αRby—An Embedding of Alloy in Ruby

Aleksandar Milicevic, Ido Efrati, and Daniel Jackson

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

Abstract. We present αRby—an embedding of the Alloy language in Ruby—and demonstrate the benefits of having a declarative modeling language (backed by an automated solver) embedded in a traditional object-oriented imperative programming language. This approach aims to bring these two distinct paradigms (imperative and declarative) together in a novel way. We argue that having the other paradigm available within the same language is beneficial to both the modeling community of Alloy users and the object-oriented community of Ruby programmers. In this paper, we primarily focus on the benefits for the Alloy community, namely, how αRby provides elegant solutions to several well-known, outstanding problems: (1) mixed execution, (2) specifying partial instances, (3) staged model finding.

1 Introduction

A common approach in formal modeling and analysis is to use (1) a declarative language (based on formal logic) for writing *specifications*, and (2) an automated constraint solver for finding valid *models* of such specifications[1]. Such models are most often either examples (of states or execution traces), or counterexamples (of correctness claims).

In many practical applications, however, the desired analysis involves more than a single model finding step. At the very least, a tool must convert the generated model into a form suitable for showing to the user; in the case of Alloy [8], this includes projecting higher-arity relations so that the model can be visualized as a set of snapshots. In some cases, the analysis may involve repeating the model finding step, e.g., to find a minimal model by requesting a solution with fewer tuples [13].

To date, these additional analyses have been hard-coded in the analysis tool. The key advantage of this approach is that it gives complete freedom to the tool developer. The disadvantage is that almost no freedom is given to the modeler, who must make do with whatever additional processing the tool developer chose to provide.

This paper explores a different approach, in which, rather than embellishing the analysis in an ad hoc fashion in the implementation of the tool, the modeling language itself is extended so that the additional processing can be expressed directly by the end user. An imperative language seems most suitable for this purpose, and the challenge therefore is to find a coherent merging of two languages, one declarative and one imperative. We show how this has been achieved in the merging of Alloy and Ruby.

[1] Throughout this paper, we use the term 'model' in its mathematical sense, and never to mean the artifact being analyzed, for which we use the term 'specification' instead.

Y. Ait Ameur and K.-D. Schewe (Eds.): ABZ 2014, LNCS 8477, pp. 56–71, 2014.

This challenge poses two questions, one theoretical and one practical. Theoretically, a semantics is needed for the combination: what combinations are permitted, and what is their meaning? Practically, a straightforward way to implement the scheme is needed. In particular, can a tool be built without requiring a new parser and engine that must handle both languages simultaneously?

This project focuses on the combination of Alloy and Ruby. In some respects, these choices are significant. Alloy's essential structuring mechanism, the signature [7], allows a relational logic to be viewed in an object-oriented way (in just the same way that instance variables in an object-oriented language can be viewed as functions or relations from members of the class to members of the instance variable type). So Alloy is well suited to an interpretation scheme that maps it to object-oriented constructs. Ruby is a good choice because, in addition to being object-oriented and providing (like most recent scripting languages) a powerful reflection interface, it offers a syntax that is flexible enough to support an almost complete embedding of Alloy with very few syntactic modifications.

At the same time, the key ideas in this paper could be applied to other languages; there is no reason, in principle, that similar functionality might not be obtained by combining the declarative language B [1] with the programming language Racket, for example.

The contributions of this paper include: (1) an argument for a new kind of combination of a declarative and an imperative language, justified by a collection of examples of functionality implemented in a variety of tools, all of which are subsumed by this combination, becoming expressible by the end-user; (2) an embodiment of this combination in αRby, a deep embedding of Alloy in Ruby, along with a semantics, a discussion of design challenges, and an open-source implementation [3] for readers to experiment with; and (3) an illustration of the use of the new language in a collection of small but non-trivial examples (out of 23 examples available on GitHub [4], over 1400 lines of specification in total).

2 Motivations

Analysis of a declarative specification typically involves more than just model finding. In this section, we outline the often needed additional steps.

Preprocessing The specification or the analysis command may be updated based on user input. For example, in an analysis of Sudoku, the size of the board must be specified. In Alloy, this size would be given as part of the 'scope', which assigns an integer bound to each basic type. For Sudoku, we would like to ensure that the length of a side is a perfect square; this cannot be specified directly in Alloy.

Postprocessing Once a model has been obtained by model finding, some processing may be needed before it is presented to the user. A common application of model finding in automatic configuration is to cast the desired configuration constraints as the specification, then perform the configuration steps based on the returned solution.

Partial instances A partial instance is a partial solution that is known (given) upfront. In solving a Sudoku problem, for example, the model finder must be given not

only the rules of Sudoku but also the partially filled board. It is easy to encode a partial solution as a formula that is then just conjoined to the specification. But although this approach is conceptually simple, it is not efficient in practice, since the model finder must work to reconstruct from this formula information (namely the partial solution) that is already known, thus needlessly damaging performance.

Kodkod (the back-end solver for Alloy) explicitly supports partial instances: it allows the user to specify relation *bounds* in terms of tuples that must (*lower bound*) and may (*upper bound*) be included in the final value. Kodkod then uses the bounds to shrink the search space, often leading to significant speedups [18]. At the Alloy level, however, this feature is not directly available[2].

Staged model finding Some analyses involve repeated calls to the model finder. In the simplest case, the bounds on the analysis are iteratively increased (when no counterexample has been found, to see if one exists within a larger scope), or decreased (when a counterexample has already been found, to see if a smaller one exists). Sometimes this is used as a workaround when a desired property cannot be expressed directly because it would require higher order logic.

Mixed execution Model finding can be used as a step in a traditional program execution. In this case, declarative specifications are executed 'by magic', as if, in a conventional setting, the interpreter could execute a program assertion by making it true despite the lack of any explicit code to establish it [10,9]. Alternatively, flipping the precedence of the two paradigms, the interpreter can be viewed as a declarative model finder that uses imperative code to setup a declarative specification to be solved. In this paper, we are primarily concerned with the latter direction, which has not been studied in the literature as much.

