An Imperative Extension to Alloy

Joseph P. Near and Daniel Jackson

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

Abstract. We extend the Alloy language with the standard imperative constructs; we show the mix of declarative and imperative constructs to be useful in modeling dynamic systems. We present a translation from our extended language to the existing first-order logic of the Alloy Analyzer, allowing for efficient analysis of models.

1 Introduction

We present an extension to the Alloy language [1] for the specification of dynamic systems. The typical approach to modeling dynamic systems, and the one taken by Z [2], VDM [3], and DynAlloy [4, 5], is to model state changes using pre- and post-conditions on each transition. Both the existing idioms for modeling dynamic systems in Alloy and our approach support this technique; we add the standard imperative constructs: assignment, sequential composition, guards, and loops. We give these operators the expected, *operational*, semantics.

Moreover, our language extension allows for the separation of the static and dynamic elements of a model. Our extension allows dynamic operations to be added to a static model: it makes updates to mutable state explicit and separates imperative operations from static properties. This separation of concerns is important to the design of a system, and is not well-supported by the Alloy idioms currently in use.

The use of imperative operators in specifications simplifies the process of implementation. Using our language extension, modelers have the option of refining a specification (in the style of Morgan [6]) until the modeler can easily translate it into an imperative implementation. Each refinement step is automatically checked by the Alloy Analyzer to ensure that no errors have been made.

These advantages come at no loss of expressive power. We place no restrictions on the existing language, and allow actions to be defined declaratively, using pre- and post-conditions; our framework and composition operators also apply to these declarative actions.

The contributions of this paper are:

- an extension to the Alloy language consisting of the standard imperative operators (Section 3);
- a set of examples showing how the extension may be used to model dynamic systems concisely (Section 4);

– a translation from the action language of the extension to the first-order logic supported by the Alloy Analyzer, allowing for the efficient analysis of models written using the extension (Section 5).

2 Alloy & Dynamic Systems

Alloy [1] is a modeling language based on first-order relational logic with transitive closure. It is designed to be simple but expressive, and to be amenable to automatic analysis. As such, few features are provided beyond first-order logic and transitive closure, making the semantics of the language easily expressible, understandable, and extendable.

The Alloy Analyzer supports fully automatic analysis of Alloy models. While this analysis is bounded and thus not capable of producing proofs, it does allow for incremental, agile development of models; and the *small-scope hypothesis* [7]—which claims that most inconsistent models have counterexamples within small bounds—means that modelers may have high confidence in the results. This sacrifice of completeness in favor of automation is in line with the *lightweight formal methods* philosophy [8].

Alloy's universe is made up of uninterpreted atoms; *signatures* define the sets into which these atoms are partitioned. For example [1], the following signatures define sets of names and addresses:

```
sig Name {}
sig Addr {}
```

Similarly, this signature defines an address book with a field "addr" mapping names to addresses:

```
sig Book { addr: Name →lone Addr }
```

Operations that modify the state of an address book may be defined as *predicates* over pre- and post-states:

```
pred add [b, b': Book, n: Name, a: Addr] {
  b'.addr = b.addr + n→a }
```

We can use Alloy's "**check**" command to check that that the "add" operation correctly updates the address book.

```
check {
  all b,b': Book, n: Name, a: Addr |
    add[b, b', n, a] ⇒ n.b'.addr = a }
```

The pre- and post-state idiom is well-known, both in the context of declarative specification and in functional programming. It is the basis of the idioms for modeling dynamic systems in Alloy and of the monadic theory of state used in functional languages such as Haskell. While this technique often produces concise, readable models, it is not adept at expressing certain types of imperative control flow. The following excerpt, for example, is taken from a previously published Alloy model of a flash filesystem [9], and uses a common trace-based idiom:

```
some stateSeq: StateSeq |
  stateSeqConds[stateSeq, numBlocksToProgram + 1] &&
  all trscSeq_idx : stateSeq.butlast.inds |
    programBlock[stateSeq, trscSeq_idx, cfsys, inode, startBlockIdx]
```

