate. The problematic quantifier in this case is the inner "`no clq2: set Node | ..`." (line 8) constraint, which requires checking that for all possible subsets of `Node`, not one of them is a clique with more nodes than the given set `clq`. A direct translation into the current SAT-based backend would require the Analyzer to explicitly, and upfront, enumerate all possible subsets of `Node`—which would be prohibitively expensive. Instead, Alloy* implements the CEGIS approach:

1) First, it finds a candidate instance, by searching for a clique `clq` and *only one* set of nodes `clq2` that *is not* a clique larger than `clq`. A possible first candidate is given in Figure 2(a) (with the clique nodes are highlighted in green). At this point `clq2` could have been anything that is either not a clique or not larger than `clq`.
2) Next, Alloy* attempts to falsify the previous candidate by finding, again, *only one* set of nodes `clq2`, but this

time such that `clq2` *is* a clique larger than `clq`, for the exact (concrete) graph found in the previous step. In this case, it finds one such *counterexample* clique (red nodes in Figure 2(b)) refuting the proposition that `clq` from the first step is a maximal clique.

3) Alloy* continues by trying to find another candidate clique, encoding the previous counterexample to prune the remainder of the search space (as explained in detail in Sections III and IV). After several iterations, it finds the candidate in Figure 2(c) which cannot be refuted, so it returns that candidate as a satisfying solution.

Once written, the `maxClique` predicate (despite containing a higher-order quantification) can be used in other parts of the model, like any other predicate, just as we used it to formulate and check Turán's theorem. In fact, the `turan` check contains another higher-order quantifier, so the analysis ends up spawning two nested CEGIS loops and exhaustively iterating over them; every candidate instance and counterexample generated in the process can be opened and inspected in the Alloy visualizer. (For a screenshot of the user interface, see [5].)

III. Background and Key Ideas

Skolemization Many first-order constraint solvers allow some form of higher-order quantifiers to appear at the language level. Part of the reason for this is that, in certain cases, quantifiers can be eliminated in a preprocessing step called *skolemization*. In a model finding setting, every top-level existential quantifier is eliminated by (1) introducing a *skolem constant* for the quantification variable, and (2) replacing every occurrence of that variable with the newly created skolem constant. For example, solving `some s: set univ | #s > 2`, which is higher-order, is equivalent to solving `$s in univ && #$s > 2`, which is first-order and thus solvable by general purpose constraint solvers. (Throughout, following the Alloy convention, skolem constants will be prefixed with a dollar sign.)

CEGIS CounterExample-Guided Inductive Synthesis [1] is an approach for solving higher-order synthesis problems, which is extended in Alloy* to the general problem of solving higher-order formulas. As briefly mentioned before, the CEGIS strategy applies only to formulas in the form $\exists p \forall e \bullet s(p, e)$ and prescribes the following three steps:

(1) search: attempt to find a candidate value for p by solving $\exists p \exists e \bullet s(p, e)$—a first-order problem;

(2) verification: if a candidate $\$p$ is found, try to verify it by checking if it holds for all possible bindings for e. The *verification condition*, thus, becomes $\forall e \bullet s(\$p, e)$. This check is done by refutation, i.e., by satisfying the negation of the verification condition; pushing the negation through yields $\exists e \bullet \neg s(\$p, e)$, which, again, is first-order.

(3) induction: if the candidate is verified, a solution is found and the algorithm terminates. Otherwise a concrete counterexample $\$e_{cex}$ is found. The search continues by searching for another candidate which must also satisfy the counterexample, that is, solving $\exists p \exists e \bullet s(p, e) \wedge s(p, \$e_{cex})$. This strategy in particular tends to be very effective at reducing the search space and improving the overall scalability.

CEGIS for a general purpose solver Existing CEGIS-based synthesis tools implement this strategy internally, optimizing for the target domain of synthesis problems. The key insight of this paper is that the CEGIS algorithm can be implemented, generically and efficiently, inside a general purpose constraint solver. For an efficient implementation, it is important that such a solver supports the following:

- *Partial Instances*. The verification condition must be solved against the previously discovered candidate; explicitly designating that candidate as a "partial instance", i.e., a part of the solution known upfront, is significantly more efficient than encoding it with constraints [3].
- *Incremental solving*. Except for one additional constraint, the induction step solves exactly the same formula as the search step. Many modern SAT solvers already allow new constraints to be added to already solved propositional formulas, making subsequent runs more efficient (because all previously learned clauses are readily reusable).
- *Atoms as expressions*. The induction step needs to be able to convert a concrete counterexample (given in terms of concrete atoms, i.e., values for each variable) to a formula to be added to the candidate search condition. All atoms, therefore, must be convertible to expressions. This is trivial for SAT solvers, but requires extra functionality for solvers offering a richer input language.
- *Skolemization*. Skolemizing higher-order existential quantifiers is necessary for all three CEGIS steps.

We formalize our approach in Section IV, assuming availability of a first-order constraint solver offering all the features above. In Section V we present our implementation as an extension to Kodkod [14] (a first-order relational constraint solver already equipped with most of the required features).

IV. Semantics

We give the semantics of our decision procedure for bounded higher-order logic in two steps. First, we formalize the translation of a boolean formula into a `Proc` datatype instance (corresponding to an appropriate *solving procedure*); next we formalize the semantics of `Proc` satisfiability solving.

Figure 3 gives an overview of all syntactic domains used throughout this section. We assume the datatypes in Figure 3(b) are provided by the solver; on top of these basic datatypes, Alloy* defines the following additional datatypes:

- `FOL`—a wrapper for a first-order formula
- `OR`—a compound type representing a disjunction of `Procs`
- `∃∀`—a compound type representing a conjunction of a first-order formula and a number of higher-order universal quantifiers (each enclosed in a `QP` datatype). The intention of the `QP` datatype is to hold the original formula, and a translation of the same formula but quantified existentially (used later to find candidate solutions).