The Alloy Analyzer—the official and the most commonly used IDE for Alloy—does not currently provide any scripting mechanisms around its core model finding engine. Instead, its Java API must be used to automate even the most trivial scripting tasks. Using the Java API, however, is inconvenient; the verbosity and inflexibility of the Java language leads to poor transparency between the API and the underlying Alloy specification, making even the simplest scripts tedious and cumbersome to write. As a result, the official API is rarely used in practice, and mostly by expert users and researchers building automated tools on top of Alloy. This is a shame, since a simple transparent scripting shell would be, in many respects, beneficial to the typical Alloy user—the user who prefers to stay in an environment conceptually similar to that of Alloy and not have to learn a second, foreign API.

This is exactly what αRby provides—an embedding of the Alloy language in Ruby. Thanks to Ruby's flexibility and a very liberal parser, the αRby language manages to offer a syntax remarkably similar to that of Alloy, while still being syntactically correct Ruby. To reduce the gap between the two paradigms further, instead of using a separate AST, αRby maps the core Alloy concepts are onto the corresponding core concepts in Ruby (e.g., sigs are classes, fields are instance variables, atoms are objects, functions and predicates are methods, most operators are the same in both languages). αRby

[2] The Alloy Analyzer recognizes certain idioms as partial instances; some extensions (discussed in Section 6) support explicit partial instance specification.

automatically interoperates with the Alloy back end, so all the solving and visualization features of the Alloy Analyzer can be easily invoked from within an αRby program. Finally, the full power of the Ruby language is at the user's disposal for other tasks unrelated to Alloy.

3 αRby for Alloy Users

A critical requirement for embedding a modeling language in a programming language is that the embedding should preserve enough of the syntax of the language for users to feel comfortable in the new setting. We first introduce a simple example to illustrate how αRby achieves this for Alloy. Next, we address the new features brought by αRby, highlighted in Section 2, which are the primary motivation for the embedding.

Consider using Alloy to specify directed graphs and the *Hamiltonian Path* algorithm. *Signatures* are used to represent unary sets: `Node`, `Edge`, and `Graph`. *Fields* are used to represent relations between the signatures: `val` mapping each `Node` to an integer value; `src` and `dst` mapping each `Edge` to the two nodes (source and destination) that it connects; and `nodes` and `edges` mapping each `Graph` to its sets of nodes and edges.

A standard Alloy model for this is shown in Fig. 1(b), lines 2–4; the same declarations are equivalently written in αRby as shown in Fig. 1(a), lines 2–4.

To specify a Hamiltonian path (that is, a path visiting every node in the graph exactly once), a predicate is defined; lines 6–12 in Figs. 1(b) and 1(a) show the Alloy and αRby syntax, with equivalent semantics. This predicate asserts that the result (`path`) is a sequence of nodes, with the property that it contains all the nodes in the graph, and that, for all but the last index `i` in that sequence, there is an edge in the graph connecting the nodes at positions `i` and `i+1`. A *run* command is defined for this predicate (line 18), which, when executed, returns a satisfying instance.

Just as a predicate can be run for examples, an assertion can be checked for counterexamples. Here we assert that starting from the first node in a Hamiltonian path and transitively following the edges in the graph reaches all other nodes in the graph (lines 13–17). We expect this check (line 19) to return no counterexample.

From the model specification in Fig. 1(a), αRby dynamically generates the class hierarchy in Fig. 1(c). The generated classes can be used to freely create and manipulate graph instances, independent of the Alloy model.

In Alloy, a command is executed by selecting it in the user interface. In αRby, execution is achieved by calling the `exe_cmd` method. Fig. 1(d) shows a sample program that calls these methods, which includes finding an arbitrary satisfying instance for the `hampath` predicate and checking that the `reach` assertion indeed cannot be refuted.

This short example is meant to include as many different language features as possible and illustrate how similar αRby is to Alloy, despite being embedded in Ruby. We discuss syntax in Section 5.1; a summary of main differences is given in Table 1.

4 Beyond Standard Analysis

Sudoku has become a popular benchmark for demonstrating constraint solvers. The solver is given a partially filled $n \times n$ grid (where n must be a square number, so that

(a) Graph specification in αRby

```
alloy :GraphModel do
 sig Node [val: (lone Int)]
 sig Edge [src, dst: (one Node)] {src != dst}
 sig Graph[nodes:(set Node), edges:(set Edge)]

 pred hampath[g: Graph, path: (seq Node)] {
  path[Int] == g.nodes and
  path.size == g.nodes.size and
  all(i: 0...path.size-1) |
   some(e: g.edges) {
    e.src == path[i] && e.dst == path[i+1] }
 }
 assertion reach {
  all(g: Graph, path: (seq Node)) |
   if hampath(g, path)
    g.nodes.in? path[0].*((~src).dst)
   end }
 run :hampath, 5, Graph=>exactly(1), Node=>3
 check :reach, 5, Graph=>exactly(1), Node=>3
end
```

(b) Equivalent Alloy specification

```
module GraphModel
sig Node {val: lone Int}
sig Edge {src, dst: one Node}{src != dst}
sig Graph{nodes: set Node, edges: set Edge}

pred hampath[g: Graph, path: seq Node] {
 path[Int] = g.nodes
 #path = #g.nodes
 all i: Int | i >= 0 && i < minus[#path,1] => {
  some e: g.edges |
   e.src = path[i] && e.dst = path[plus[i,1]] }
 }
 assert reach {
  all g: Graph, path: seq Node |
   hampath[g, path] =>
    g.nodes in path[0].*(~src.dst)
 }
run hampath for 5 but exactly 1 Graph, 3 Node
check reach for 5 but exactly 1 Graph, 3 Node
```

(c) Automatically generated Ruby classes

```
module GraphModel
 class Node;  attr_accessor :val end
 class Edge;  attr_accessor :src, :dst end
 class Graph; attr_accessor :nodes, :edges end

 def self.hampath(g, path) #same as above end
 def self.reach()          #same as above end
 def self.run_hampath() exe_cmd :hampath end
 def self.check_reach() exe_cmd :reach end
end
```

(d) Running `hampath`, checking `reach`

```
# find an instance satisfying the :hampath pred
sol = GraphModel.run_hampath
assert sol.satisfiable?
g, path = sol["$hampath_g"], sol["$hampath_path"]
puts g.nodes # => e.g., {<Node$0>, <Node$1>}
puts g.edges # => e.g., {<Node$1, Node$0>}
puts path    # => {<0, Node$1>, <1, Node$0>}
# check that the "reach" assertion holds
sol = GraphModel.check_reach
assert !sol.satisfiable?
```

Fig. 1. *Hamiltonian Path* example

the grid is perfectly divided into n times $\sqrt{n} \times \sqrt{n}$ sub-grids), and is required to fill the empty cells with integers from $\{1, \ldots, n\}$ so that all cells within a given row, column, and sub-grid have distinct values.