The specification for the flash memory requires blocks to be written in sequence. In the traditional approach, this would be expressed using multiple operations, one for each write; a sequence of such operations would then be shown to refine an abstract write that occurs in a single step. This approach can be tedious and unnatural, however, as it becomes necessary to encode the control flow explicitly in the state, using preconditions to constrain the ordering. Consequently, many modelers prefer to describe such a behavior using a single operation whose execution involves multiple steps. This notion has no standard formulation in Alloy; here, the modeler has introduced a special signature, "StateSeq", to model a sequence of states, which happens to be used only inside this operation.

The Haskell community, having encountered precisely the same situation, introduced special syntax for expressing sequential operations. One way to view this paper is as an attempt to provide the same facilities to Alloy modelers. Using our language extension, the excerpt above can be written as follows:

```
Cnt.idx := 0 ;
loop {
  programBlock[Cnt.idx, cfsys, inode, startBlockIdx];
  Cnt.idx := Cnt.idx + 1
} && after Cnt.idx = numBlocksToProgram
```

An operational language extension, with operational semantics, can thus make some models easier to write. The basic operations—state update, conditionals, loops, and so on—can be proved correct. All models written in the extension use the same mechanism for expressing dynamic operations, making models easier to read. Imperative operators can make sequential operations more concise. And models written using a standard operational mechanism can be optimized for efficient analysis.

3 Language Extension

A small extension to the Alloy language, summarized in this section, supports the modeling of dynamic systems.

3.1 Dynamic Fields

Immutable fields are declared in the traditional way:

```
sig Addr {}
sig Name {}
```

Mutable fields, whose values may vary with time, are defined using the "**dynamic**" keyword:

```
one sig Book { addr: dynamic (Name →lone Addr) }
```

3.2 Named Actions

Named actions can be defined at the top level, and can be invoked from within other actions. Adding an entry to the address book, for example, can be written as a named action that adds the appropriate tuple:

```
action add[n:Name, a:Addr] {
  Book.addr := Book.addr + (n →a) }
```

The deletion operation, on the other hand, removes all tuples containing a given name from the book:

```
action del[n:Name] {
  Book.addr := Book.addr − (n →Addr) }
```

3.3 Action Language

Our action language includes operators for imperative programming: field update, sequential composition, and loops. Pre- and post-conditions employ boolean-valued formulas (written φ) with the existing syntax and semantics of Alloy.

Act ::=	$o_1.f_1, \ ..., \ o_n.f_n := e_1, \ ..., \ e_n$	(field updates)
|	*Act* ; *Act*	(sequential composition)
|	**loop** { *Act* }	(loop)
|	*action*$[a_1, \ ..., \ a_n]$	(action invocation)
|	**before** φ | **after** φ	(pre- and post-conditions)
|	**some** v : τ| *Act*	(existential quantification)
|	*Act* $\Rightarrow$ *Act* | *Act* $\wedge$ *Act* | *Act* $\vee$ *Act*	

A *field update* action changes the state of *exactly* those mutable fields mentioned, *simultaneously*. The action

```
b.addr := b.addr + (n→a)
```

for example, adds the mapping n→a to the address book b, while

```
a.addr, b.addr := b.addr, a.addr
```

swaps the entries of address books "a" and "b".

Sequential composition composes two actions, executing one before the other:

```
add[n,a]; del[n,a]
```

performs the "add" operation and then the "del" operation.

A loop executes its body repeatedly, nondeterministically choosing when to terminate. The standard conditional loop may be obtained through the use of a post-condition; the action

```
loop { dec[Cnt.idx] } && after Cnt.idx = 0
```

for example, runs the "dec" action until "Cnt.idx" reaches zero. Because they are nondeterministic, execution of these loops generally requires backtracking.

