DynAlloy: Upgrading Alloy with Actions

Marcelo F. Frias[*]
Juan P. Galeotti
Carlos G. López Pombo
Department of Computer Science
FCEyN
Universidad de Buenos Aires
Argentina
{mfrias, jgaleotti, clpombo}@dc.uba.ar

Nazareno M. Aguirre
Department of Computer Science
FCEFQyN - Universidad Nacional de Río Cuarto
Argentina
naguirre@dc.exa.unrc.edu.ar

ABSTRACT

We present DynAlloy, an extension to the Alloy specification language to describe dynamic properties of systems using actions. Actions allow us to appropriately specify dynamic properties, particularly, properties regarding execution traces, in the style of dynamic logic specifications.

We extend Alloy's syntax with a notation for partial correctness assertions, whose semantics relies on an adaptation of Dijkstra's weakest liberal precondition. These assertions, defined in terms of actions, allow us to easily express properties regarding executions, favoring the separation of concerns between the static and dynamic aspects of a system specification.

We also extend the Alloy tool in such a way that DynAlloy specifications are also automatically analyzable, as standard Alloy specifications. We present the foundations, two case-studies, and empirical results evidencing that the analysis of DynAlloy specifications can be performed efficiently.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.10 [**Software Engineering**]: Design—*Representation*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Design, Verification.

Keywords

Alloy, dynamic logic, software specification, software validation.

[*]and CONICET.

1. INTRODUCTION

Alloy [4, 6] is a formal specification language, which belongs to the class of the so-called model-oriented formal methods. Alloy is defined in terms of a simple relational semantics, its syntax includes constructs ubiquitous in object-oriented notations, and it features automated analysis capabilities [5]; these characteristics have made Alloy an appealing formal method.

Alloy has its roots in the Z specification language [8], and, as Z, is appropriate for describing structural properties of systems. However, in contrast with Z, Alloy has been designed with the goal of making specifications automatically analyzable.

Alloy's representations of systems are based on abstract models. These models are defined essentially in terms of *data domains*, and *operations* between these domains. In particular, one can use data domains to specify the state space of a system or a component, and employ operations as a means for the specification of state change. Semantically, operations correspond to *predicates*, in which certain variables are assumed to be *output* variables, or, more precisely, are meant to describe the system state *after* the operation is executed. By looking into Alloy's semantics, it is easy to verify that "output" and "after" are *intentional concepts*, i.e., the notions of output or temporal precedence are not reflected in the semantics and, therefore, understanding variables this way is just a (reasonable) convention. Variable naming conventions are a useful mechanism, which might lead to a simpler semantics of specifications. However, as we advocate in this paper, the inclusion of *actions* (understood as a general concept associated with state change, covering transactions and events, for example), with a well defined input/output semantics, in order to specify properties of executions, might provide a significant improvement to Alloy's expressiveness and analyzability. Moreover, actions enable us to characterise properties regarding execution traces in a convenient way.

In order to see how actions might improve Alloy's expressiveness, suppose, for instance, that we need to define the combination of certain operations describing a system. Some combinations are representable in Alloy; for instance, if we have two operations $Oper_1$ and $Oper_2$, and denote by $Oper_1 ; Oper_2$ and $Oper_1 + Oper_2$ the sequential composition and nondeterministic choice of these operations, respectively, then these can be easily defined in Alloy as follows:

$$\begin{aligned}&\mathit{Oper}_1 ; \mathit{Oper}_2(x, y) = \\ &\qquad\qquad \text{some } z \mid (\mathit{Oper}_1(x, z) \text{ and } \mathit{Oper}_2(z, y)),\end{aligned}$$

$$\mathit{Oper}_1 + \mathit{Oper}_2(x, y) = \mathit{Oper}_1(x, y) \text{ or } \mathit{Oper}_2(x, y)\ .$$

However, if we aim at specifying properties of executions, then it is reasonable to think that we will need to predicate at least about all terminating executions of the system. This demands some kind of iteration of operations. While it is possible to define sequential composition or nondeterministic choice, as we showed before, finite (unbounded) iteration of operations cannot be defined in Alloy.