Implementing a Sudoku solver directly in Alloy poses a few problems. A practical one is that such an implementation cannot easily be used as a stand-alone application, e.g., to read a puzzle from some standard format and display the solution in a user-friendly grid. A more fundamental problem is the inability to express the information about the pre-filled cell values as a partial instance; instead, the given cell values have to be enforced with logical constraints, resulting in significant performance degradation [18]. The αRby solution in Fig. 2 addresses both of these issues: on the left is the formal αRby specification, and on the right is the Ruby code constructing bounds and invoking the solver for a concrete puzzle.

Mixed Execution The imperative statements (lines 2, 7, 8) used to dynamically produce a Sudoku specification for a given size would not be directly expressible in Alloy. A concrete Ruby variable `N` is declared to hold the size, and can be set by the user before the specification is symbolically evaluated. Another imperative statement calculates the square root of `N` (line 6); that value is later embedded in the symbolic expression specifying uniqueness within sub-grids (line 13). For illustration purposes, a lambda function is defined (line 7) and used to compute sub-grid ranges (line 14).

(a) Sudoku specification in αRby

```
1  alloy :SudokuModel do
2   SudokuModel::N = 9
3
4   sig Sudoku[grid: Int ** Int ** (lone Int)]
5
6   pred solved[s: Sudoku] {
7    m   = Integer(Math.sqrt(N))
8    rng = lambda{|i| m*i...m*(i+1)}
9
10   all(r: 0...N) {
11    s.grid[r][Int] == (1..N) and
12    s.grid[Int][r] == (1..N)
13   } and
14   all(c, r: 0...m) {
15    s.grid[rng[c]][rng[r]] == (1..N)
16   }
17  }
18 end
```

(b) Solving the specification for a partial instance

```
1  class SudokuModel::Sudoku
2   def pi
3    bnds = Arby::Ast::Bounds.new
4    inds = (0...N)**(0...N) - self.grid.project(0..1)
5    bnds[Sudoku]          = self
6    bnds.lo[Sudoku.grid] = self ** self.grid
7    bnds.hi[Sudoku.grid] = self ** inds ** (1..N)
8    bnds.bound_int(0..N)
9   end
10  def solve() SudokuModel.solve :solved, self.pi end
11  def display() puts grid end
12  def self.parse(s) Sudoku.new grid:
13    s.split(/;\s*/).map{|x| x.split(/,/).map(&:to_i)}
14  end
15 end
16 SudokuModel.N = 4
17 s = Sudoku.parse "0,0,1; 0,3,4; 3,1,1; 2,2,3"
18 s.solve(); s.display(); # => {<0,0,1>, <0,1,3>, ...}
```

Fig. 2. A declarative *Sudoku* solver using αRby with partial instances

Partial Instances Fig. 2(b) shows how the bounds are computed for a given Sudoku puzzle, using a Ruby function `pi` (for "partial instance"). Remember that bounds are just tuples (sequences of atoms) that a relation must or may include; since signature definitions in αRby are turned into regular Ruby classes, instances of those classes will be used as atoms. The `Sudoku` signature is bounded by a singleton set containing only the `self` Sudoku object (line 5). Tuples that must be included in the `grid` relation are the values currently present in the puzzle (line 6); additional tuples that may be included are values from `1` to `N` for the empty cells (line 7; empty cell indexes computed in line 4). We also bound the set of integers to be used by the solver; Alloy, in contrast, only allows a cruder bound, and would include all integers within a given bitwidth. Finally, a Sudoku instance can be parsed from a string, and the solver invoked to find a solution satisfying the `solved` predicate (lines 17–18). When a satisfying solution is found, if a partial instance was given, fields of all atoms included in that partial instance are automatically populated to reflect the solution (confirmed by the output of line 18). This particular feature makes for seamless integration of *executable specifications* into otherwise imperative programs, since there is no need for any manual back and forth conversion of data between the program and the solver.

Staged Model Finding Consider implementing a Sudoku puzzle generator. The goal is now to find a partial assignment of values to cells such that the generated puzzle has a unique solution. Furthermore, the generator must be able to produce various difficulty levels of the same puzzle by iteratively decrementing the number of filled cells (while maintaining the uniqueness property). This is a higher-order problem that cannot be solved in one step in Alloy. With αRby, however, it takes only the following 8 lines to achieve this with a simple search algorithm on top of the already implemented solver:

```
1 def dec(sudoku, order=Array(0...sudoku.grid.size).shuffle)
2   return nil if order.empty? # all possibilities exhausted
3   s_dec = Sudoku.new grid: sudoku.grid.delete_at(order.first) # delete a tuple at random position
4   sol   = s_dec.clone.solve() # clone so that "s_dec" doesn't get updated if a solution is found
5   (sol.satisfiable? && !sol.next.satisfiable?) ? s_dec : dec(sudoku, order[1..-1])
6 end
7 def min(sudoku) (s1 = dec(sudoku)) ? min(s1) : sudoku end
8 s = Sudoku.new; s.solve(); s = min(s); puts "local minimum found: #{s.grid.size}"
```

The strategy here is to generate a solved puzzle (line 8), and keep removing one tuple from its `grid` at a time until a local minimum is reached (line 7); the question is which one can be removed without violating the uniqueness property. The algorithm first generates a random permutation of all existing grid tuples (line 1) to determine the order of trials. It then creates a new Sudoku instance with the chosen tuple removed (line 3) and runs the solver to find a solution for it. It finally calls `next` on the obtained solution (line 5) to check if a different solution exists; if it does not, a decremented Sudoku is found, otherwise moves on to trying the rest of the tuples. On a commodity machine, on average it takes about 8 seconds to minimize a Sudoku of size 4 (generating 13 puzzles in total, number of filled cells ranging from 16 down to 4), and about 6 minutes to minimize a puzzle of size 9 (55 intermediate puzzles), number of filled cells ranging from 81 down to 27).

