αRby—An Embedding of Alloy in Ruby

September 20, 2013

Aleksandar Milicevic

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

Abstract. A preliminary idea of embedding the Alloy modeling language in Ruby is presented in an informal and informative way. The main goal is to design a unified programming language (called αRby from now on) which is both formally analyzable (symbolic) and fully executable (concrete). This duality of the language, which allows two different execution modes of the same code, is what makes it different from a plethora of other research projects revolving around the idea of mixing in declarative programming in an imperative language (by means of embedding a symbolic constraint solver).
A brief introduction to Alloy is presented first. The main language constructs of αRby are explained next, as well as how the actuall embedding in Ruby is achieved. Several examples are given to illustrate what certain (well-known) Alloy models look like in αRby. Finally, benefits to both Alloy users and Ruby programmers are discussed.

1 Alloy Background

The Alloy Language Alloy is a first-order, relational, modeling language. Alloy is declarative (based on logic), and as such, it is not executable per se; it is instead used primarily to formally model systems for the purpose of checking various logical properties against them.

Example Figure 1(b) shows an Alloy model of a file system. The model consists of several *signatures* (sigs): `Name`, `Obj`, `Entry`, `File`, `Folder`, and `Root`. `Obj` is a base type for all file system objects, `File` and `Folder` are concrete file system objects, and `Root` is a singleton instance of `Folder` representing the root of the file system. `Name` is just a symbolic name and `Entry` represents a name assigned to a file system object.

Object Models Sigs in Alloy are similar to classes in object-oriented languages: each sig represents a data type, a sig can be abstract (e.g., **`abstract sig`** `Obj`), a sig can *extend* another sig and create a subtype (e.g., `File` **`extends`** `Obj`), and sigs can also have *fields* (e.g., an `Entry` is a pair of exactly one[1] Name and one Obj, and a `Folder` contains a set of Entries and at most one parent Folder).

Relational Nature In Alloy everything is a relation. Every sig is a unary relation (i.e., set), and every field is a relation of arity greater than 1, mapping the field's owner type to its declared type. All fields in this example are binary; for example, the `contents` field is a binary relation (from `Entry` to `Obj`), and so is the `entries` field (don't let the **`set`** `Entry` part confuse you, that is just a constraint on that relation, allowing the same folder to be mapped to multiple entries). Alloy also supports relations of arity greater than 2; instead of having the `Entry` sig, we could have declared the `entries` field in `Folder` to be of type `Name -> Obj`, which would have made it a ternary relation).

Model Finding The *Alloy Analyzer* is a fully automated solver for the Alloy language. It takes an Alloy model and with a single click of a button finds satisfying instances of that model. Ignoring the rest of the file system model for now (all the logic constraints appended to sigs and all the predicates in Figure 1(b)), we can ask the Alloy Analyzer to find some instances satisfying the object structure defined in the model so far. Instantiating a model means assigning a set of *atoms* to each sig, and a set of *tuples* for each field (a tuple is just a list of existing atoms). Note that those are two separate object spaces: sigs and fields exist in the *model*, and atoms and tuples exist in the *instance* space.

[1] There are no `null`s or `nil`s in Alloy. Everything is a set or relation, so there are empty sets instead. When a field doesn't have an explicit modifier (e.g., **`one`**, **`lone`**, **`set`**, etc.), **`one`** is the default, meaning that the field value is constrained to be a singleton set.

Type Modifiers Type modifiers (e.g., **one**, **set**, **abstract** ...) impose constraints on corresponding relations. For example, **abstract sig** Obj means that the model instances may not contain any atoms of the Obj set, and **one sig** Root similarly means that there must be exactly one atom assigned to the Root set. Field types also accept modifiers (**abstract** is not a valid field type modifier, however) with a similar semantics.

Appended Facts Unlike in object-oriented languages, sigs can have appended facts (blocks enclosed in curly braces immediately following the field declaration block) to further constrain what valid instances may be. Appended fact for the Root sig is the simplest one; it says that for every instance (atom) of Root (of which there must be exactly one, due to "one sig Root") the value of its parent field (this.@parent) must be an empty relation (**no**).