We view actions as relations between initial and final states. This view of actions allows for the lifting of the standard logical connectives and existential

quantification into our action language, and for the mixing of declarative constraints with actions. The "**before**" and "**after**" actions, for example, introduce declarative pre- and post-conditions; these act as filters on other actions when combined using the logical connectives. The action

add[n,a] ⇒ **after** n.Book.addr = a

for example, has executions that either end with the correct mappings in the address book or are not executions of "add."

3.4 Temporal Quantifiers

Actions have as free variables their beginning and ending states. Temporal quantifiers bind these variables: "**sometimes**," existentially; and "**always**," universally.

φ ::= < Alloy Formula >
| **sometimes** | *Act*
| **always** | *Act*

Given our view of actions as relations, a "**sometimes**" formula holds if and only if the action in its body relates *some* initial and final states; an "**always**" formula holds if and only if it relates *all* states. To visualize the result of adding the mapping n→a to the address book, for example, one executes the Alloy command:

run { **sometimes** | add[n,a] }

One can also check that "add" adds the mapping in all cases:

check { **always** | add[n,a] ⇒ **after** n→a **in** Book.addr }

4 Examples

4.1 River Crossing

River crossing problems are a classic form of logic puzzle involving a number of items that must be transported across a river. Some items cannot be left alone with others: in our problem, the fox cannot be left with the chicken, or the chicken with the grain. A correct solution moves all items to the far side of the river without violating these constraints. We begin by defining an abstract signature for objects, each of which eats a set of other objects and has a dynamic location. The objects of the puzzle are then defined as singleton subsets of the set of objects. Similarly, an abstract signature defines the set of locations, and two singleton sets partition it into the near and far sides of the river.

```
abstract sig Object { eats: set Object,
                      location: dynamic Location }
one sig Farmer, Fox, Chicken, Grain extends Object {}
abstract sig Location {}
one sig Near, Far extends Location {}
```

We define the "eats" relation to reflect the puzzle by constraining it to contain exactly the two appropriate tuples.

```
fact eating { eats = (Fox →Chicken) + (Chicken →Grain) }
```

The "cross" action picks an object o for the farmer to carry across the river, a new location fl for the farmer, and a (possibly new) location ol for o, and moves the farmer and the object.

```
action cross {                                    -- pick an object & two locations
  some o: Object − Farmer, fl: Location − Farmer.location, ol: Location |
  (Farmer.location := fl , o.location := ol) && -- move the object and farmer;
    after (all o: Object |                        -- all objects end up with
              o.location = Farmer.location ||     -- the farmer, or not with
            (all o': (Object − o) |               -- objects they eat
              o'.location = o.location ⇒ o !in o'.eats)) }
```

To obtain a solution, we find an execution that begins with all objects on the near side, calls "cross" repeatedly, and ends with all objects on the far side.

```
pred solvePuzzle {
  sometimes |                                     -- find some execution in which:
    before (all o: Object | o.location = Near) && -- objects start on near side,
    loop {
      cross []                                    -- cross runs repeatedly, and
    } && after (all o: Object | o.location = Far) } -- objects end on far side.
```

The "cross" action relies on the ability to mix declarative and imperative constructs: it chooses an object and a destination nondeterministically and then formulates the requirement that no object be eaten as a postcondition. In obtaining a solution, we have applied another imperative construct—**loop**—illustrating our ability to declaratively construct abstract actions and then compose them imperatively.

4.2 Filesystem

As an example of the addition of dynamic operations to a static model, we present a simple filesystem. We begin with signatures for filenames and paths. File paths are represented by linked lists of directories terminated by filenames.

```
sig Name {}
abstract sig Path {}
sig NonEmptyPath extends Path { first: Name, rest: Path }
sig EmptyPath extends Path {}
```

Next, we define the filesystem: an inode is either a directory node or a file node; a directory node maps names of files and directories to other inodes, and a file node contains some mutable data. The root node is a directory.

```
abstract sig INode {}
sig DirNode extends INode { files: Name →INode }
one sig RootNode extends DirNode {}
```

```
sig FileNode extends INode { data: dynamic Data }
sig Data {}
```