5 The αRby Language

αRby is implemented as a domain-specific language in Ruby, and is (in standard parlance) "deeply embedded". *Embedded* means that all syntactically correct αRby programs are syntactically correct Ruby programs; *deeply* means that αRby programs exist as an AST that can be analyzed, interpreted, and so on. Ruby's flexibility makes it possible to create embedded languages that look quite different from standard Ruby. αRby exploits this, imitating the syntax of Alloy as closely as possible. Certain differences are unavoidable, mostly because of Alloy's infix operators that cannot be defined in Ruby.

The key ideas behind our approach are: (1) mapping the core Alloy concepts directly to those of object-oriented programming (OOP), (2) implementing keywords as methods, and (3) allowing mixed (concrete and symbolic) execution in αRby programs.

Mapping Alloy to OOP is aligned with the general intuition, encouraged by Alloy's syntax, that signatures can be understood as classes, atoms as objects, fields as instance variables, and all function-like concepts (functions, predicates, facts, assertions, commands) as methods [2].

Implementing keywords as methods works because Ruby allows different formats for specifying method arguments. αRby defines many such methods (e.g., **`sig`**, **`fun`**, **`fact`**, etc.) that (1) mimic the Alloy syntax and (2) dynamically create the underlying Ruby class structure using the standard Ruby metaprogramming facilities. For an example of the syntax mimicry, compare Figs. 1(a) and 1(b); for an example of metaprogramming, see Fig. 1(c).

Note that the meta information that appears to be lost in Fig. 1(c) (for example, the types of fields) is actually preserved in separate *meta* objects and made available

via the *meta* methods added to each of the generated modules and classes (e.g., `Graph.meta.field("nodes").type`).

Mixed execution, implemented on top of the standard Ruby interpreter, translates αRby programs into symbolic Alloy models. Using the standard interpreter means adopting the Ruby semantics of name resolution and operator precedence (which is inconvenient when it conflicts with Alloy's); a compensation, however, is the benefit of being able to mix symbolic and concrete code. We override all the Ruby operators in our symbolic expression classes to match the semantics of Alloy, and using a couple of other tricks (Section 5.3), are able to keep both syntactic (Section 5.1) and semantic (Section 5.2) differences to a minimum.

5.1 Syntax

A grammar of αRby is given in Fig. 3 and examples of principal differences in Table 1. In a few cases (e.g., function return type, field declaration, etc.) Alloy syntax has to be slightly adjusted to respect the syntax of Ruby (e.g., by requiring different kind of brackets). More noticeable differences stem from the Alloy operators that are illegal or cannot be overridden in Ruby; as a replacement, either a method call (e.g., `size` for cardinality) or a different operator (e.g., `**` for cross product) is used.

The difference easiest to overlook is the equality sign: `==` versus `=`. Alloy has no assignment operator, so the single equals sign always denotes boolean equality; αRby, in contrast, supports both concrete and symbolic code, so we must differentiate between assignments and equality checks, just as Ruby does.

The tokens for the join and the two closure operators (`.`, `^` and `*`) exist in Ruby, but have fundamentally different meanings than in Alloy (object dereferencing and an infix binary operator in Ruby, as opposed to an infix binary and a prefix unary operator in Alloy). Despite this, αRby preserves Alloy syntax for many idiomatic expressions. Joins in Alloy are often applied between an expression and a field whose left-hand type matches the type of the expression (ie, in the form `e.f`, where `f` is a field from the type of `e`). This corresponds closely to object dereferencing, and is supported by αRby (e.g., `g.nodes` in Fig. 1(a)). In other kinds of joins, the right-hand side must be enclosed in parentheses. Closures are often preceded by a join in Alloy specifications. Those constructs yield *join closure* expressions of the form `x.*f`. In Ruby, this translates to calling the `*` method on object `x` passing `f` as an argument, so we simply override the `*` method to achieve the same semantics (e.g., line 15, Fig. 1(a)).

This grammar is, for several reasons, an under-approximation of programs accepted by αRby: (1) Ruby allows certain syntactic variations (e.g., omitting parenthesis in method calls, etc.), (2) αRby implements special cases to enable the exact Alloy syntax for certain idioms (which do not always generalize), and (3) αRby provides additional methods for writing expression that have more of a Ruby-style feel.

5.2 Semantics

This section formalizes the translation of αRby programs into Alloy. We provide semantic functions (summarized in Fig. 5) that translate the syntactic constructs of Fig. 3

```
spec      ::= "alloy" cname "do" [open*] paragraph* "end"
open      ::= "open" cnameID
paragraph ::= factDecl   | funDecl | cmdDecl | sigDecl
sigQual   ::= "abstract" | "lone"  | "one"   | "some"  | "ordered"
sigDecl   ::= sigQual* "sig"  cname,+ ["extends" cnameID] ["[" rubyHash "]"] [block]
factDecl  ::= "fact"         [fname] block
funDecl   ::= "fun"           fname  "[" rubyHash "]"  "[" expr "]" block
            | "pred"          fname ["[" rubyHash "]"]              block
cmdDecl   ::= ("run"|"check") fname "," scope
            | ("run"|"check") "(" scope ")" block
expr      ::= ID | rubyInt | rubyBool | "(" expr ")"
            | unOp expr         | unMeth "(" expr ")"
            | expr binOp expr   | expr "[" expr "]"    | expr "if" expr
            | expr "." "(" expr ")"                  // relational join
            | expr "." (binMeth | ID) "(" expr,* ")"  // function/predicate call
            | "if" expr "then" expr ["else" expr] "end"
            | quant "(" rubyHash ")" block
quant     ::= "all" | "no" | "some" | "lone" | "one" | "sum" | "let" | "select"
binOp     ::= "||" | "or" | "&&" | "and" | "**" | "&" | "+" | "-" | "*" | "/" | "%"
            | "<<" | ">>" | "==" | "<=>" | "!=" | "<" | ">" | "<=" | ">="
binMeth   ::= "closure" | "rclosure" | "size" | "in?" | "shr" | "<" | ">" | "*" | "^"
unOp      ::= "!" | "~" | "not"
unMeth    ::= "no" | "some" | "lone" | "one" | "set" | "seq"
block     ::= "{" stmt* "}" | "do" stmt* "end"
stmt      ::= expr | rubyStmt
scope     ::= rubyInt "," rubyHash  // global scope, individual sig scopes
ID        ::= cnameID | fnameID
cname     ::= cnameID | '"'cnameID'"' | "'"cnameID"'" | ":"cnameID
fname     ::= fnameID | '"'fnameID'"' | "'"fnameID"'" | ":"fnameID
cnameID   ::= constant identifier in Ruby (starts with upper case)
fnameID   ::= function identifier in Ruby (starts with lower case)
```

Fig. 3. Core αRby syntax in BNF. Productions starting with: `ruby` are defined by Ruby.