Figure 4(a) lists all the semantic functions defined in this paper. The main two are translation of formulas into `Procs` ($\mathcal{T}$, defined in Figure 5) and satisfiability solving ($\mathcal{S}$, defined in Figure 6). Relevant functions assumed to be either exported

(a) Alloy* syntactic domains

```
QP       = QP(forAll: Quant, pExists: Proc)
Proc     = FOL(form: Formula)
         | OR(disjs: Proc[])
         | ∃∀(conj: FOL, qps: QP[])
```

(b) Solver data types

```
Mult     = ONE | SET
Decl     = Decl(mult: Mult, var: Expr)
QuantOp  = ∀ | ∃
BinOp    = ∧ | ∨ | ⟺ | ⟹
Formula  = Quant(op: QuantOp, decl: Decl, body: Formula)
         | BinForm(op: BinOp, lhs, rhs: Formula)
         | NotForm(form: Formula)
         | ...
Expr     = ... // relational expressions, irrelevant here
```

Figure 3. Overview of the syntactic domains.

(a) Semantic functions

$\mathcal{T}$:	`Formula→ Proc`	top-level formula translation
$\mathcal{S}$:	`Proc → Instance`	`Proc` evaluation (solving)
τ :	`Formula→ Proc`	intermediate formula translation
$\curlywedge$:	`Proc→ Proc→ Proc`	`Proc` composition: conjunction
$\curlyvee$:	`Proc→ Proc→ Proc`	`Proc` composition: disjunction

(b) Functions exported by first-order solver

solve	: `Formula → Instance option`	first-order solver
eval	: `Instance → Expr → Value`	evaluator
replace	: `Formula → Expr → Value → Formula`	
nnf	: `Formula → Formula`	NNF conversion
skolemize	: `Formula → Formula`	skolemization
$\wedge$	: `Formula → Formula → Formula`	conjunction
$\vee$	: `Formula → Formula → Formula`	disjunction
`TRUE`	: `Formula`	true formula
`FALSE`	: `Formula`	false formula

(c) Built-in functions

fold	: `(A → E → A) → A → E[] → A`	functional fold
reduce	: `(A → E → A) → E[] → A`	fold w/o init value
map	: `(E → T) → E[] → T[]`	functional map
length	: `E[] → int`	list length
hd	: `E[] → E`	list head
tl	: `E[] → E[]`	list tail
+	: `E[] → E[] → E[]`	list concatenation
×	: `E[] → E[] → E[]`	list cross product
fail	: `String → void`	runtime error

Figure 4. Overview of used functions: (a) semantic functions, (b) functions provided by the first-order solver, (c) built-in functions.

by the solver or provided by the host programming language are listed in Figures 4(b) and 4(c), respectively.

For simplicity of exposition, we decided to exclude the treatment of bounds from our formalization, as it tends to be mostly straightforward; we will, however, come back to this point and accurately describe how the bounds are constructed before every solver invocation.

```
𝒯 : Formula→ Proc
1.  $\mathcal{T}[\![f]\!]$          ≡ $\tau[\![skolemize\ nnf\ f]\!]$

τ : Formula→ Proc
2.  $\tau[\![f_1 \vee f_2]\!]$  ≡ $\mathcal{T}[\![f_1]\!] \curlyvee \mathcal{T}[\![f_2]\!]$
3.  $\tau[\![f_1 \wedge f_2]\!]$  ≡ $\tau[\![f_1]\!] \curlywedge \tau[\![f_2]\!]$
4.  $\tau[\![\exists d \mid f]\!]$  ≡ fail "can't happen"
5.  $\tau[\![\forall d \mid f]\!]$  ≡ let $p$ = $\mathcal{T}[\![\exists d \mid f]\!]$ in
6.                  if $d.mult$ is ONE && $p$ is FOL then
7.                    FOL($\forall d \mid f$)
8.                  else
9.                    ∃∀(FOL(TRUE), [QP($\forall d \mid f$, $p$)])
10. $\tau[\![f]\!]$           ≡ FOL($f$)

⋏ : Proc → Proc → Proc
11. $p_1 \curlywedge p_2$      ≡ match $p_1$, $p_2$ with
12. | FOL, FOL → FOL($p_1.form \wedge p_2.form$)
13. | FOL, OR  → OR($map(\lambda_p \cdot p_1 \curlywedge p,\ \ p_2.disjs)$)
14. | FOL, ∃∀  → ∃∀($p_1 \curlywedge p_2.conj,\ \ p_2.qps$)
15. | OR , OR  → OR($map(\lambda_{p,q} \cdot p \curlywedge q,\ \ p_1.disjs \times p_2.disjs)$)
16. | OR , ∃∀  → OR($map(\lambda_p \cdot p \curlywedge p_2,\ \ p_1.disjs)$)
17. | ∃∀ , ∃∀  → ∃∀($p_1.conj \curlywedge p_2.conj,\ \ p_1.qps + p_2.qps$)
18. | _  , _   → $p_2 \curlywedge p_1$

⋎ : Proc → Proc → Proc
19. $p_1 \curlyvee p_2$      ≡ match $p_1$, $p_2$ with
20. | FOL, FOL → OR([$p_1$, $p_2$]) //wrong: FOL($p_1.form \vee p_2.form$)
21. | FOL, OR  → OR([$p_1$] + $p_2.disjs$)
22. | FOL, ∃∀  → OR([$p_1$, $p_2$])
23. | OR , OR  → OR($p_1.disjs + p_2.disjs$)
24. | OR , ∃∀  → OR($p_1.disjs$ + [$p_2$])
25. | ∃∀ , ∃∀  → OR([$p_1$, $p_2$])
26. | _  , _   → $p_2 \curlyvee p_1$
```

Figure 5. Translation of boolean `Formulas` to `Procs`.

Syntax notes Our notation is reminiscent of F#. We use the "." syntax to refer to field values of datatype instances. If the left-hand side in such constructs resolves to a list, we assume the operation is mapped over the entire list (e.g., *ea.qps.forAll*, is equivalent to *map* $\lambda_q \cdot q.forAll$, *ea.qps*).

A. Translation of Formulas into Proc Objects

The top-level translation function ($\mathcal{T}$, Figure 5, line 1) ensures that the formula is converted to negation normal form (NNF), and that all top-level existential quantifiers are subsequently skolemized away, before the formula is passed to the τ function. Conversion to NNF pushes the quantifiers towards the roots of the formula, while skolemization eliminates top-level existential quantifiers (including the higher-order ones). Alloy* applies these techniques aggressively to achieve completeness in handling arbitrary formulas.