Relational Transpose and Inverse Fields In Alloy, all fields are global, i.e., accessible from anywhere in the model, regardless of where they are defined. This (together with the transpose operator (~), which when applied to a relation simply reverses the order of the columns in that relation) allows us to easily implement a concept of *inverse fields*. It is now easy and succinct to append a fact to the Entry sig to assert that for each entry there must be exactly one folder pointing to it (via its entries field): **one** this.~@entries (without using the transpose operator, this could also be written as **one** @entries[this]).

Join Chains The dot (.) operator in Alloy, which in many ways looks an behaves like the field dereferencing operator in traditional OO languages, is actually a plain relational join operator. This allows for multi-step join chains, without worrying about "null dereferencing" at some point in the middle. For example, to obtain all contents of a folder d, we can simply write d.entries.contents, which always evaluates a set of Objs. If d.entries happens to be empty, the result will be empty as well. In contrast to OO languages, the programmer would have to check separately that d.entries is not nil, but then also, since d.entries is of type "set of Entry" (or "list of Entry"), the d.entries.contents syntax wouldn't work (in Ruby it would have to be replaces with d.entries.map(&:contents)). In our file system model, this construct is used in the File appended fact to assert that for each file there is at least some folder that contains it[2].

Transitive Closure Another powerful feature of relational arithmetic is the *transitive closure* operator (^). It can only be applied to binary relations, and its meaning is that the relation should be fully expanded by iteratively joining the current result with the original relation until a fixpoint is reached (e.g., ^b = b + b.b + (b + b.b).b + ...). This in practice allows us to succinctly express graph traversal operations, and to say stuff like "start from object 'o' and return all objects reachable from it by following a given field as many times as it takes". The *reflexive transitive closure* operator (*) works the same as ^, except that (loosely speaking) it includes the "starting object" in the final result (object 'o' above). The appended fact for Folder uses these two operators to assert that for each folder the root is reachable by following its parent pointer (Root **in** this.*@parent), and that there must be no cycles in the folder hierarchy (this !**in** this.^@parent).

Quantifiers and Bounded Analysis Alloy is based on first-order logic, so it supports first-order quantification: $\forall x \in X \bullet f(x)$ (in Alloy **all** x: X | f[x]) and $\exists x \in X \bullet f(x)$ (in Alloy **some** x: X | f[x]). Quantifiers are used in Alloy very frequently, as it is often needed to assert that a formula holds for either all or some elements of a set. Quantifiers make first order logic undecidable, so to implement a fully automated solver, the Alloy Analyzer requires the user to explicitly provide finite scopes for all sigs and therefore bound the analysis to a finite universe. The Alloy Analyzer then performs an extensive search of the universe, by means of unrolling all the sigs up to their bounds, then translating the model into a propositional formula, and finally using an off-the-shelf SAT solver to search for a satisfying assignment.

Assertion Checking As stated earlier, the main application of the Alloy language is in formal system modeling for the purpose of checking various logical properties against them. In the running file system example, we might want to check whether forbidding folder aliasing (by means of asserting that the noDirAliases predicate holds true, that is, that for each folder there is at most one other folder containing it) implies that each folder other than Root has

[2] The @ prefix for field names is not necessary in general, but it is commonly used in appended facts to help the Alloy type checker decide whether the field should be implicitly dereferenced for the implicit this instance or not (the @ prefix instructs not to perform implicit dereferencing.)

exactly one parent. There are two versions on the latter statement: one is buggy (`oneParent_buggyVersion`) and one is correct (`oneParent_correctVersion`). Analyzing the first check (`onParent_buggyVersion[] => noDirAliases[]`) returns a counterexample, meaning that the assertion is invalid. The second check does not produce a counterexample, meaning that the assertion is valid for the given small scope (which in this case is up to 5 atoms of each sig).