Table 1. Examples of differences in syntax between αRby and Alloy

description	Alloy	αRby
equality	`x = y`	`x == y`

sigs and fields	`sig S {`
	`  f: lone S -> Int`
	`}`
		`sig S [`
		`  f: lone(S) ** Int`
		`]`

fun return type declaration	`fun f[s: S]: set S {}`	`fun f[s: S][set S] {}`
set comprehension	`{s: S | p1[s]}`	`S.select{|s| p1(s)}`

quantifiers	`all s: S {`
	`  p1[s]`
	`  p2[s]`
	`}`
		`all(s: S) {`
		`  p1(s) and`
		`  p2(s)`
		`}`

illegal Ruby operators	`x in y, x !in y`
	`x != y`
	`x -> y`
	`x . y`
	`#x`
	`x => y`
	`x => y else z`
	`S <: f, f >: Int`
		`x.in?(y), x.not_in?(y)`
		`not x > y`
		`x ** y`
		`x.(y)`
		`x.size`
		`y if x`
		`if x then y else z`
		`S.< f, f.> Int`

operator arity mismatch	`^x, *x`	`x.closure, x.rclosure`
fun/pred calls	`f1[x]`	`f1(x)`

```
Expr  = VarExpr(name: String, domain: Expr | Type)
      | IntExpr(value: Int) | BoolExpr(value: Bool)
      | UnExpr(sub: Expr) | BinExpr(lhs: Expr, rhs: Expr)
      | CallExpr(target: Expr, fun: FunDecl, args: Expr*)
      | QuantExpr(kind: String, vars: VarExpr*, body: Expr)
Decl  = Spec(name: String, opens: Spec*, sigs: SigDecl*, funs: FunDecl*)
      | SigDecl(name: String, parent: SigDecl, fields: VarExpr*, inv: FunDecl)
      | FunDecl(name: String, params: VarExpr*, ret: Expr, body: Expr)
Type  = Univ | None | Int | SigDecl | ProductType(lhs: Type, rhs: Type)
Store = {name: String; binding: Expr | Decl}
```

Fig. 4. Overview of the semantic domains. (`Expr` and `Decl` correspond directly to the Alloy AST)

$\mathcal{A}$:	`specification → Store → Spec`	$\mathcal{E}$:	`expr → Store → Expr`
ξ :	`sigDecl → Store → SigDecl`	β:	`block → Store → Expr`
ϕ :	`funDecl → Store → FunDecl`	δ:	`decl* → Store → (VarExpr*, Store)`

Fig. 5. Overview of the semantic functions which translate grammar rules to semantic domains

to Alloy AST elements defined in Fig. 4. A store, binding names to expressions or declarations, is maintained throughout, representing the current evaluation context.

Expressions The evaluation of the αRby expression production rules (`expr`) into Alloy expressions (`Expr`) is straightforward for the most part (Fig. 6). Most of the unary and binary operators have the same semantics as in Alloy; exceptions are $**$ and **`if`**, which translate to `->` and `=>` (lines 5–11). For the operators that do not exist in Ruby, an equivalent substitute method is used (lines 12–20). A slight variation of this approach is taken for the `^` and $*$ operators (lines 21–22), to implement the "join closure" idiom (explained in Section 5.1).

The most interesting part is the translation of previously undefined method calls (lines 23–27). We first apply the τ function to obtain the type of the left-hand side expression, and then the $\oplus$ function to extend the current store with that type (line 23). In a nutshell, this will create a new store with added bindings for all fields and functions defined for the range signature of that type (the $\oplus$ function is formally defined in Fig. 8 and discussed in more detail shortly). Afterward, we look up `meth` as an identifier in the new store (line 24) and, if an expression is found (line 25), the expression is interpreted as a join; if a function declaration is found (line 26), it is interpreted a function call; otherwise, it is an error.

For quantifiers (lines 29-30), quantification domains are evaluated in the context of the current store (using the δ helper function, defined in Fig. 8) and the body is evaluated (using the β function, defined in Fig. 7) in the context of the new store with added bindings for all the quantified variables (returned previously by δ).

Blocks The semantics of αRby blocks differs from Alloy's. An Alloy block (e.g., a quantifier body) containing a sequence of expressions is interpreted as a conjunction of all the constituent constraints (a feature based on Z [17]). In αRby, in contrast, such as sequence evaluates to the meaning of the last expression in the sequence. This was