Translating a binary formula (which is either a conjunction or disjunction, since it is in NNF) involves translating both left-hand and right-hand sides and composing the resulting `Procs` using the corresponding composition operator (lines 2–4). A disjunction demands that both sides be skolemized again (thus

```
$\mathcal{S}$ : Proc→ Instance option
27. $\mathcal{S}\llbracket p \rrbracket$ ≡ match $p$ with
28.         | FOL→ solve $p.form$
29.         | OR  → if length $p.disjs$ = 0 then None else match $\mathcal{S}\llbracket hd\ p.disjs \rrbracket$ with
30.                                                  | None → $\mathcal{S}\llbracket$OR($tl\ p.disjs$)$\rrbracket$ | Some(inst) → Some(inst)
31.         | ∃∀  → let $p_{cand}$ = fold ∧, $p.conj$, $p.qps.pExists$ in
32.                 match $\mathcal{S}\llbracket p_{cand} \rrbracket$ with
33.                 | None       → None
34.                 | Some(cand) → let $f_{check}$ = fold ∧, TRUE, $p.qps.forAll$ in
35.                                match $\mathcal{S}\llbracket \mathcal{T}\llbracket \neg f_{check} \rrbracket \rrbracket$ with
36.                                | None      → Some(cand)
37.                                | Some(cex) → fun repl($q$) = replace($q.body$, $q.decl.var$, eval(cex, $q.decl.var$))
38.                                              let $f^*_{cex}$ = map repl, $p.qps.forAll$ in
39.                                              let $f_{cex}$ = fold ∧, TRUE, $f^*_{cex}$ in $\mathcal{S}\llbracket p_{cand} \wedge \mathcal{T}\llbracket f_{cex} \rrbracket \rrbracket$
```

Figure 6. The model finding algorithm. The resulting Instance object encodes the solution, if one is found.

the use of $\mathcal{T}$ instead of τ), since they were surely unreachable by any previous skolemization attempts. This ensures that any higher-order quantifiers found in a clause of a disjunction will eventually either be skolemized or converted to an ∃∀ Proc.

A first-order universal quantifier (determined by $d.mult$ being equal to ONE) whose body is also first-order (line 6) is simply enclosed in a FOL Proc (line 7). Otherwise, an ∃∀ Proc is returned, wrapping both the original formula $(\forall d \mid f)$ and the translation of its existential counterpart ($p = \mathcal{T}\llbracket \exists d \mid f \rrbracket$, used later to find candidate solutions).

In all other cases, the formula is wrapped in FOL (line 10).

Composition of Procs is straightforward for the most part, directly following the distributivity laws of conjunction over disjunction and vice versa. The common goal in all the cases in lines 11–26 is to reduce the number of Proc nodes. For example, a conjunction of two ∃∀ nodes can be merged into a single ∃∀ node (line 17), as can a conjunction of a FOL and an ∃∀ node (line 14). With disjunction, however, we need to be more careful. Remember that before applying the ∨ operator skolemization had to be performed on both sides (line 2); since skolemization through disjunction is not sound, it would be wrong, for example, to try and recombine two FOL Procs into a single FOL (line 20). Instead, a safe optimization would be to modify line 2 to first check if $f_1 \vee f_2$ is first-order as a whole, and if so return FOL($f_1 \vee f_2$).

B. Satisfiability Solving

The procedure for satisfiability solving is given in Figure 6.

A first-order formula (enclosed in FOL) is given to the solver to be solved directly, in one step (line 28).

An OR Proc is solved by iteratively solving its disjuncts (lines 29–30). An instance is returned as soon as one is found; otherwise, None is returned.

The procedure for the ∃∀ Procs implements the CEGIS loop (lines 31–39), following the algorithm in Section III. The candidate condition is a conjunction of the first-order $p.conj$ Proc and all the existential Procs from $p.qps.pExists$ (line 31); the verification condition is a conjunction of all original universal quantifiers within this ∃∀ (line 34). Encoding the counterexample back into the search formula boils down to obtaining a concrete value that each quantification variable has in that counterexample (by calling the *eval* function exported by the solver) and embedding that value directly in the body of the corresponding quantifier (lines 37-39).

C. Treatment of Bounds

Bounds are a required input of any bounded analysis; for an analysis involving structures, the bounds may include not only the cardinality of the structures, but may also indicate that a structure includes or excludes particular tuples. Such bounds serve not only to finitize the universe of discourse and the domain of each variable, but may also specify a *partial instance* that embodies information known upfront about the solution to the constraint. If supported by the solver, specifying the partial instance through bounds (as opposed to enforcing it with constraints) is an important mechanism that generally improves scalability significantly.

Although essential, the treatment of bounds in Alloy* is mostly straightforward—including it in the above formalization (Figures 5 and 6) would only clutter the presentation and obscure the semantics of our approach. Instead, we informally provide the relevant details next.

Bounds may change during the translation phase by means of skolemization: every time an existential quantifier is skolemized, a fresh variable is introduced and a bound for it is added. Therefore, we associate bounds with Procs, as different Procs may have different bounds. Whenever a composition of two Procs is performed, the resulting Proc gets the union of the two corresponding bounds.

During the solving phase, whenever the *solve* function is applied (line 28), bounds must be provided as an argument. We simply use the bounds associated with the input Proc (p). When supplying bounds for the translation of the verification condition ($\mathcal{T}\llbracket \neg f_{check} \rrbracket$, line 37), it is essential to encode the candidate solution (*cand*) as a partial instance, to ensure that the check is performed against that particular candidate, and not some other arbitrary one. This is done by bounding every variable from $p.bounds$ to the exact value it was given in *cand*:

```
fun add_bound($b, var$) = $b + r \mapsto eval(cand,\ var)$
$b_{check}$ = $fold$ add_bound, $p.bounds$, $p.bounds.variables$
```

Finally, when translating the formula obtained from the counterexample (f_{cex}) to be used in a search for the next candidate (line 39), the same bounds are used as for the current candidate ($p_{cand}.bounds$).

V. Implementation

We implemented our decision procedure for higher-order constraint solving as an extension to Kodkod [14]. Kodkod, the backend engine used by the Alloy Analyzer, is a bounded constraint solver for *relational* first-order logic (thus, 'variable', as used previously, translates to 'relation' in Kodkod, and 'value' translates to 'tuple set'). It works by translating a given relational formula (together with bounds finitizing relation domains) into an equisatisfiable propositional formula and using an of-the-shelf SAT solver to check its satisfiability. The Alloy Analyzer delegates all its model finding (constraint solving) tasks to Kodkod. No change was needed to the existing translation from Alloy to Kodkod.