Nevertheless, some effort has been put toward representing the iteration of operations, in order to analyze properties of executions in Alloy. By enriching models with the inclusion of a new signature (type) for execution traces [6], and constraints that indicate how these traces are constructed from the operations of the system, it is possible to *simulate* operation iteration. Essentially, traces are defined as being composed of all intermediate states visited along specific runs. While adding traces to specifications provides indeed a mechanism for dealing with executions (and even specifications involving execution traces can be automatically analyzed), this approach requires the specifier to explicitly take care of the definition of traces (an *ad hoc* task which depends on the properties of traces one wants to validate). Furthermore, the resulting specifications are cumbersome, since they mix together two clearly separated aspects of systems, the *static* definition of domains and operations that constitute the system, and the *dynamic* specification of traces of executions of these operations. Modules might help in organizing a specification, by separating the static and dynamic aspects of a system; however, the specifier still needs to manually provide the specification of traces, since, as we said, this is an *ad hoc* activity, dependent on the particular property of executions that needs to be validated.

We consider that actions, if appropriately used, constitute a better candidate for specifying assertions regarding the dynamics of a system (i.e., assertions regarding execution traces), leading to cleaner specifications, with clearer separation of concerns.

In order to compare these two approaches, let us suppose that we need to specify that every terminating arbitrary execution of two operations Oper_1 and Oper_2 beginning in a state satisfying a formula α terminates in a state satisfying a formula β. Using the approach presented in [6], it is necessary to provide an explicit specification of execution traces complementing the specification of the system, as follows:

1. specify the initial state as a state satisfying α,
2. specify that every pair of consecutive states in a trace is either related by Oper_1 or by Oper_2,
3. specify that the final state satisfies β.

Using the approach we propose, based on actions, execution traces are only implicitly used. The above specification can be written in a simple and elegant way, as follows:

$$\begin{gathered}\{\alpha\} \\ (\mathit{Oper}_1 + \mathit{Oper}_2)^* \\ \{\beta\}\end{gathered}$$

This states, as we required, that every terminating execution of $(\mathit{Oper}_1 + \mathit{Oper}_2)^*$ (which represents an unbounded iteration of the nondeterministic choice between Oper_1 and Oper_2) starting in a state satisfying α, ends up in a state satisfying β. This notation corresponds to the traditional and well-known notation for partial correctness assertions. Notice that no explicit reference to traces is required. Nevertheless, traces exist and are well taken care of in the semantics of actions, far from the eyes of the software engineer writing a model. It seems clear then that pursuing our task of adding actions to Alloy might indeed contribute toward the usability of the language. Note that finite unbounded iteration is, in our approach, expressible via the iteration operation "*".

As we mentioned, one of the main features of Alloy is its analyzability. The Alloy tool allows us to automatically analyze specifications by searching for counterexamples of assertions with the help of the off-the-shelf SAT solvers MChaff, ZChaff [7] and Berkmin [2]. Therefore, extending the language with actions, while still an interesting intellectual puzzle, is not important if it cannot be complemented with efficient automatic analysis. So, we modify the Alloy tool in order to deal with the analysis of Alloy specifications involving actions and execution traces assertions. Notice that, even though finite unbounded iteration is expressible in DynAlloy, a bound on the depth of the iterations needs to be imposed for the analysis tasks. So, for SAT solving based analysis, our extension only covers bounded iteration.

The contributions of this paper are then summarized as follows.

- We add to Alloy the possibility of defining actions and asserting properties using partial correctness assertions, as a mechanism for the specification of operations. We refer to this extension of Alloy as DynAlloy.
- We present a modification of the Alloy tool in order to allow for an efficient verification of DynAlloy specifications.
- We present two case-studies for which we compare the analysis running time when using actions and traces. We conclude that efficiency increases when using actions and partial correctness assertions.

The remainder of this paper is organized as follows. In Section 2 we present the Alloy specification language. In Section 3 we present the DynAlloy language. In Section 4 we present the DynAlloy tool. In Section 5 we present the case-studies and their corresponding running times. Finally, in Section 6 we present our conclusions and proposals for further work.