$\mathcal{E}$: expr → Store → Expr

1. $\mathcal{E}[\![\texttt{ID}]\!]\sigma$ ≡ σ[ID]
2. $\mathcal{E}[\![\texttt{rubyInt}]\!]\sigma$ ≡ IntExpr(rubyInt)
3. $\mathcal{E}[\![\texttt{rubyBool}]\!]\sigma$ ≡ BoolExpr(rubyBool)
4. $\mathcal{E}[\![(e)]\!]\sigma$ ≡ $\mathcal{E}[\![e]\!]\sigma$
5. $\mathcal{E}[\![\texttt{unOp}\ e]\!]\sigma$ ≡ UnExpr(unOp, $\mathcal{E}[\![e]\!]\sigma$)
6. $\mathcal{E}[\![\texttt{unMeth}(e)]\!]\sigma$ ≡ UnExpr(unMeth, $\mathcal{E}[\![e]\!]\sigma$)
7. $\mathcal{E}[\![e_1\ \texttt{**}\ e_2]\!]\sigma$ ≡ BinExpr("->", $\mathcal{E}[\![e_1]\!]\sigma$, $\mathcal{E}[\![e_2]\!]\sigma$)
8. $\mathcal{E}[\![e_1\ \texttt{binOp}\ e_2]\!]\sigma$ ≡ BinExpr(binOp, $\mathcal{E}[\![e_1]\!]\sigma$, $\mathcal{E}[\![e_2]\!]\sigma$)
9. $\mathcal{E}[\![e_1[e_2]]\!]\sigma$ ≡ BinExpr("[]", $\mathcal{E}[\![e_1]\!]\sigma$, $\mathcal{E}[\![e_2]\!]\sigma$)
10. $\mathcal{E}[\![e_1\ \textbf{if}\ e_2]\!]\sigma$ ≡ BinExpr("=>", $\mathcal{E}[\![e_2]\!]\sigma$, $\mathcal{E}[\![e_1]\!]\sigma$)
11. $\mathcal{E}[\![e_1.(e_2)]\!]\sigma$ ≡ BinExpr(".", $\mathcal{E}[\![e_1]\!]\sigma$, $\mathcal{E}[\![e_2]\!]\sigma$)
12. $\mathcal{E}[\![e.\texttt{binMeth}]\!]\sigma$ ≡ **match** binMeth **with**
13. | closure → UnExpr("^", $\mathcal{E}[\![e]\!]\sigma$)
14. | rclosure → UnExpr("*", $\mathcal{E}[\![e]\!]\sigma$)
15. | size → UnExpr("#", $\mathcal{E}[\![e]\!]\sigma$)
16. $\mathcal{E}[\![e.\texttt{binMeth}(a_1)]\!]\sigma$ ≡ **match** binMeth **with**
17. | in? → BinExpr("in", $\mathcal{E}[\![e]\!]\sigma$, $\mathcal{E}[\![a_1]\!]\sigma$)
18. | shr → BinExpr(">>>", $\mathcal{E}[\![e]\!]\sigma$, $\mathcal{E}[\![a_1]\!]\sigma$)
19. | < → BinExpr("<:", $\mathcal{E}[\![e]\!]\sigma$, $\mathcal{E}[\![a_1]\!]\sigma$)
20. | > → BinExpr(":>", $\mathcal{E}[\![e]\!]\sigma$, $\mathcal{E}[\![a_1]\!]\sigma$)
21. | ^ → $\mathcal{E}[\![e.(a_1.\text{closure})]\!]\sigma$
22. | * → $\mathcal{E}[\![e.(a_1.\text{rclosure})]\!]\sigma$
23. $\mathcal{E}[\![e.\texttt{ID}(a_1, \ldots)]\!]\sigma$ ≡ **let** σ_{sub} = $\sigma \oplus \tau(e)$ **in**
24. **match** σ_{sub}[ID] **as** x **with**
25. | Expr → BinExpr(".", $\mathcal{E}[\![e]\!]\sigma$, x)
26. | FunDecl → CallExpr($\mathcal{E}[\![e]\!]\sigma$, x, $\mathcal{E}[\![a_1]\!]\sigma$, ...)
27. | → $fail$
28. $\mathcal{E}[\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \textbf{end}]\!]\sigma$ ≡ $\mathcal{E}[\![(e_2\ \textbf{if}\ e_1)\ \textbf{and}\ (e_3\ \textbf{if}\ !e_1)]\!]\sigma$
29. $\mathcal{E}[\![\texttt{quant}(d_*)\ \texttt{block}]\!]\sigma$ ≡ **let** v_*, σ_b = $\delta(d_*)\sigma$ **in**
30. QuantExpr(quant, v_*, $\beta[\![\texttt{block}]\!]\sigma_b$)

Fig. 6. Evaluation of αRby expressions (expr production rules) into Alloy expressions (Expr)

a design decision, necessary to support mixed execution (as in Fig. 2(a), lines 7–16). Since Ruby is not a pure functional language, previous statements can affect the result of the last statement by mutating the store, which effectively gives us the opportunity to easily mix concrete and symbolic execution.

This behavior is formally captured in the β function (Fig. 7). Statements (s_1, ..., s_n) are evaluated in order (line 32). If a statement corresponds to one of the expression rules from the αRby grammar (line 34), it is evaluated using the previously defined $\mathcal{E}$ function; otherwise (line 35), it is interpreted by Ruby (abstracted as a call to the $\mathcal{R}$ functions). Statements interpreted by Ruby may change the store, which is then passed on to the subsequent statements.

Function Declarations The evaluation function (ϕ, Fig. 7, lines 37–38) is similar to quantifier evaluation, except that the return type is different. The semantics of other function-like constructs (predicates, facts, etc.) is analogous, and is omitted for brevity.

```
β : block → Store → Expr
31. β⟦do s1; ...; sn end⟧σ ≡ β⟦{ s1; ...; sn }⟧σ ≡ σcurr = σ, res = nil
32.                                              for si: {s1, ..., sn} do
33.                                                match si with
34.                                                | expr →   res ←E⟦si⟧σcurr
35.                                                |      →   res, σcurr ←R(si)σcurr
36.                                              return res

φ : funDecl → Store → FunDecl
37. φ⟦fun fname[d*][eret] block⟧σ ≡ let v*, σb = δ(d*)σ in
38.                                   FunDecl(fname, v*, E⟦eret⟧σ, β⟦block⟧σb)

ξ : sigDecl → Store → SigDecl
39. ξ⟦sig cname extends sup [d*] block⟧σ ≡
40.   let sp       = τ(σ[sup]) in
41.   let fld*, _  = δ(d*)σ in
42.   let this     = VarExpr("this", cname) in
43.   let tfld*    = map(λfi · BinExpr(".", this, fi), fld*) in
44.   let σs       = σ⊕SigDecl(cname, sp, tfld*, BoolExpr(true)) in
45.     SigDecl(cname, sp, fld*, β⟦block⟧σs["this" ↦ this])

A : spec → Store → Spec
46. A⟦alloy cname do open* paragraph* end⟧σ ≡
47.   let opn*    = map(λcnameID · σ[cnameID], open*) in
48.   let sig*    = map(ξ, filter(sigDecl, paragraph*)) in
49.   let fun*    = map(φ, filter(funDecl, paragraph*)) in
50.   let a       = Spec(cname, opn*, sig*, fun*) in
51.     if resolved(a) then a
52.     elsif σ⊕a ≠ σ then A⟦alloy cname do open* paragraph* end⟧σ⊕a else fail
```

Fig. 7. Evaluation of blocks and all declarations

Signature Declarations The evaluation function (function ξ, Fig. 7, lines 39–45) is conceptually straightforward: as before, functions δ and β can be reused to evaluate the field name-domain declarations and the appended facts block, respectively. The caveat is that appended facts in Alloy must be evaluated in the context of Alloy's *implicit this*, meaning that the fields from the parent signature should be implicitly joined on the left with an implicit `this` keyword. To achieve this, we create a variable corresponding to `this` and a new list of fields with altered domains (lines 42–43). A temporary `SigDecl` containing those fields is then used to extend the current store (line 44). A binding for `this` is also added and the final store is used to evaluate the body (line 45). The temporary signature is created just for the convenience of reusing the semantics of the $\oplus$ operator (explained shortly).