The official Kodkod distribution already offers most of the required features identified in Section III. While efficient support for partial instances has always been an integral part of Kodkod, only the latest version (2.0) comes with incremental SAT solvers. Kodkod performs skolemization of top-level (including higher-order) existential quantifiers.

Conversion from atoms to expressions, however, was not available in Kodkod prior to this work. Being able to treat all atoms from a single domain as indistinguishable helps generate a stronger symmetry-breaking predicate. We extended Kodkod with the ability to create a singleton relation for each declared atom, after which converting atoms back to expressions (relations) becomes trivial. We also updated the symmetry-breaking predicate generator to ignore all such singleton relations that are not explicitly used. As a result, this modification does not seem to incur any performance overhead; we ran the existing Kodkod test suite with and without the modification and observed no time difference (in both cases running the 249 tests took around 230s).

Our Java implementation directly follows the semantics defined in Figures 5 and 6. Additionally, it performs the following important optimizations: (1) rather than creating an OR node for every disjunction (line 2, Figure 5), it first checks if the disjunction is a first-order formula as a whole, in which case it creates a FOL node instead, and (2) it uses incremental solving to implement line 39 from Figure 6 whenever possible.

VI. Case Study: Program Synthesis

Program synthesis is one of the most popular applications of higher-order constraint solving. The goal is to produce a program that satisfies a given (high-level) specification. The SyGuS [4] (syntax-guided synthesis) project has proposed an extension to SMTLIB for encoding such problems. The project has also organized a competition between solvers for the format, and provides three reference solvers.

We encoded a subset of the SyGuS benchmarks in Alloy* to test its expressive power and scalability. These benchmarks have a standard format, are well tested, and allow comparison to the performance of the reference solvers, making them a good target for evaluating Alloy*.

To demonstrate our strategy for encoding program synthesis problems in Alloy*, we present the Alloy* specification for the problem of finding a program to compute the maximum of two numbers (the max-2 benchmark).

We encode the max-2 benchmark in Alloy* using signatures to represent the production rules of the program grammar, and predicates to represent both the semantics of programs and the constraints restricting the target program's semantics. Programs are composed of abstract syntax nodes, which can be integer- or boolean-typed.

```
abstract sig Node {}
abstract sig IntNode, BoolNode extends Node {}
abstract sig Var               extends IntNode {}
one      sig X, Y              extends Var {}

sig ITE extends IntNode {        sig GTE extends BoolNode {
  condition: BoolNode,             left, right: IntNode }
  then, elsen: IntNode }
```

Integer-typed nodes include variables and if-then-else expressions, while boolean-typed nodes include greater-than-or-equal expressions. Programs in this space evaluate to integers or booleans; integers are built into Alloy, but we must model boolean values explicitly.

```
abstract sig Bool{}   one sig BoolTrue, BoolFalse extends Bool{}
```

The standard evaluation semantics can be encoded in a predicate that constrains the evaluation relation. It works by constraining all compound syntax tree nodes based on the results of evaluating their children, but does not constrain the values of variables, allowing them to range over all values.

```
pred semantics[eval: Node -> (Int+Bool)] {
  all n: ITE | eval[n] in Int and
    eval[n.condition] = BoolTrue implies
      eval[n.then] = eval[n] else eval[n.elsen] = eval[n]
  all n: GTE | eval[n] in Bool and
    eval[n.left] >= eval[n.right] implies
      eval[n] = BoolTrue else eval[n] = BoolFalse
  all v: Var | one eval[v] and eval[v] in Int }
```

The specification says that the maximum of two numbers is equal to one of them, and greater than or equal to both.

```
pred spec[root: Node, eval: Node -> (Int+Bool)] {
  (eval[root] = eval[X] or eval[root] = eval[Y]) and
  (eval[root] >= eval[X] and eval[root] >= eval[Y]) }
```

Finally, the problem itself requires solving for some abstract syntax tree such that for all possible valuations for the variables, the specification holds.

```
pred synth[root: IntNode] {
  all eval: Node -> (Int+Bool) |                          (A.1)
    semantics[eval] implies spec[root, eval] }
run synth for 4
```

We present the results of our evaluation, including a performance comparison between Alloy* and existing program synthesizers, in Section VIII-B. A similar case study, where we use Alloy* to synthesize Margrave [15] security policies that satisfy given properties, can be found in our technical-report version of this paper [16, Sec. 2.2].

VII. Optimizations

Originally motivated by the formalization of the synthesis problem (as presented in Section VI), we designed and implemented two general purpose optimizations for Alloy*.

A. *Quantifier Domain Constraints*

As defined in Listing A.1, the `synth` predicate, although logically sound, suffers from serious performance issues. The main issue is the effect on the CEGIS loop of the implication inside the universal quantifier. To trivially satisfy the implication, the candidate search step can simply return an instance for which the semantics does not hold. Furthermore, adding the encoding of the counterexample refuting the previous instance is not going to constrain the next search step to find a program and a valuation for which the spec holds. This cycle can go on for unacceptably many iterations.

To overcome this problem, we can add syntax to identify the constraints that should be treated as part of the bounds of a quantification. The `synth` predicate, e.g., now becomes

```
pred synth[root: IntNode] {
  all eval: Node -> (Int + Bool) when semantics[eval] |
    spec[root, eval] }
```

The existing first-order semantics of Alloy is unaffected, i.e.,

```
all  x when D[x] | P[x]  ⟺  all  x | D[x] implies P[x]        (A.2)
some x when D[x] | P[x]  ⟺  some x | D[x] and P[x]
```

The rule for pushing negation through quantifiers (used by the converter to NNF) becomes:

```
not (all  x when D[x] | P[x])  ⟺  some x when D[x] | not P[x]
not (some x when D[x] | P[x])  ⟺  all  x when D[x] | not P[x]
```

(which is consistent with classical logic).

The formalization of the Alloy* semantics needs only a minimal change. The change in semantics is caused by essentially **not** changing (syntactically) how the existential counterpart of a universal quantifier is obtained— only by flipping the quantifier, and keeping the domain and the body the same (line 5, Figure 5). Consequently, the candidate condition always searches for an instance satisfying both the domain and the body constraint (i.e., both the semantics and the spec). The same is automatically true for counterexamples obtained in the verification step. The only actual change to be made to the formalization is expanding $q.body$ in line 38 according to the rules in Listing A.2.