2. THE ALLOY SPECIFICATION LANGUAGE

In this section, we introduce the reader to the Alloy specification language by means of an example extracted from [6]. This example serves as a means for illustrating the standard features of the language and their associated semantics, and will also help us demonstrate the shortcomings we wish to overcome.

Suppose we want to specify systems involving memories with cache. We might recognize that, in order to specify memories, data types for data and addresses are especially necessary. We can then start by indicating the existence of disjoint sets (of atoms) for data and addresses, which in Alloy are specified using *signatures*:

```
sig Addr { }        sig Data { }
```

These are basic signatures. We do not assume any special properties regarding the structures of data and addresses.

With data and addresses already defined, we can now specify what constitutes a memory. A possible way of defining memories is by saying that a memory consists of set of addresses, and a (total) mapping from these addresses to data values:

```
sig Memory {
    addrs: set Addr
    map: addrs ->! Data
}
```

The symbol "!" in the above definition indicates that "map" is functional and total (for each element a of addrs, there exists exactly one element d in *Data* such that map$(a) = d$).

Alloy allows for the definition of signatures as subsets of the set denoted by another "parent" signature. This is done via what is called *signature extension*. For the example, one could define other (perhaps more complex) kinds of memories as extensions of the *Memory* signature:

```
sig MainMemory extends Memory {}

sig Cache extends Memory {
    dirty: set addrs
}
```

As specified in these definitions, *MainMemory* and *Cache* are special kinds of memories. In caches, a subset of addrs is recognized as *dirty*.

A system might now be defined to be composed of a main memory and a cache:

```
sig System {
    cache: Cache
    main: MainMemory
}
```

As the previous definitions show, signatures are used to define data domains and their structure. The attributes of a signature denote *relations*. For instance, the "addrs" attribute in signature *Memory* represents a binary relation, from memory atoms to sets of atoms from *Addr*. Given a set m (not necessarily a singleton) of *Memory* atoms, m.addrs denotes the relational image of m under the relation denoted by addrs. This leads to a relational view of the dot notation, which is simple and elegant, and preserves the intuitive navigational reading of dot, as in object orientation. Signature extension, as we mentioned before, is interpreted as inclusion of the set of atoms of the extending signature into the set of atoms of the extended signature.

In Fig. 1, we present the grammar and semantics of Alloy's relational logic, the core logic on top of which all of Alloy's syntax and semantics are defined. An important difference with respect to previous versions of Alloy, as the one presented in [4], is that expressions now range over relations of arbitrary rank, instead of being restricted to binary relations. Composition of binary relations is well understood; but for relations of higher rank, the following definition for the composition of relations has to be considered:

$$R;S = \{\langle a_1, \ldots, a_{i-1}, b_2, \ldots, b_j\rangle : \exists b\,(\langle a_1, \ldots, a_{i-1}, b\rangle \in R \ \wedge\ \langle b, b_2, \ldots, b_j\rangle \in S)\}\ .$$

Operations for transitive closure and transposition are only defined for binary relations. Thus, function X in Fig. 1 is partial.

2.1 Operations in a Model

So far, we have just shown how the structure of data domains can be specified in Alloy. Of course, one would like to be able to define operations over the defined domains. Following the style of Z specifications, operations in Alloy can be defined as expressions, relating states from the state spaces described by the signature definitions. Primed variables are used to denote the resulting values, although this is just a convention, not reflected in the semantics.