We now define operations over this static filesystem, beginning with navigation. We use a global MVar to hold the destination path, the current inode, and the data to be written to or read from the destination. One navigation step involves moving one step down the list representing the destination path and following the appropriate pointer to the corresponding inode.

```
one sig MVar { path: dynamic Path,
   current: dynamic INode, mdata: dynamic Data }

action navigate {                                  -- follow the path one step and then
   MVar.path := MVar.path.rest;                    -- update ''current'' to point to the
   MVar.current := (MVar.path.first).(MVar.current.files)}-- corresponding inode
```

Reading from a file involves calling "navigate" until the destination inode has been reached and then reading its data into "MVar." Writing, similarly, involves navigation followed by a write.

```
action read {
   loop {
     navigate []                                   -- call navigate repeatedly
   } && after MVar.current in FileNode;            -- until we have reached the file inode
   MVar.mdata := MVar.current.data }               -- then read its data into MVar

action write {
   loop {
     navigate []                                   -- call navigate repeatedly
   } && after MVar.current in FileNode;            -- until we have reached the file inode
   let  file  = MVar.current |                     -- take the data from MVar
      file .data := MVar.mdata }                   -- and write it to the file inode
```

We would like a write to the filesystem followed by a read to yield the written data. We can verify this property by writing arbitrary data to an arbitrary file, reading it back, and checking that the result is the original data. We use a global "Temp" to hold the original data.

```
one sig Temp { tdata: dynamic Data }

assert readMatchesPriorWrite {
  always |                                        -- if we begin at the root node,
    before (MVar.current = RootNode &&            -- and no file contains
            no f: FileNode | f.data = MVar.mdata) && -- MVar.mdata
    write [];                                     -- and we write MVar.mdata,
    Temp.tdata := MVar.mdata;                     -- store the original data,
    read [] ⇒                                     -- and read back the data
      after Temp.tdata = MVar.mdata }             -- then they're the same
```

This model illustrates the ability to build up multi-step actions using loops and sequential composition, and to verify properties of those actions.

4.3 Insertion Sort

Following Morgan [6], we present insertion sort as a refinement from a declarative specification to a deterministic, imperative implementation. We begin by defining mutable sequences of naturals and a declarative sortedness predicate.

```
sig Sequence { elts: dynamic seq Natural }
pred sorted[elts: seq Natural] { -- each element is less than the next
  all i: elts.inds - elts.lastIdx | let i' = i + 1 | i.elts <= i'.elts }
```

Using this predicate, we can define a declarative sorting operation.

```
action declarativeSort [s: Sequence] {
  some s': Sequence |
    before (sorted[s'.elts] && s.elts = s'.elts) &&
    s.elts := s'.elts }
```

To bring this model closer to executable code, we define insertion sort as a series of swaps of elements of a sequence. We begin with a global counter and a declarative predicate to find the index of a sequence's smallest element, leaving the imperative definition of this predicate for later.

```
one sig Cnt { cur: dynamic Int }
pred minIdx [s: seq Natural, c, i: Int] { -- i is the index greater than c whose
  i >= c && no i': s.inds | i' >= c && i'.s < i.s }    -- value in s is smallest
```

Next, we define the insertion step, in which the first element in the sequence is swapped, using relational override (++), with the smallest one.

```
action insertionStep [s: Sequence] {
  some i: s.elts.inds |                          -- nondeterministically pick an index
    (before minIdx[s.elts, Cnt.cur, i]) &&  -- whose element is smallest
    Cnt.cur := Cnt.cur + 1,                  -- and swap it with the first element
     s.elts := s.elts ++((Cnt.cur)→i.(s.elts)) ++(i→Cnt.cur.(s.elts)) }
```

The sorting action simply sets the counter to zero and runs the insertion step to the end of the sequence.

```
action insertionSort [s: Sequence] {
  Cnt.cur := 0;
  loop {
    insertionStep[s]
  } && after Cnt.cur = s.elts.lastIdx  }
```