Going back to the synthesis example, even after rewriting the `synth` predicate, unnecessary overhead is still incurred by quantifying over valuations for all the nodes, instead of valuations for just the input variables. Consequently, the counterexamples produced in the CEGIS loop do not guide the search as effectively. This observation leads us to our final formulation of the `synth` predicate:

```
pred synth[root: IntNode] {
  all env: Var -> Int | some eval: Node -> (Int+Bool)          (A.3)
    when env in eval && semantics[eval] | spec[root, eval] }
```

Despite using nested higher-order quantifiers, it is the most efficient: the inner quantifier (over `eval`) always takes exactly one iteration (to either prove or disprove the current `env`), because for a fixed `env`, `eval` is uniquely determined.

B. *Strictly First-Order Increments*

We already pointed out the importance of implementing the induction step (line 39, Figure 6) using incremental SAT solving. A problem, however, arises when the encoding of the counterexample (as defined in lines 38-40) is not a first-order formula—since not directly translatable to SAT, it cannot be incrementally added to the existing SAT translation of the candidate search condition (p_{cand}). In such cases, the semantics in Figure 6 demands that the conjunction of p_{cand} and $\mathcal{T}[\![f_{cex}]\!]$ be solved from scratch, losing any benefits from previously learned SAT clauses.

This problem occurs in our final formulation of the `synth` predicate (Listing A.3), due to the nested higher-order quantifiers. To address this issue, we relax the semantics of the induction step by replacing $\mathcal{S}[\![p_{cand} \wedge \mathcal{T}[\![f_{cex}]\!]]\!]$ (line 39) with

```
fun $\mathcal{T}_{\mathtt{fo}}(f)$ = match $p = \mathcal{T}[\![f]\!]$ with
  | FOL  → $p$
  | OR   → $reduce \vee,\ map(\mathcal{T}_{\mathtt{fo}},\ p.disjs)$
  | ∃∀   → $fold \wedge,\ p.conj,\ map(\mathcal{T}_{\mathtt{fo}},\ p.qps.pExists)$
$\mathcal{S}[\![p_{cand} \wedge \mathcal{T}_{\mathtt{fo}}(f_{cex})]\!]$
```

The $\mathcal{T}_{\mathtt{fo}}$ function ensures that f_{cex} is translated to a first-order `Proc`, which can always be added as an increment to the current SAT translation of the candidate condition. The trade-off involved here is that this new encoding of the counterexample is potentially not as strong, and therefore may lead to more CEGIS iterations before a resolution is reached. For that reason, Alloy* accepts a configuration parameter (accessible via the "Options" menu), offering both strategies. In Section VIII we provide experimental data showing that for all of our synthesis examples, the strictly first-order increments yielded better performance.

VIII. Evaluation

A. *Micro Benchmarks*

To assess scalability, we measure the time Alloy* takes to solve 4 classical, higher-order graph problems for graphs of varying size: `max clique`, `max cut`, `max independent set`, and `min vertex cover`. We used the Erdős-Rényi model [17] to randomly generate graphs to serve as inputs to the benchmark problems. We generated graphs of sizes ranging from 2 to 50. To cover a wide range of edge densities, for each size we generated 5 graphs, using different probabilities of inserting an edge between a pair of nodes: 0.1, 0.3, 0.5, 0.7, 0.9. We specified [16, Fig. 9] the graph problems in Alloy and used Alloy* to solve them. To ensure correctness of the results returned by Alloy*, we made sure they matched those of known imperative algorithms for the same problems[1]. The timeout for each Alloy* run was set to 100 seconds.

Figure 7 plots two graphs: (a) the average solving time across graphs size, and (b) the average number of explored candidates per problem. More detailed results, including individual times for each of the 5 densities, can be found in our technical-report version of this paper [16].

[1] For `max clique` and `max independent set`, we used the Bron-Kerbosch heuristic algorithm; for the other two, no good heuristic algorithm is known, and so we implemented enumerative search. In both cases, we used Java.

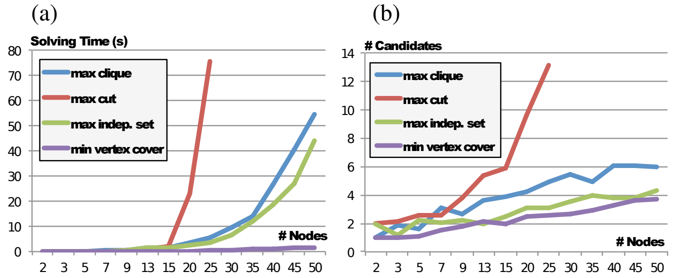

Figure 7. Average (over 5 different edge densities) (a) solving times, and (b) number of explored candidates for the graph algorithms.

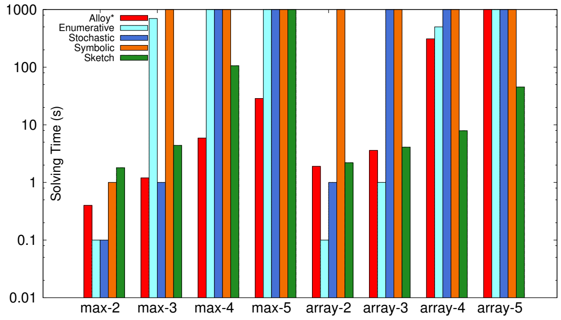

Figure 8. Comparison between Alloy* and Reference Solvers.

The performance results show that for all problems but `max cut`, Alloy* was able to handle graphs of sizes up to 50 nodes in less than a minute (`max cut` started to time out at around 25 nodes). Our original goal for these benchmarks was to be able to solve graphs with 10-15 nodes, and claim that Alloy* can be effectively used for teaching, specification animation, and small scope bounded verification, all within the Alloy Analyzer IDE (which is one of the most common uses of the Alloy technology). These results, however, suggest that executing higher-order specifications may be feasible even for declarative programming (where a constraint solver is embedded in a programming language, e.g., [10], [18], [19]), which is very encouraging.

The average number of explored candidates (Figure 7(b)) confirms the effectiveness of the CEGIS induction step at pruning the remainder of the search space. Even for graphs of size 50, in most cases the average number of explored candidates is around 6; the exception is, again, `max cut`, where this curve is closer to being linear. We further analyze how the total time is split over individual candidates in Section VIII-D.