In order to illustrate the definition of operations in Alloy, consider, for instance, an operation that specifies the writing of a value to an address in a memory:

```
Write(m, m': Memory, d: Data, a: Addr) {
    m'.map = m.map ++ (a -> d)                (1)
}
```

The intended meaning of this definition can be easily understood, having in mind that m' is meant to denote the memory (or memory state) resulting of the application of function Write, a -> d denotes the ordered pair $\langle$a, d$\rangle$, and ++ denotes relational overriding, defined as follows[1]:

$$R{+}{+}S = \{\,\langle a_1, \ldots, a_n\rangle : \langle a_1, \ldots, a_n\rangle \in R \ \wedge\ a_1 \notin \mathsf{dom}\,(S)\,\} \cup S\ .$$

We have already seen a number of constructs available in Alloy, such as the dot notation and signature extension, that resemble object oriented definitions. Operations, however, represented by functions in Alloy, are not "attached" to signature definitions, as in traditional object-oriented approaches. Instead, functions describe operations of the whole set of signatures, i.e., the model. So, there is no notion similar to that of class, as a mechanism for encapsulating data (attributes or fields) and behavior (operations or methods).

In order to illustrate a couple of further points, consider the following more complex function definition:

```
fun SysWrite(s, s': System, d: Data, a: Addr) {
    Write(s.cache, s'.cache, d, a)
    s'.cache.dirty = s.cache.dirty + a
    s'.main = s.main
}
```

There are two important issues exhibited in this function definition. First, function SysWrite is defined in terms of the more primitive Write. Second, the use of Write takes advantage of the *hierarchy* defined by signature extension: note that function Write was defined for memories, and in SysWrite it is being "applied" to cache memories.

As explained in [6], an operation that *flushes* lines from a cache to the corresponding memory is necessary in order to have a realistic model of memories with cache, since usually

[1]Given a n-ary relation R, $\mathsf{dom}\,(R)$ denotes the set $\{\,a_1 : \exists a_2, \ldots, a_n$ such that $\langle a_1, a_2, \ldots, a_n\rangle \in R\,\}$.

$\mathit{problem} ::= \text{decl}^{*}\text{form}$
$\text{decl} ::= \mathit{var} : \mathit{typexpr}$
$\mathit{typexpr} ::=$
type
$\mid \mathit{type} \rightarrow \mathit{type}$
$\mid \mathit{type} \Rightarrow \mathit{typexpr}$

$\text{form} ::=$
$\text{expr } \mathit{in} \text{ expr (subset)}$
$\mid !\text{form (neg)}$
$\mid \text{form \&\& form (conj)}$
$\mid \text{form } || \text{ form (disj)}$
$\mid \mathit{all}\ v : \mathit{type}/\text{form (univ)}$
$\mid \mathit{some}\ v : \mathit{type}/\text{form (exist)}$

$\text{expr} ::=$
$\text{expr} + \text{expr (union)}$
$\mid \text{expr \& expr (intersection)}$
$\mid \text{expr} - \text{expr (difference)}$
$\mid \sim \text{expr (transpose)}$
$\mid \text{expr.expr (navigation)}$
$\mid +\text{expr (transitive closure)}$
$\mid \{v : t/\text{form}\} \text{ (set former)}$
$\mid Var$

$Var ::=$
$\mathit{var} \text{ (variable)}$
$\mid Var[\mathit{var}] \text{ (application)}$

$M : \text{form} \rightarrow \mathit{env} \rightarrow \mathit{Boolean}$
$X : \text{expr} \rightarrow \mathit{env} \rightarrow \mathit{value}$
$\mathit{env} = (\mathit{var} + \mathit{type}) \rightarrow \mathit{value}$
$\mathit{value} = (\mathit{atom} \times \cdots \times \mathit{atom}) +$
$\quad (\mathit{atom} \rightarrow \mathit{value})$

$M[a\ \mathit{in}\ b]e = X[a]e \subseteq X[b]e$
$M[!F]e = \neg M[F]e$
$M[F \&\& G]e = M[F]e \wedge M[G]e$
$M[F\ ||\ G]e = M[F]e \vee M[G]e$
$M[\mathit{all}\ v : t/F] =$
$\quad \bigwedge\{M[F](e \oplus v \mapsto \{\, x \,\})/x \in e(t)\}$
$M[\mathit{some}\ v : t/F] =$
$\quad \bigvee\{M[F](e \oplus v \mapsto \{\, x \,\})/x \in e(t)\}$