Next, we show that the sort is correct by verifying that an arbitrary sequence is sorted when the sort completes.

```
assert sortWorks {
  all s: Sequence |
    always | insertionSort[s] ⇒ after sorted[s.elts] }
```

We now return to the problem of finding the minimum unsorted element in the sequence. We begin with a bit of global state to hold the current index in the search and the value and index of the minimal element found so far.

```
one sig Temp {idx: dynamic Int, min: dynamic Natural, minIdx: dynamic Int}
```

Next, we define an action to iterate over the subsequence s, checking each element against the minimal one found so far.

```
action findMin[s: Sequence] {
  Temp.idx := Temp.idx + 1;          -- increment the current index
  -- if the current value is less than the previous minimum, remember it
  (before Temp.idx.(s.elts) < Temp.min ⇒
    (Temp.min := Temp.idx.(s.elts), Temp.minIdx := Temp.idx)) &&
  (before Temp.idx.(s.elts) >= Temp.min ⇒ skip) } -- else nothing
```

Finally, we redefine insertionStep to use our new action.

```
action insertionStep [s: Sequence] {   -- start at the current index,
  Temp.idx := Cnt.cur, Temp.min := Cnt.cur.(s.elts), Temp.minIdx := Cnt.cur;
  loop {   -- run findMin over the suffix of the sequence,
    findMin[s]
  } && after Temp.idx = s.elts.lastIdx;
  (Cnt.cur := Cnt.cur + 1,   -- and swap minimum element with the current one
   s.elts := s.elts ++((Temp.minIdx)→Cnt.cur.(s.elts))
                    ++(Cnt.cur→Temp.minIdx.(s.elts))) }
```

Since our change was only incremental, we can show that the new sort refines the old one by verifying that repeating findMin yields the same element as our declarative minIdx.

```
assert findMinWorks {
  all s: Sequence |                                   -- for all sequences...
    always |
      (before (Temp.idx = Cnt.cur &&
               Temp.min = Cnt.cur.(s.elts) &&
               Temp.minIdx = Cnt.cur) &&
      loop {                -- running findMin over the suffix of the sequence...
        findMin[s]
      } && after Temp.idx = s.elts.lastIdx) ⇒          -- finds the same element
             after minIdx[s.elts, Cnt.cur, Temp.minIdx] } -- as minIdx
```

Thus we can use the automated analysis our language extension affords us to support the stepwise refinement of a specification to executable, imperative code: our final version of insertionSort could easily be translated into an imperative programming language. Moreover, we have kept the analysis of our refinements tractable by performing it in a modular fashion, refining declarative specifications one at a time and analyzing the implementation of each separately.

5 Translation to Alloy

We now present the translation (Figure 1) of our action language and associated operators into the first-order logic supported by the Alloy Analyzer.