B. Program Synthesis

To demonstrate the expressive power of Alloy*, we encoded 123 out of 173 benchmarks available in the SyGuS 2014 GitHub repository [20]—all except those from the "icfp-problems" folder. We skipped the 50 "icfp" benchmarks because they all use large (64-bit) bit vectors, which are not supported by Alloy; none of them could be solved anyway by any of the solvers that entered the SyGuS 2014 competition. All of our encoded benchmarks come with the official Alloy* distribution [5] and are accessible from the main menu: `File→Open Sample Models`, then choose `hol/sygus`.

Some SyGuS benchmarks require synthesizing multiple functions at once, and some require multiple applications of the synthesized functions. All of these cases can be handled with small modifications to our encoding presented in Section VI. For example, to synthesize multiple functions at once, we add additional root node pointers to the signature of the `synth` predicate; to allow for multiple applications of the same function we modify the `synth` predicate to compute multiple `eval` relations (one for each application). Finally, to support SyGuS benchmarks involving bit vectors, we exposed the existing Kodkod bitwise operators over integers at the Alloy level, and also made small changes to the Alloy grammar to allow for a more flexible integer scope specification (so that integer atoms can be specified independently of integer bitwidth).

To evaluate Alloy*'s performance, we ran the same benchmarks on the same computer using the three reference solvers. We limit the benchmarks to those found in the "integer-benchmarks" folder because they: (1) do not use bit vectors (which Alloy does not support natively), and (2) allow for the scope to be increased arbitrarily, and thus are suitable for performance testing. Our test machine had an Intel dual-core CPU, 4GB of RAM, and ran Ubuntu GNU/Linux and Sun Java 1.6. We set Alloy*'s solver to be MiniSAT.

Figure 8 compares the performance of Alloy* against the three SyGuS reference solvers and Sketch [1], a highly-optimized, state-of-the-art program synthesizer. According to these results, Alloy* scales better than the three reference solvers, and is even competitive with Sketch. On the `array-search` benchmarks, Sketch outperforms Alloy* for larger problem sizes, but on the `max` benchmarks, the opposite is true. Both solvers scale far more predictably than the reference solvers, but Alloy* has the additional advantage, due to its generality, of a flexible encoding of the target language's semantics, while Sketch relies on the semantics of the benchmark problems being the same as its own.

Other researchers have reported [21] that benefits can be gained by specifying tighter bounds on the abstract syntax tree nodes considered by the solver. Table I confirms that for `max` and `array` significant gains can be realized by tightening the bounds. In the case of `max`, tighter bounds allow Alloy* to improve from solving the 5-argument version of the problem to solving the 7-argument version. For these experiments, Scope 1 specifies the exact number of each AST node required; Scope 2 specifies exactly which types of AST nodes are necessary; and Scope 3 specifies only how many total nodes are needed. Other solvers also ask the user to bound the analysis—Sketch, for example, requires both an integer and recursion depth bound—but do not provide the same fine-grained control over the bounds as Alloy*. For the comparison results in Figure 8, we set the most permissive scope (Scope 3) for Alloy*.

These results show that Alloy*, in certain cases, not only scales better than the reference solvers, but can also be competitive with state-of-the-art solvers based on years of optimization. Such cases are typically those that require a

Table I
PERFORMANCE ON SYNTHESIS BENCHMARKS.

Problem	Scope 1		Scope 2		Scope 3	
	Steps	Time (s)	Steps	Time (s)	Steps	Time (s)
max-2	3	0.3	3	0.4	3	0.4
max-3	6	0.9	7	0.9	8	1.2
max-4	8	1.5	8	3.0	15	5.9
max-5	25	4.2	23	36.3	19	28.6
max-6	29	16.3	n/a	t/o	n/a	t/o
max-7	34	256.5	n/a	t/o	n/a	t/o
array-2	8	1.6	8	2.4	8	1.9
array-3	13	4.0	9	8.1	7	3.6
array-4	15	16.1	11	98.0	15	310.5
array-5	19	386.9	n/a	t/o	n/a	t/o

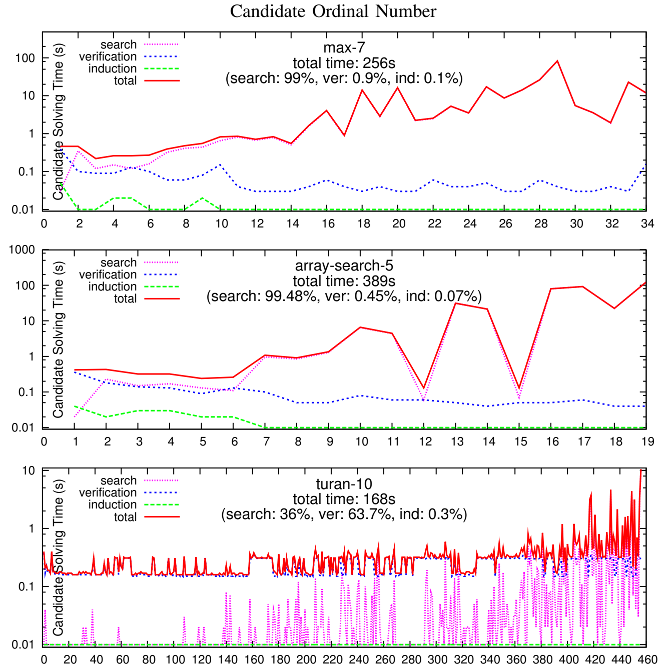

Figure 9. Distribution of total solving time over individual (sequentially explored) candidates for the three hardest benchmarks. Each candidate time is further split into times for each CEGIS phase.

structurally complex program AST (adhering to complex relational invariants) be discovered from a large search space. When the size of the synthesized program is small, however, the results of the SyGuS 2014 competition show [22] that non-constraint based techniques, such as enumerative and stochastic search, tend to be more efficient.

Finally, Alloy* requires only the simple model presented here—which is easier to produce than even the most naive purpose-built solver. Due to its generality, Alloy* is also, in some respects, a more flexible program synthesis tool—it makes it easy, for example, to experiment with the semantics of the target language, while solvers like Sketch have their semantics hard-coded.

C. Benefits of the Alloy* Optimizations

We used the program synthesis benchmarks (with the tightest, best performing scope), and the bounded verification of Turán's theorem from Section II to evaluate the optimizations introduced in Section VII by running the benchmarks with and without them. The baseline was a specification written without using domain constraints and an analysis without first-order increments. The next two rows correspond to (1) adding exactly one optimization (rewriting the specification to use the domain constraints for the synthesis benchmarks, and using first-order increments for Turán's theorem[2], and (2) adding both optimizations. Table II shows the results. Across the board, writing domain constraints makes a huge difference; using first-order increments often decreases solving time significantly, and, in the synthesis cases, causes the solver to scale to slightly larger scopes.