$X[a + b]e = X[a]e \cup X[b]e$
$X[a \& b]e = X[a]e \cap X[b]e$
$X[a - b]e = X[a]e \setminus X[b]e$
$X[\sim a]e = \{\, \langle x, y\rangle : \langle y, x\rangle \in X[a]e \,\}$
$X[a.b]e = X[a]e ; X[b]e$
$X[+a]e = \text{the smallest } r \text{ such that}$
$\quad r ; r \subseteq r \text{ and } X[a]e \subseteq r$
$X[\{v : t/F\}]e =$
$\quad \{x \in e(t)/M[F](e \oplus v \mapsto \{\, x \,\})\}$
$X[v]e = e(v)$
$X[a[v]]e = \{\langle y_1, \ldots, y_n\rangle /$
$\quad \exists x. \langle x, y_1, \ldots, y_n\rangle \in e(a) \wedge \langle x\rangle \in e(v)\}$

Figure 1: Grammar and semantics of Alloy

caches are smaller than main memories. A (nondeterministic) operation that flushes information from the cache to main memory can be specified in the following way:

```
fun Flush(s, s': System) {
    some x: set s.cache.addrs {
        s'.cache.map = s.cache.map - { x->Data }
        s'.cache.dirty = s.cache.dirty - x
        s'.main.map = s.main.map ++
          {a: x, d: Data | d = s.cache.map[a]}
    }
}
```

In the third line of the above definition of function Flush, x->Data denotes all the ordered pairs whose domains fall into the set x, and that range over the domain Data.

Functions can also be used to represent *special* states. For instance, we can characterize the states in which the cache lines not marked as dirty are consistent with main memory:

```
fun DirtyInv(s: System) {
    all a : !s.cache.dirty |                        (2)
              s.cache.map[a] = s.main.map[a] }
```

In this context, the symbol "!" denotes negation, indicating in the above formula that "a" ranges over atoms that are non dirty addresses.

2.2 Properties of a Model

As the reader might expect, a model can be enhanced by adding properties (axioms) to it. These properties are written as logical formulas, much in the style of the Object Constraint Language (OCL). Properties or constraints in Alloy are defined as *facts*. To give an idea of how constraints or properties are specified, we reproduce some here. It might be necessary to say that the sets of main memories and cache memories are disjoint:

fact {no (*MainMemory* & *Cache*)}

In the above expression, "no x" indicates that x has no elements, and & denotes intersection. Another important constraint inherent to the presented model is that, in every system, the addresses of its cache are a subset of the addresses of its main memory:

fact {all s: System | s.cache.addrs in s.main.addrs}

More complex facts can be expressed by using the quite considerable expressive power of the relational logic.

2.3 Assertions

Assertions are the *intended* properties of a given model. Consider, for instance, the following simple Alloy assertion, regarding the presented example:

```
assert {
    all s: System | DirtyInv(s) && no s.cache.dirty
       => s.cache.map in s.main.map
}
```

This assertion states that, if "DirtyInv" holds in system "s" and there are no dirty addresses in the cache, then the cache agrees in all its addresses with the main memory.

Assertions are used to check specifications. Using the Alloy analyzer, it is possible to validate assertions, by searching for possible (finite) counterexamples for them, under the constraints imposed in the specification of the system.

3. DYNALLOY: ADDING PARTIAL CORRECTNESS ASSERTIONS TO ALLOY

In this section we extend Alloy's relational logic syntax and semantics with the aim of dealing with properties of executions of operations specified in Alloy. It will follow that DynAlloy extends Alloy and its relational logic.

The reason for this extension is that we want to provide a setting in which, besides functions describing sets of states, actions are made available, to represent state changes (i.e., to describe relations between input and output data). As opposed to the use of functions for this purpose, actions have an input/output meaning reflected in the semantics, and can be composed to form more complex actions, using well-known constructs from imperative programming languages.

The syntax and semantics of DynAlloy is described in Section 3.1. It is worth mentioning at this point that both were strongly motivated by dynamic logic [3], and the suitability of dynamic logic for expressing partial correctness assertions.