$$
\begin{aligned}
[\![o.f := e]\!](t,t') &\triangleq o.f.t' = e[.]t \ \wedge \\
&\quad \forall f' : (fields - f) \mid o.f'.t = o.f'.t' \ \wedge \\
&\quad \forall o' : (sigs - o),\ f' : fields \mid o'.f'.t = o'.f'.t' \ \wedge \\
&\quad t' = t.next \wedge t'.pc = fresh\ pc \\
\\
[\![\ c_1\ ;\ c_2\ ]\!](t,t') &\triangleq \exists t_1 : Time \mid [\![c_1]\!](t,t_1) \wedge \\
&\qquad\qquad [\![c_2]\!](t_1,t') \\
[\![\textbf{loop}\ \{c\}]\!](t,t') &\triangleq \\
&\quad \exists\ begin, end : t.^*next - t'.\hat{}next \mid \\
&\qquad [\![c]\!](t,begin) \wedge [\![c]\!](end,t') \wedge \\
&\qquad \forall\ mid, mid' : t.^*next - end.\hat{}next \mid \\
&\qquad\quad [\![c]\!](mid,mid') \Rightarrow \exists\ mid'' : mid'.\hat{}next \mid \\
&\qquad\qquad\qquad [\![c]\!](mid',mid'') \\
[\![act[a_1,...,a_n]]\!](t,t') &\triangleq act[a_1,...,a_n,t,t'] \\
[\![\textbf{before}\ \varphi\ ]\!](t,t') &\triangleq \varphi[.]t \\
[\![\textbf{after}\ \varphi\ ]\!](t,t') &\triangleq \varphi[.]t' \\
[\![\textbf{some}\ \ v : \tau \mid c]\!]C &\triangleq \exists v : \tau \mid [\![c]\!]C \\
[\![c_1 \wedge c_2]\!]C &\triangleq [\![c_1]\!]C \wedge [\![c_2]\!]C \\
[\![c_1 \vee c_2]\!]C &\triangleq [\![c_1]\!]C \vee [\![c_2]\!]C \\
[\![c_1 \Rightarrow c_2]\!]C &\triangleq [\![c_1]\!]C \Rightarrow [\![c_2]\!]C
\end{aligned}
$$

$$
\begin{aligned}
&[\![\textbf{action}\ name[\text{a1, ..., an}]\ \{\ Act\ \}]\!] \triangleq \\
&\qquad \textbf{pred}\ name[\text{a1, ..., an, t, t'}]\ \{\ [\![Act]\!](t,t')\ \} \\
\\
&[\![\textbf{sometimes}\mid Act\ ]\!] \triangleq \exists t,t' : Time | [\![Act]\!](t,t') \\
&[\![\textbf{always}\mid Act\ ]\!] \triangleq \forall t,t' : Time | [\![Act]\!](t,t')
\end{aligned}
$$

Fig. 1. Rules for Translating the Action Language to Alloy

```
one sig Book {
  addr: dynamic (Name→lone Addr)}

action add[n:Name, a:Addr] {
  Book.addr := Book.addr + (n→a)}

assert addAdds {
  all n: Name, a: Addr |
    always | add[n,a] ⇒
      after n.Book.addr = a}
```

```
one sig Book {
  addr: Name→lone Addr→Time}

pred add[n:Name, a:Addr, t, t':Time] {
  t' = t.next && t'.pc = pc0 &&
  all o:Book−Book | o.addr.t = o.addr.t' &&
  Book.addr.t' = Book.addr.t' + (n→a) }

assert addAdds {
  all n: Name, a: Addr |
    all t, t': Time |
      add[n, a, t, t'] ⇒
        n.Book.addr.t' = a}
```

Fig. 2. Address Book Example (Left) and its Translation (Right)

5.1 Dynamic Idiom

Our translation uses two idioms that are common in the Alloy community for modelling dynamic systems. The first involves the addition of a "Time" column to each relation that represents local mutable state; the second involves the creation of a global execution trace using a total ordering on "Time" atoms.

Our translation adds a "Time" column to each **dynamic** field, and actions become predicates representing transitions from one time step to the next. We do not, however, enforce a global total ordering on time steps; instead, time steps are only partially ordered, allowing many traces to exist simultaneously.

In avoiding the single global trace, we gain the ability to compare executions, to run executions from within executions, and to run concurrent executions. The global trace does have performance and visualization benefits, however; fortunately, it is not difficult to infer that a particular analysis requires only a single trace, and then to enforce a total ordering on time steps. Our implementation performs this optimization, improving the performance and visualizability of many analyses considerably.

5.2 Translation

To translate our action language into a declarative specification following the trace-based idiom, we add a "Time" column to dynamic fields and thread a pair of variables through the action execution to represent the starting and ending time steps of that execution. We define a partial ordering on times using a field named "next:"

sig Time { next: **lone** Time }

We write the translation of action c into first-order logic in a *translation context* as $[\![c]\!](t, t')$ (or $[\![c]\!]C$ when the parts of C are not needed separately) where the context contains start and end time steps t and t'. We also assume a global set $sigs$ representing signatures with dynamic fields, and a global set of dynamic relations $fields$. We write $e[.]t$ to denote the replacement of every reference to a dynamic relation $f \in fields$ in e by the relational join $f.t$; this operation represents the evaluation of e at time t. We give the complete translation in Figure 1, and an example translation in Figure 2.