Top-Level Specifications Evaluation of an αRby specification (function $\mathcal{A}$, Fig. 7, lines 46–52) uses the previously defined semantic functions to evaluate the nested signatures and functions. Since declaration order does not matter in Alloy, multiple passes may be needed until everything is resolved or a fixed point is reached (lines 51–52).

Name-Domain Declaration Lists Name-domain lists are used in several places (for fields, method parameters, and quantification variables); common functionality is

δ : decl* → Store → (VarExpr*, Store)

53. $\delta(v_1 : e_1, \ldots, v_n : e_n)\sigma \equiv$ **let** $vars = \bigcup_{1<=i<=n}$ VarExpr(v_i, $\mathcal{E}[\![e_i]\!]\sigma$) **in**
54. $[vars, \sigma \oplus vars]$

$\oplus$: Store → *Any* → Store

55. $\sigma \oplus x \equiv$ **match** x **with**
56. | VarExpr(n,_) | FunDecl(n,_) → $\sigma[n \mapsto x]$)
57. | VarExpr* | FunDecl* → $fold(\oplus, \sigma, x)$
58. | ProductType(_, rhs) → $\sigma \oplus rhs$
59. | SigDecl(n, s_p, fld_*,_) → **let** $\sigma_p = \sigma \oplus s_p$ **in**
60. **let** $\sigma_s = \sigma_p[n \mapsto$ VarExpr$(n, x)]$ **in**
61. $fold(\oplus, \sigma_s,$ funs$(x) + fld_*)$
62. | Spec(n, opn_*, sig_*, fun_*) → $fold(\oplus, \sigma, opn_* + fun_* + sig_*)$
63. | → σ

$\mathcal{R}$	: rubyStmt → Store → (*Object* → Store)	Executes arbitrary Ruby code
τ	: Expr → Type	Type of the given expression
funs	: SigDecl → FunDecl*	Functions where given sig is first param
resolved	: Spec → Bool	Whether all references are resolved

Fig. 8. Helper functions

extracted and defined in the δ function (Fig. 8). It simply maps the input list into a list of VarExpr expressions, each having $name$ the same as in the declaration list and $domain$ equal to the evaluation of the declared domain against the current store (line 53). It returns that list and the current store extended with those variables (line 54).

Store Extension The $\oplus$ operator (Fig. 8) is used to extend a store with one or more VarExpr or FunDecl, a Type, and a Spec. If a VarExpr or a FunDecl is given, its name is bound to itself. If a list is given, the operation is folded over the entire list. Extending with a Type reduces to extending with the range of that type. Extending with a SigDecl means recursively adding bindings for its parent signature, adding a binding for the name of that signature, bindings for all the functions that take that signature as the first argument (an auxiliary function funs(x) discovers such functions), and bindings for all its fields. Extending with a Spec adds bindings for all the sigs and functions defined in it, including those from all opened specifications.

5.3 Implementation Considerations

Symbolic Execution Using the standard Ruby interpreter to symbolically execute αRby programs relieves us from having to keep an explicit representation of the store; instead, the store is implicit in the states of the object in which the execution takes place. Having signatures, fields, and functions represented directly as classes, instance variables, and methods, means having most of the bindings (as defined in Section 5.2) already in place for all sigs and atoms; for all other expressions, missing methods are dynamically forwarded to the signature class corresponding to the expression's type.

One technical challenge is that the semantics of quantifiers requires a new scope to be created, which, for our syntax, Ruby does not already ensure. Consider the following αRby code: **all**(s: S){**some** s}. This is just a hash and a block passed to our

domain-specific-language method. When the block is eventually executed (to obtain the symbolic body for this universal quantifier), `s` must be available as a symbolic variable inside of that block. We do that by first dynamically defining a method with the same name in the context of that block, then calling the block, and, finally, redefining the same method to call `super`:

```
ctx = block.binding.eval("self")
ctx.define_singleton_method :s, lambda{VarExpr.new(:s, S)}
begin block.call ensure ctx.define_singleton_method(:s) do super() end end
```

Responding to Missing Methods and Constants To avoid requiring strings instead of identifiers for every new definition (e.g., **`sig`** `:Graph` instead of **`sig`** `Graph`, where `Graph` is previously undefined), αRby overrides `const_missing` and `method_missing` and instead of failing returns a `MissingBuilder` instance. Furthermore, `MissingBuilder` instances also accept a block at creation time, and respond to several operator methods, making constructs like **`fun`** `f[s: S][set S] {}` possible. To guard against unintended conversions (e.g., typos), αRby raises a syntax error every time a `MissingBuilder` is not "consumed" (by certain DSL methods, like **`sig`** and **`fun`**) by the end of its scope block.

Online Source Instrumentation For the purpose of symbolic evaluation, the source code of every αRby function/predicate is instrumented before it is turned into a Ruby method. The need for instrumentation arises because certain operators and control structures, which we would like to treat symbolically, cannot be overridden; examples include all the if-then-else variants, as well as the logic operators. Our instrumentation uses an off-the-self parser and implements a visitor over the generated AST to replace these constructs with appropriate αRby expressions (e.g., `x` **`if`** `y` gets translated to `BinExpr.new(IMPLIES, proc{y}, proc{x})`). This "traverse and replace" algorithm is far simpler than implementing a full parser for the entire Alloy grammar.

Distinguishing Equivalent Ruby Constructs Ruby allows different syntactic constructs for the same underlying operation. For example, some built-in infix operators can be written with or without a dot between the left-hand side and the operator (e.g., `a*b` is equivalent to `a.*b`). Since αRby already performs online source instrumentation, it additionally detects the following syntactic nuances for the purpose of assigning different semantics: (1) in Ruby, "`<b2>` **`if`** `<b1>`" is equivalent to "`b1` **`and`** `b2`", but our instrumenter always rewrites **`and`** and **`or`** to boolean conjunction and disjunction; (2) when prefixed with a dot, operators `*`, `<` and `>` are translated to join closure, domain restriction, and range restriction, respectively (`.*`, `<:`, and `:>` in Alloy).