Executability Alloy is not executable per se because of its fully declarative nature based on first-order logic. Traditional OO-style methods, which instead of checking a proposition and returning a boolean value (like predicates) actually compute a new state, can be specified in Alloy, but again, fully declaratively, saying **what** the new state should look like, and not **how** that state can be algorithmically reached (see the `add` and `delete` predicates in the Address Book example in Figure 2(b)). To execute declarative specification like these, constraint solving typically required (for which there is no polynomial-time algorithm), so it is rarely used in practice.

Benefits of αRby Most of Alloy's expressive power, however, comes from its relational base and its relational operators, which, in contrast, can be efficiently executable in the context of object-oriented programs. Even the quantifiers can be executed by using a straightforward iterative search over the bounded domains of quantified variables, which can be appropriate when the domain is small and/or the constraints are simple. The goal of the αRby language is therefore to find a middle ground and combine the two approaches by providing a single language with two execution modes. αRby as such can be used either (1) mostly as a modeling language (for writing models with complex declarative constraints mostly for the purpose of formal analysis, but still integrated with a full-blown programming language for easy interaction with the solver, preprocessing of user inputs, post-processing of the analysis results, etc.), or (2) mostly as an imperative OO programming language inside Ruby, with added benefits of types and type checking (when needed), simple logic assertions that can be efficiently and automatically checked at runtime, with very limited possibilities for formal analysis, or (3) anywhere in between.

2 The αRby Language

αRby is implemented as a domain-specific language embedded in Ruby—all syntactically correct αRby programs must at the same time be syntactically correct Ruby programs. Thanks to Ruby's great flexibility and a fairly liberal parser, it is possible to create embedded languages that allows programs to look very different from the standard Ruby programs. αRby uses that to its advantage to imitate the syntax of Alloy as closely as possible; the biggest difference lies in the syntax for expressions, since Ruby does not allow new infix operators to be defined (e.g., an Alloy expression like "`x in y`" in αRby must be written in as "`x.in? y`", or "`x.in?(y)`", or "`in? x, y`", etc.

2.1 Dynamic Creation of Modules, Classes and Methods

One of the main goals of αRby, as previously stated, is to have a dual semantics—although αRby programs are primarily used to represent Alloy models, they should (as much as possible) at the same time have an intuitive object-oriented semantics as standalone Ruby programs. To achieve this goal, the main top-level language concepts in Alloy have their counterparts in language concepts of Ruby: Alloy modules are represented as Ruby modules, sigs are Ruby classes, and functions, predicates, assertions, facts, and commands are Ruby methods. For each of those concepts, αRby provides appropriate builder functions:

- **`alloy`** — creates a new Alloy model (module) and generates a Ruby module to represent (implement) the newly created model;

```
# Creates a new module and assigns a constant to it (named +name+
# in the current scope).  Then executes the +&body+ block using
# +module_eval+.  All Alloy sigs must be created inside an "alloy"
# block.  Inside of this module, all undefined constants are
# automatically converted to symbols.
#
# @param name [String, Symbol] --- model name
# @return [ModelBuilder] --- the builder used to create the module
def alloy(name="", &body)
```

– **sig** — creates a new Alloy sig and generates a Ruby class (in the current scope) to represent (implement) the newly created sig.

```
# Creates a new class, subclass of either +Alloy::Ast::Sig+ or a
# user supplied super class, and assigns a constant to it (named
# +name+ in the current scope)
#
# @param args [Array] --- valid formats are:
#    (1) +args.all?{|a| a.respond_to :to_sym}+
#          for each +a+ in +args+ creates an empty sig with that
#          name and default parent
#
#    (2) [Class, String, Symbol], [Hash, NilClass]
#          - for class name:   +args[0].to_s+ is used
#          - for parent sig:   the default is used
#          - for class params: +args[1] || {}+ is used
#
#    (3) [MissingBuilder], [Hash, NilClass]
#          - for class name:   +args[0].name+ is used
#          - for parent sig:   +args[0].super || default_supercls+ is used
#          - for class params: +args[0.args.merge(args[1])+ is used
#
# @param body [Proc] --- if the block is given insided the
#    curly braces, it is interpreted as appended facts;
#    otherwise (given inside do ... end) it is evaluated using
#    +class_eval+.
#
# @return [SigBuilder]
def sig(*args, &body)
```