Assignment simulates the process of updating an implicit store. The first generated conjunct updates the field $o.f$ with the value of e at time t. The second and third represent the *frame condition* that the transition updates only f at o: the second ensures that the other fields of o do not change, while the third ensures the same for objects other than o. The fourth conjunct specifies that an update takes exactly one time step, and the fifth constrains the final time step's program counter.

Sequential composition is accomplished by existentially quantifying the time step connecting its two actions; loops are defined in terms of sequential composition. Action invocation passes the current time interval to the called action.

Named action definitions are translated into Alloy predicates with two extra arguments: the action's starting and ending times. The action representing the

body is translated in the context of those times. A definition of an action is translated to a standard Alloy predicate, with the before and after times made explicit.

The translation of a "**sometimes**" formula existentially quantifies the beginning and ending states related by the result of translating the action in the body of the formula, while an "**always**" formula universally quantifies these states.

5.3 Semantic Implications

Our translation gives the language's imperative constructs the same relational semantics given by Nelson [10] to Dijkstra's original language of guarded commands [11]; these semantics also correspond to the standard operational semantics [12]. In addition, the relational semantics implies the existence of a corresponding semantics in terms of the weakest liberal precondition (namely, the *wlp*-semantics of Dijkstra's guarded commands, also given by Nelson [10]). Our translation does not, however, correspond to a semantics in terms of weakest preconditions. The use of *wp*-semantics allows termination to be expressed; our language can only express partial correctness properties.

The property that an abstract action of only one step is refined by another action is directly expressible. The same property for actions of more than one step, however, is not expressible due to the known problem of unbounded universal quantifiers in Alloy [13].

6 Related Work

Our approach to modeling dynamic systems is similar to Carroll Morgan's [14], the primary difference being that Morgan defines a programming language and then adds specification statements, while we begin with a specification language and extend it with commands. Like Morgan's language, however, our command language supports the practice of refinement-based program development [6]. Our language is also similar to Butler Lampson's system specification language Spec [15], which also provides both declarative and imperative constructs. The B Method [16] also provides the same imperative constructs that we present here, and gives them the same semantics. Abstract State Machines [17] represent another operational specification technique, but ASMs lack the declarative features of Alloy. None of these approaches currently support the Alloy Analyzer's style of analysis.

Other traditional methods (such as Z [2] and VDM [3]) for specifying dynamic systems and analyzing those specifications center around the definitions of single-step operations, and do not offer a command language. Z does provide sequential composition, but no looping construct.

DynAlloy [4, 5] has a very similar motivation to our work. It likewise extends Alloy, and offers operational constructs, but based on dynamic logic rather than relational commands. Unlike our extension, however, DynAlloy extends the semantics of Alloy, and translations are not intended to be human-readable.

Alchemy [18] also defines state transitions declaratively, but has the goal of compiling Alloy specifications into imperative implementations. Since it uses an idiom-based approach to state transitions, this work has prompted an exploration [19] of the properties that a declarative specification must have in order to correctly define a transition system. The specifications generated by our translation satisfy the necessary conditions by construction.

Some similar executable languages also exist: Crocopat [20] and RelView [21] both allow the definition and execution of relational programs. While these tools can execute commands over very large relations, they cannot perform the kind of exhaustive analysis that the Alloy Analyzer supports.

7 Conclusions and Future Work

We have extended the Alloy language with imperative operators. Our examples are indicative of our experience using the extension: dynamic models can be built statically and the dynamic elements added after verification of the static model. Moreover, the addition of sequential composition and looping constructs make models of dynamic systems more concise and easier to read.