D. Distribution of Solving Time over Individual Candidates

In Figure 9 we take the three hardest benchmarks (`max-7`, `array-search-5`, and `turan-10`, with tightest bounds and both optimizations applied) and show how the total solving time is distributed over individual candidates (which are sequentially explored by Alloy*). Furthermore, for each candidate we show the percentage of time that went into each of the three CEGIS phases (search, verification, induction). Instead of trying to draw strong conclusions, the main idea behind this experiment is to illustrate the spectrum of possible behaviors the Alloy* solving strategy may exhibit at runtime.

The problems seen early in the search tend to be easy in all cases. The results reveal that later on, however, at least three different scenarios can happen. In the case of `max-7`, the SAT solver is faced with a number of approximately equally hard problems, which is where the majority of time is spent. In `array-search-5`, in contrast, saturation is reached very quickly, and the most time is spent solving very few hard problems. For `turan-10`, the opposite is true: the time spent solving a large number of very easy problems dominates the total time.

IX. RELATED WORK

Constraint solvers SMT solvers, by definition, find satisfying interpretations of first-order formulas over unbounded domains. In that context, only quantifier-free fragments are decidable. Despite that, many solvers (e.g., Z3 [23]) support certain forms of quantification by implementing an efficient matching heuristic based on patterns provided by the user [24]. Certain non-standard extension allow quantification over functions and relations for the purpose of checking properties over recursive predicates [25], as well as model-based quantifier instantiation [26]. In the general case, however, this approach often leads to "unknown" being return as the result. Tools that build on top of SMT raise the level of abstraction of the input language and provide quantification patterns that work more reliably in practice (e.g., [27], [28]), but are limited to first-order forms.

SAT solvers, on the other hand, are designed to work with

[2] If we first rewrote Turán's theorem to use domain constraints, there would be no nested CEGIS loops left, so increments would be first-order even without the other optimization.

Table II
PERFORMANCE OF ALLOY* (IN SECONDS) WITH AND WITHOUT OPTIMIZATIONS.

	max2	max3	max4	max5	max6	max7	max8	arr2	arr3	arr4	arr5	tur5	tur6	tur7	tur8	tur9	tur10
base	0.4	7.6	t/o	t/o	t/o	t/o	t/o	140	t/o	t/o	t/o	3.5	12.8	235	t/o	t/o	t/o
base + 1 optimization	0.4	1.0	4.7	10.3	136.4	t/o	t/o	2.9	6.3	76.9	t/o	1.1	5.1	43	t/o	t/o	t/o
base + both	0.3	0.9	1.5	4.2	16.3	163.6	987.3	1.6	4.0	16.1	485.6	0.5	2.1	3.8	15	45	168

bounded domains. Tools built on top may support logics richer than propositional formulas, including higher-order quantifiers. One such tool is Kodkod [14]. At the language level, it allows quantification over arbitrary relations, but the analysis engine, however, is not capable of handling those that are higher-order. Rosette [19] builds on top of Kodkod a suite of tools for embedding constraint solvers into programs for a variety of purposes, including synthesis. It implements a synthesis algorithm internally, so at the user level, unlike Alloy*, this approach enables only one predetermined form of synthesis, namely, finding an instantiation of a user-provided grammar that satisfies a specified property.

Synthesizers State-of-the-art synthesizers today are mainly purpose-built. Domains of application include program synthesis (e.g., [1], [29]–[32]), automatic grading of programming assignments [33], synthesis of data manipulation regular expressions [34], and so on, all using different ways for the user to specify the property to be satisfied. Each such specialized synthesizer, however, would be hard to apply in a domain different than its own. A recent effort has been made to establish a standardized format for program synthesis problems [4]; this format is syntax-guided, similar to that of Rosette, and thus less general than the language (arbitrary predicate logic over relations) offered by Alloy*.

Program Verifiers Program verifiers benefit directly from expressive specification languages equipped with more powerful analysis tools. In recent years, many efforts have been made towards automatically verifying programs in higher-order languages. Liquid types [35] and HMC [36] respectively adapt known techniques for type inference and abstract interpretation for this task. Bjørner et al. examine direct encodings into Horn clauses, concluding that current SMT solvers are effective at solving clauses over integers, reals, and arrays, but not necessarily over algebraic datatypes. Dafny [28] is the first SMT-based verifier to provide language-level mechanisms specifically for automating proofs by co-induction [37].

Executable Specifications Many research projects explore the idea of extending a programming language with symbolic constraint-solving features (e.g., [10], [11], [19], [38], [39]). Limited by the underlying constraint solvers, none of these tools can execute a higher-order constraint. In contrast, we used αRby [18] (our most recent take on this idea where we embed the entire Alloy language directly into Ruby), equipped with Alloy* as its engine, to run all our graph experiments (where αRby automatically translated input partial instances from concrete graphs, as well as solutions returned from Alloy back to Ruby objects), demonstrating how a higher-order constraint solver can be practical in this area.

Existing Alloy Tools Certain tools built using Alloy already provide means for achieving tasks similar to those we used as Alloy* examples. Aluminum [40], for instance, extends the Alloy Analyzer with a facility for minimizing solutions. It does so by using the low-level Kodkod API to selectively remove tuples from the resulting tuple set. To achieve minimality, by contrast, we used a purely declarative constraint to assert that there is no other satisfying solution with fewer tuples. In another instance, Montaghami et al. recognize a particular Alloy idiom involving a universal quantifier, and change the semantics to force all atoms in the domain of that quantifier to be always included in the analysis [41], effectively supporting only a small class of higher-order models.

Rayside et al. used the Alloy Analyzer to synthesize iterators from abstraction functions [42], as well as complex (non-pure) AVL tree operations from abstract specifications [43]. In both cases, they target very specific categories of programs, and their approach is based on insights that hold only for those particular categories.