– **fun**, **pred**, **fact**, **assertion**, **check**, **run** — each respectively creates a new Alloy function, predicate, fact, assertion, check or run, and generates a Ruby method in the current scope (either current class or module) to represent (implement) the newly created function.

```
# Creates a new function inside the current scope (either class
# corresponding to an Alloy sig, or module corresponding to an
# Alloy model.
#
# @param args [Array] --- valid formats are
#    (1) [String, Symbol]
#          - for fun name:    +args[0].to_s+ is used
#          - for args:        +block+ param names mapped to nil
#                             types are used
#          - for return type: nil is used
#          - for body:        +block+ is used
#
#    (2) [String, Symbol], [Hash(String, AType)]
#          - for fun name:    +args[0].to_s+ is used
#          - for args:        +args[1][0..-1]+ is used
#          - for return type: +args[1][-1].value is used
#          - for body:        +block+ is used
#
#    (3) [String, Symbol], [Hash(String, AType)], [AType]
#          - for fun name:    +args[0].to_s+ is used
#          - for args:        +args[1]+ is used
#          - for return type: +args[2]+ is used
#          - for body:        +block+ is used
#
#    (4) [MissingBuilder]
#          - for fun name:    +args[0].name+ is used
#          - for args:        +args[0].args+ is used
#          - for return type: +args[0].ret_type+ is used
#          - for body:        +args[0].body || block+ is used
#
# @param block [Proc] --- defaults to +proc{}+
#
# @return [Alloy::Ast::Fun]
def fun(*args, &body)
```

2.2 Responding to Missing Methods and Constants

To avoid requiring the programmer to pass strings or symbols for every new sig or fun definition (e.g., `sig :Name`), αRby overrides methods `const_missing` and `method_missing` and instead of raising an error returns an instance of `MissingBuilder`; consequently, definitions like `sig Name` become possible. The `method_missing` method is overridden in the Ruby module corresponding to the current Alloy model; the `const_missing` method, however, has to be overridden in the `Object` and `Module` classes as well, which has system-wide effects. To avoid disruptive changes to the entire Ruby environment, our implementation checks and only acts if the call was made from inside an Alloy model definition (that is, block passed to the **`alloy`** method). Furthermore, to catch unintended conversions (e.g., typos) as early as possible, αRby raises a syntax error every time a `MissingBuilder` instance is not "consumed" by the end of its scope block (missing builders get consumed only by certain language constructs, e.g., the **`sig`** method, **`fun`** method, etc.).

Automatically converting undefined methods/constants to `MissingBuilder`s comes convenient especially for providing special syntax for defining signature fields and super sigs, and function parameters and return types. This syntax is achieved by implementing methods `:"[]"` and `:"<"` in class `MissingBuilder`: the first invocation of `:"[]"` stores named arguments (a hash of `Symbol` $\rightarrow$ `AType` pairs), the second invocation of the same method stores the return type, while the invocation of `:"<"` stores the super type.

```
# @attribute name [Symbol]     --- the original missing symbol
# @attribute args [Hash]       --- arguments (first invocation of :[])
# @attribute ret_type [Object] --- return type (second invocation of :[])
# @attribute body [Proc]       --- block passed to :initialize
# @attribute super [Object]    --- syper type (passed in an invocation of :<)
class MissingBuilder
  # Sets the value of the @super attribute.  If another
  # +MissingBuilder+ is passed as an argument, it merges the
  # values of its attributes with attribute values in +self+.
  def <(super_type)

  # The first invocation expects a +Hash+ or an +Array+ and sets
  # the value of @args.  The second invocatin expets a singleton
  # array and sets the value of @ret_tupe.  Any subsequent
  # invocations raise +SDGUtils::DSL::SyntaxError+.
  def [](*args)
end
```

2.3 Symbolic Execution

The purpose of symbolic execution is to obtain a symbolic expression equivalent to a Ruby-defined method (corresponding to an Alloy function). To this end, instead of implementing a full parser for Alloy expressions, we simply run the Ruby method, passing a list of symbolic variables as its arguments, and observe the return value (which is expected to be the final symbolic expression). This is possible in Ruby since most operators can be overridden: the initial symbolic arguments can therefore override all the relevant operators and return symbolic expressions instead of concrete values. For example, if `a` is symbolic variable, and therefore it overrides the `==` operator, then `a == b` returns `Alloy::Ast::Expr::BinaryExpr.equals(a, b)` and not a concrete boolean value (which would be the default behavior).