αRby to Alloy Bridge All model-finding tasks are delegated to a slightly modified version of the official Alloy Analyzer Java implementation. The main modification we made was adding an extra API method, which additionally accepts a partial instance (represented in a simple textual format independent of the Alloy language). The Alloy Analyzer already has a complex heuristic for computing bounds from the scope specification and certain (automatically detected) idioms; we retain all those features, and on top of them use the αRby-provided partial instance to shrink the bounds further (formalization of which is beyond the scope of this paper). To interoperate between Ruby and Java, we use RJB [14], which conveniently automates most of the process.

6 Related Work

Montaghami and Rayside [11] extended the Alloy language with special syntax for specifying partial instances. They argue convincingly for the importance of having partial instances for Alloy, giving use cases such as test-driven model development, regression testing of models, modeling by example etc. They also provide experimental evidence that staged model finding can lead to better scalability. Their approach is limited to partial instances only, and it does not provide any scripting mechanisms for automating such tasks. Thus to carry out their staged model finding experiments, after obtaining an instance in the first stage, they manually inspected it (e.g., in the visualizer), rewrote it using the new syntax, and then solved in the second stage. Using αRby would automate the whole process, since an αRby instance can provide a set of exact bounds for all included relations, and can handle all the use cases discussed.

A number of tools built on top of Alloy have implemented (often in an ad hoc fashion) one or more features that can now be provided by αRby. Aluminum [13] implements an interesting heuristic for minimizing Alloy instances and by default showing the minimal one first. It also allows the user to augment the current instance by selecting one or more tuples to be included in the next instance. αRby provides a more generic mechanism that lets the user provide an arbitrary formula (possibly involving atoms from the current instance) to be satisfied in the next solution. TACO [5] is a bounded verifier for Java that achieves scalability by relying heavily on the Alloy Analyzer to recognize certain idioms as partial instances; we believe αRby would have made their implementation much simpler.

Our mixed execution was inspired by Rubicon's [12] symbolic evaluator, which also uses the standard Ruby interpreter. Unlike αRby, Rubicon stubs the library code with custom expressions in order to symbolically execute and verify existing web apps.

Many research projects explore the idea of extending a programming language with symbolic constraint-solving features (e.g., [15,10,9,20,19,16]). αRby can be understood as a kind of dual, with the opposite goal. While these efforts aim to bring declarative features in imperative programming, αRby aims to bring imperative features to declarative modeling. Although the basic idea of combining declarative model finding and imperative model finding is shared, the research challenges are very different. As this paper has explained, αRby addresses the challenge of embedding an entire modeling language in a programming language, whereas these related projects instead tend to use a constraint language that is only a modest extension of the programming language's existing expression sublanguage. αRby also addresses the challenge of reconciling two different views of a data structure: one as objects on a heap, and the other as relations (and in this respect is related to work on relational data representation, such as [6]).

7 Conclusion

On the one hand, αRby addresses a collection of very practical problems in the use of a model finding tool. This paper's contribution can thus be regarded as primarily architectural, in demonstrating a different way to build an analysis tool that uses a DSL embedding to allow end-user scripting, rather than a closed compiler-like tool that can be extended only by one of the tool's developers.

On the other hand, αRby suggests a new way to think about a modeling language. The constructs of the language are not treated as functions that generate abstract syntax trees only in a mathematical sense, but are implemented as these functions in a manner that the end user can exploit. This leads us to wonder whether it might be possible to use this style of embedding in the very design of the modeling language. Perhaps, had this approach been available when Alloy was designed, an essential core might have been more cleanly separated from a larger collection of structuring idioms, implemented as functions on top of the core's functions.

Practically speaking, we hope that the developers of tools that use Alloy as a backend will be able to use αRby in their implementations, at the very least making it easier to prototype new functionality. And perhaps the implementors of tools for other declarative languages will find ideas here that they can exploit in similar embeddings.

References

1. Abrial, J., Hoare, A.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
2. How to think about an Alloy model: 3 levels, `http://alloy.mit.edu/alloy/tutorials/online/sidenote-levels-of-understanding.html`
3. aRby—An Embedding of Alloy in Ruby, `https://github.com/sdg-mit/arby`
4. Online collection of aRby examples, `https://github.com/sdg-mit/arby/tree/master/lib/arby_models`
5. Galeotti, J.P., Rosner, N., López Pombo, C.G., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: ISSTA. ACM (2010)
6. Hawkins, P., Aiken, A., Fisher, K., Rinard, M., Sagiv, M.: Data representation synthesis. ACM SIGPLAN Notices 46 (2011)
7. Jackson, D.: Micromodels of software: Lightweight modelling and analysis with alloy (2002)
8. Jackson, D.: Software Abstractions: Logic, language, and analysis. MIT Press (2006)
9. Köksal, A.S., Kuncak, V., Suter, P.: Constraints as control. ACM SIGPLAN Notices (2012)
10. Milicevic, A., Rayside, D., Yessenov, K., Jackson, D.: Unifying execution of imperative and declarative code. In: ICSE (2011)
11. Montaghami, V., Rayside, D.: Extending alloy with partial instances. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 122–135. Springer, Heidelberg (2012)
12. Near, J.P., Jackson, D.: Rubicon: bounded verification of web applications. In: FSE. ACM (2012)
13. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: ICSE. IEEE Press (2013)
14. Ruby Java Bridge, `http://rjb.rubyforge.org/`
15. Samimi, H., Aung, E.D., Millstein, T.: Falling Back on Executable Specifications. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 552–576. Springer, Heidelberg (2010)
16. Siskind, J.M., McAllester, D.A.: Screamer: A portable efficient implementation of nondeterministic common lisp. IRCS Technical Reports Series (1993)
17. Spivey, J.: Understanding Z: a specification language and its formal semantics. Cambridge tracts in theoretical computer science. Cambridge University Press (1988)
18. Torlak, E.: A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications. PhD thesis, MIT (2008)
19. Torlak, E., Bodik, R.: Growing solver-aided languages with rosette. In: Proceedings of the 2013 Onward! ACM (2013)
20. Yang, J., Yessenov, K., Solar-Lezama, A.: A language for automatically enforcing privacy policies. ACM SIGPLAN Notices (2012)