X. CONCLUSION

Software analysis and synthesis tools have typically progressed by the discovery of new algorithmic methods in specialized contexts, and then their subsequent generalization as solutions to more abstract mathematical problems. This trend—evident in the history of dataflow analysis, symbolic evaluation, abstract interpretation, model checking, and constraint solving—brings many benefits. First, the translation of a class of problems into a single, abstract and general formulation allows researchers to focus more sharply, resulting in deeper understanding, cleaner APIs and more efficient algorithms. Second, generalization across multiple domains allows insights to be exploited more widely, and reduces the cost of tool infrastructure through sharing of complex analytic components. And third, the identification of a new, reusable tool encourages discovery of new applications.

In this paper, we have argued that the time is ripe to view higher-order constraint solving in this context, and we have proposed a generalization of a variety of algorithms that we believe suggests that the productive path taken by first-order solving might be taken by higher-order solving too.

ACKNOWLEDGMENT

This material is based upon work partially supported by the National Science Foundation under Grants No. CCF-1138967, CRI-0707612, and CCF-1438982. We also thank the anonymous reviewers for their thoughtful comments on the drafts of this paper.

REFERENCES

[1] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 404–415.

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

[3] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 632–647.

[4] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*. IEEE, 2013, pp. 1–17.

[5] "Alloy* Home Page," http://alloy.mit.edu/alloy/hola.

[6] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias, "Analysis of invariants for efficient bounded verification," in *ISSTA*. ACM, 2010, pp. 25–36.

[7] G. Dennis, "A relational framework for bounded program verification," Ph.D. dissertation, MIT, 2009.

[8] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Automated Software Engineering, 2001*. IEEE, 2001, pp. 22–31.

[9] N. Rosner, J. Galeotti, S. Bermúdez, G. M. Blas, S. P. De Rosso, L. Pizzagalli, L. Zemín, and M. F. Frias, "Parallel bounded analysis in code with rich invariants by refinement of field bounds," in *ISSTA*. ACM, 2013, pp. 23–33.

[10] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson, "Unifying execution of imperative and declarative code," in *ICSE*, 2011, pp. 511–520.

[11] H. Samimi, E. D. Aung, and T. D. Millstein, "Falling back on executable specifications," in *ECOOP*, 2010, pp. 552–576.

[12] J. F. Ferreira, A. Mendes, A. Cunha, C. Baquero, P. Silva, L. S. Barbosa, and J. N. Oliveira, "Logic training through algorithmic problem solving," in *Tools for Teaching Logic*. Springer, 2011, pp. 62–69.

[13] M. Aigner and G. M. Ziegler, "Turán's graph theorem," in *Proofs from THE BOOK*. Springer, 2001, pp. 183–187.

[14] E. Torlak, "A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications," Ph.D. dissertation, MIT, 2008.

[15] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proceedings of the 27th ICSE*. ACM, 2005, pp. 196–205.

[16] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A Higher-Order Relational Constraint Solver," MIT-CSAIL-TR-2014-018, Massachusetts Institute of Technology, Tech. Rep., 2014. [Online]. Available: http://hdl.handle.net/1721.1/89157

[17] P. Erdos and A. Renyi, "On the evolution of random graphs," *Mathematical Institute of the Hungarian Academy of Sciences, 5: 17-61*, 1960.

[18] A. Milicevic, I. Efrati, and D. Jackson, "αRby—An Embedding of Alloy in Ruby," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 56–71.

[19] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 2013, pp. 135–152.

[20] "SyGuS github repository," https://github.com/rishabhs/sygus-comp14.git.

[21] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 215–224.

[22] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis competition report," 2014, http://sygus.seas.upenn.edu/files/sygus_extended.pdf.

[23] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS 2008*, ser. lncs, vol. 4963. Springer, 2008, pp. 337–340.

[24] L. De Moura and N. Bjørner, "Efficient e-matching for smt solvers," in *Automated Deduction–CADE-21*. Springer, 2007, pp. 183–198.

[25] N. Bjørner, K. McMillan, and A. Rybalchenko, "Program verification as satisfiability modulo theories," in *SMT Workshop at IJCAR*, vol. 20, 2012.

[26] Y. Ge and L. De Moura, "Complete instantiation for quantified formulas in satisfiabiliby modulo theories," in *Computer Aided Verification*. Springer, 2009, pp. 306–320.

[27] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO 2005*, ser. lncs, vol. 4111. Springer, 2006, pp. 364–387.

[28] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *LPAR-16*, ser. lncs, vol. 6355. Springer, 2010, pp. 348–370.

[29] R. Singh and A. Solar-Lezama, "Synthesizing data structure manipulations from storyboards," in *Proceedings of the Symposium on the Foundations of Software Engineering*, 2011, pp. 289–299.

[30] K. R. M. Leino and A. Milicevic, "Program extrapolation with Jennisys," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, 2012, pp. 411–430.

[31] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Comfusy: A tool for complete functional synthesis," in *CAV*, 2010, pp. 430–433.

[32] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster, "Path-based inductive synthesis for program inversion," in *PLDI 2011*. ACM, 2011, pp. 492–503.

[33] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th PLDI*. ACM, 2013, pp. 15–26.

[34] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Commun. ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[35] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 159–169.

[36] R. Jhala, R. Majumdar, and A. Rybalchenko, "Hmc: Verifying functional programs using abstract interpreters," in *Computer Aided Verification*. Springer, 2011, pp. 470–485.

[37] K. R. M. Leino and M. Moskal, "Co-induction simply: Automatic co-inductive proofs in a program verifier," Technical Report MSR-TR-2013-49, Microsoft Research, Tech. Rep., 2013.

[38] A. S. Köksal, V. Kuncak, and P. Suter, "Constraints as control," *ACM SIGPLAN Notices*, 2012.

[39] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," in *Proceedings of the Symposium on Principles of Programming Languages*, 2012, pp. 85–96.

[40] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Aluminum: principled scenario exploration through minimality," in *ICSE*. IEEE Press, 2013, pp. 232–241.

[41] V. Montaghami and D. Rayside, "Staged evaluation of partial instances in a relational model finder," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 318–323.

[42] D. Rayside, V. Montaghami, F. Leung, A. Yuen, K. Xu, and D. Jackson, "Synthesizing iterators from abstraction functions," in *Proceedings of the International Conference on Generative Programming and Component Engineering*, 2012, pp. 31–40.

[43] D. Kurilova and D. Rayside, "On the simplicity of synthesizing linked data structure operations," in *Proceedings of the 12th international conference on Generative programming: concepts & experiences*. ACM, 2013, pp. 155–158.