2.4 Online Source Instrumentation

For each Alloy function two Ruby methods are generated: (1) a verbatim copy of the user defined function, and (2) an Alloy version of the same method obtained by instrumenting the original source for the purpose of symbolic execution. The need for the instrumentation rises from the fact that certain operators and control structures cannot be overridden; examples include all if-then-else variations (e.g., "**`if`** `<cond>; <then_block>` **`else`** `<else_block>` **`end`**", "`<block>` **`unless`** `<cond>`", `<block1>` **`and`** `<block2>`" etc.), and logic operators (e.g., `&&`, `||`, etc.). Our instrumentation works by using an off-the-self parser to parse the source code, then traversing the generated AST, and finally replacing the aforementioned constructs with appropriate Alloy expressions. For example,

```
one d.parent if d.not_in?(Root)
```

gets translated to

```
Alloy::Ast::Expr::ITEExpr(d.not_in?(Root), proc{one d.parent}, proc{})
```

This "traverse and replace" algorithm is by far simpler than implementing a full parser for Alloy expressions.

2.5 Making Distinction Between Equivalent Ruby Constructs

Ruby allows different syntactic constructs for the same underlying operation. For example, Ruby blocks can be enclosed either between the **`do`** and **`end`** keywords or between curly braces (`{}`)[3]. Similarly, some built-in infix operators

[3] There is a caveat, however: using different syntactic forms for specifying blocks can affect the Ruby parse tree. For example, **`abstract sig`** `S {}` has a different parse tree from **`abstract sig`** `S` **`do end`**; in the former case the block is passed to the **`sig`**

(which translate to regular method calls) 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 is easy to detect and instrument some of these cases for the purpose of assigning different semantics to different syntactic forms.

It makes a lot of sense to do this in αRby, especially for the `*` operator, since `a*b` and `a.*b` have different semantics in Alloy (the former is an arithmetic multiplication, and the latter is a reflective transitive closure). αRby also makes a distinction between the two different forms of blocks appended to sig definitions: if the syntax with curly braces is used, the block is interpreted as Alloy appended facts (e.g., all sig definitions in Figure 1(a)); if the **`do end`** syntax is used, the block is interpreted as a class body and it is evaluated using Ruby's `class_eval` (convenient for defining "instance" functions/predicates for sigs, as in Figure 2(a)). Finally, in Ruby "`<b2>` **`if`** `<b1>`" and "`b1` **`and`** `b2`" have exactly the same semantics, in αRby, however, the **`and`** and **`or`** Ruby keywords always have the semantics of boolean conjunction and disjunction operators.

method, and its result then to the **`abstract`** method, whereas in the latter case, the block is passed to the **`abstract`** method, along with the result previously produced by **`sig`** `S`.

(a) File System in αRby

```
alloy :FileSystem do
  sig Name

  abstract sig Obj

  sig Entry [
    name: Name,
    contents: Obj
  ] {
    one self.entries!
  }

  sig :File < Obj {
    some d: Folder do
      self.in? d.entries.contents
    end
  }

  sig (Folder < Obj) [
    entries: (set Entry),
    parent: (lone Folder)
  ] {
    parent == self.contents!.entries! and
    self.not_in? self.^:parent and
    (self.*:parent).contains? Root and
    all e1: Entry, e2: Entry do
      e1 == e2 if (e1 + e2).in?(entries) && e1.name == e2.name
    end
  }

  one sig Root < Folder {
    no parent
  }

  # all directories besides root have one parent: buggy
  pred oneParent_buggyVersion {
    all d: Folder do
      one d.parent if d.not_in?(Root)
    end
  }

  # all directories besides root have one parent: correct
  pred oneParent_correctVersion {
    all d: Folder do
      (one d.parent and one d.contents!) if d.not_in?(Root)
    end
  }

  # Only files may be linked (i.e., have more than one entry).
  # That is, all directories are the contents of at most one
  # directory entry.
  pred noDirAliases {
    all o: Folder do
      lone o.contents!
    end
  }

  check("for 5") {
    noDirAliases if oneParent_buggyVersion
  }
  check("for 5") {
    noDirAliases if oneParent_correctVersion
  }
end
```