We have also experimented with refinement-oriented development from specifications to implementations. The similarity of our language extension to the programming language used by Morgan [6], in addition to our ability to perform automated analysis on each refinement step, makes this strategy very attractive.

Finally, the move towards a mix of imperative and declarative constructs blurs the line between models and implementations. We have presented example models that can be translated easily into imperative implementations; given the simplicity of this translation, we plan to explore the possibility of automating it. Nondeterministic execution strategies like Prolog's backtracking search combined with a clever translation of Alloy's relational logic may allow for the execution of a large fragment of our extended Alloy language, making possible the a more direct execution that may perform well enough to be a practical implementation.

Acknowledgements

We are grateful to Jonathan Edwards, Eunsuk Kang, and Derek Rayside for their lively discussions and helpful comments. This research was funded in part by the National Science Foundation under grants 0541183 (Deep and Scalable Analysis of Software), and 0707612 (CRI: CRD – Development of Alloy Tools, Technology and Materials), and by the Northrop Grumman Cybersecurity Research Consortium under the Secure and Dependable Systems by Design project.

References

1. Jackson, D.: Software Abstractions: logic, language, and analysis. The MIT Press (2006)

2. Spivey, J.: The Z notation: a reference manual. (1992)
3. Jones, C.: Systematic software development using VDM. Prentice Hall New York (1990)
4. Frias, M., Galeotti, J., Pombo, C., Aguirre, N.: DynAlloy: upgrading alloy with actions. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. (2005) 442–450
5. Frias, M., Pombo, C., Galeotti, J., Aguirre, N.: Efficient Analysis of DynAlloy Specifications. (2007)
6. Morgan, C.: Programming from specifications. (1990)
7. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the "small scope hypothesis". In: In Popl '02: Proceedings Of The 29th Acm Symposium On The Principles Of Programming Languages. (2002)
8. Jackson, D., Wing, J.: Lightweight formal methods. In: Roundtable contribution to: An invitation to formal methods, Hossein Saiedian, ed. Volume 29., IEEE Computer (1996) 16–30
9. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. In: Proceedings of the 1st international conference on Abstract State Machines, B and Z, Springer (2008) 294–308
10. Nelson, G.: A generalization of Dijkstra's calculus. ACM Transactions on Programming Languages and Systems (TOPLAS) **11**(4) (1989) 517–561
11. Dijkstra, E.: A discipline of programming. Prentice Hall PTR Upper Saddle River, NJ, USA (1997)
12. Pierce, B.: Types and programming languages. The MIT Press (2002)
13. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. ACM SIGSOFT Software Engineering Notes **30**(5) (2005) 216
14. Morgan, C.: The specification statement. ACM Transactions on Programming Languages and Systems (TOPLAS) **10**(3) (1988) 403–419
15. Lampson, B.: 6.826 class notes (2009) (http://web.mit.edu/6.826/www/notes/).
16. Abrial, J.: The B-book: assigning programs to meanings. Cambridge Univ Pr (1996)
17. Börger, E., Stärk, R.: Abstract state machines: a method for high-level system design and analysis. Springer Verlag (2003)
18. Krishnamurthi, S., Fisler, K., Dougherty, D., Yoo, D.: Alchemy: transmuting base alloy specifications into implementations. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, ACM New York, NY, USA (2008) 158–169
19. Giannakopoulos, T., Dougherty, D., Fisler, K., Krishnamurthi, S.: Towards an Operational Semantics for Alloy. In: Proceedings of the 16th International Symposium on Formal Methods. To appear. (2009)
20. Beyer, D.: Relational programming with CrocoPat. In: Proceedings of the 28th International Conference on Software engineering, ACM New York, NY, USA (2006) 807–810
21. Behnke, R., Berghammer, R., Meyer, E., Schneider, P.: RELVIEW-A system for calculating with relations and relational programming. Lecture Notes in Computer Science **1382** (1998) 318–321