(b) File System in Alloy

```
module FileSystem
sig Name  {}

abstract sig Obj  {}

sig Entry  {
  name: Name,
  contents: Obj
} {
  one this.~@entries
}

sig File extends Obj {} {
  some d: Folder {
    this in d.@entries.@contents
  }
}

sig Folder extends Obj {
  entries: set Entry,
  parent: lone Folder
} {
  this.@parent = this.~@contents.~@entries
  this !in this.^@parent
  Root in this.*@parent
  all e1: Entry, e2: Entry {
    e1 + e2 in this.@entries && e1.@name = e2.@name => e1 = e2
  }
}

one sig Root extends Folder {} {
  no this.@parent
}

// all directories besides root have one parent: buggy
pred oneParent_buggyVersion {
  all d: Folder {
    d !in Root => one d.parent
  }
}

// all directories besides root have one parent: correct
pred oneParent_correctVersion {
  all d: Folder {
    d !in Root => one d.parent && one d.~contents
  }
}

// Only files may be linked (i.e., have more than one entry).
// That is, all directories are the contents of at most one
// directory entry.
pred noDirAliases {
  all o: Folder {
    lone o.~contents
  }
}

check  {
  oneParent_buggyVersion[] => noDirAliases[]
} for 5

check  {
  oneParent_correctVersion[] => noDirAliases[]
} for 5
```

Fig. 1. "File System" example in αRby (on the left) and its automated translation into Alloy (on the right)

(a) Address Book in αRby

```
alloy :AddressBook do
  sig Name
  sig Addr
  sig Book [
    addr: Name ** (lone Addr)
  ] do
    pred add[ans: Book, n: Name, a: Addr] {
      ans.addr == addr + n**a
    }

    pred del[ans: Book, n: Name] {
      ans.addr == addr - n**Addr
    }
  end

  assertion delUndoesAdd {
    all [:b1, :b2, :b3] => Book, n: Name, a: Addr do
      if b1.addr[n].empty? && b1.add(b2, n, a) && b2.del(b3, n)
        b1.addr == b3.addr
      end
    end
  }

  assertion addIdempotent {
    all [:b1, :b2, :b3] => Book, n: Name, a: Addr do
      if b1.add(b2, n, a) && b2.add(b3, n, a)
        b2.addr == b3.addr
      end
    end
  }
end
```

(b) Address Book in Alloy

```
module AddressBook
sig Name  {}
sig Addr  {}
sig Book  {
  addr: Name -> lone Addr
}
pred add[self: Book, ans: Book, n: Name, a: Addr] {
  ans.addr = self.addr + n -> a
}

pred del[self: Book, ans: Book, n: Name] {
  ans.addr = self.addr - n -> Addr
}


assert delUndoesAdd {
  all b1: Book, b2: Book, b3: Book, n: Name, a: Addr {
    ((no n.(b1.addr)) && b1.add[b2, n, a] && b2.del[b3, n]) implies {
      b1.addr = b3.addr
    }
  }
}

assert addIdempotent {
  all b1: Book, b2: Book, b3: Book, n: Name, a: Addr {
    (b1.add[b2, n, a] && b2.add[b3, n, a]) implies {
      b2.addr = b3.addr
    }
  }
}
```

Fig. 2. The “Address Book” example in αRby (on the left) and its automated translation into Alloy (on the right)