3.1 Functions vs. Actions

Functions in Alloy are just parameterized formulas. Some of the parameters are considered input parameters, and the relationship between input and output parameters relies on the convention that the second argument is the result of the function application. Recalling the definition of function Write, notice that there is no actual change in the state of the system, since no variable actually changes its value.

Dynamic logic [3] arose in the early '70s, with the intention of faithfully reflecting state change. Motivated by dynamic logic, we propose the use of actions to model state change in Alloy, as described below.

What we would like to say about an action is how it transforms the system state after its execution. A (now) traditional way of doing so is by using pre and post condition assertions. An assertion of the form

$$\begin{array}{c}\{\alpha\}\\ A\\ \{\beta\}\end{array}$$

affirms that whenever action A is executed on a state satisfying α, if it terminates, it does so in a state satisfying β. This approach is particularly appropriate, since behaviors described by functions are better viewed as the result of performing an action on an input state. Thus, the definition of function Write could be expressed as an action definition, of the following form:

$$\begin{array}{c}\{\mathit{true}\}\\ \mathit{Write}(\mathrm{m} : \mathit{Memory}, \mathrm{d} : \mathit{Data}, \mathrm{a} : \mathit{Addr})\\ \{\mathrm{m}'.\mathrm{map} = \mathrm{m.map} \mathbin{++} (\mathrm{a} \;\rightarrow\; \mathrm{d})\}\ .\end{array} \tag{3}$$

At first glance it is difficult to see the differences between (1) and (3), since both formulas seem to provide the same information. The crucial differences are reflected in the semantics, as well as in the fact that actions can be sequentially composed, iterated or composed by nondeterministic choice, while Alloy functions, in principle, cannot.

An immediately apparent difference between (1) and (3) is that action *Write* does not involve the parameter m$'$, while function Write uses it. This is so because we use the convention that m$'$ denotes the state of variable m *after* execution of action *Write*. This time, "*after*" means that m$'$ gets its value in an environment reachable through the execution of action *Write* (cf. Fig. 3). Since *Write* denotes a binary relation on the set of environments, there is a precise notion of input/output inducing a before/after relationship.

3.2 Syntax and Semantics of DynAlloy

The syntax of DynAlloy's formulas extends the one presented in Fig. 1 with the addition of the following clause for building partial correctness statements:

$$\mathit{formula} \;::=\; \ldots \mid \{\mathit{formula}\}\ \mathit{program}\ \{\mathit{formula}\} \quad \text{“partial correctness”}$$

The syntax for programs (cf. Fig. 2) is the class of regular programs defined in [3], plus a new rule to allow for the construction of atomic actions from their pre and post conditions. In the definition of atomic actions, $\overline{x}$ denotes a sequence of formal parameters. Thus, it is to be expected that the precondition is a formula whose free variables are within $\overline{x}$, while postcondition variables might also include primed versions of the formal parameters.

In Fig. 3 we extend the definition of function M to partial correctness assertions and define the denotational semantics of programs as binary relations over *env*. The definition of function M on a partial correctness assertion makes clear that we are actually considering a partial correctness semantics. This follows from the fact that we are not requesting environment e to belong to the domain of the relation $P[p]$. In order to provide semantics for atomic actions, we will assume that there is a function A assigning, to each atomic action, a binary relation on the environments. We define function A as follows:

$$A(\langle \mathit{pre}, \mathit{post}\rangle) = \left\{ \left\langle e, e' \right\rangle : M[\mathit{pre}]e \wedge M[\mathit{post}]e' \right\}\ .$$

There is a subtle point in the definition of the semantics of atomic programs. While actions may modify the value of all variables, we assume that those variables whose primed versions do not occur in the post condition retain their input value. Thus, the atomic action *Write* modifies the value of variable m, but a and d keep their initial values. This allows us to use simpler formulas in pre and post conditions.

$$\begin{array}{l}M[\{\alpha\}p\{\beta\}]e = \\ \qquad M[\alpha]e \implies \forall e' \left(\left\langle e, e' \right\rangle \in P[p] \implies M[\beta]e'\right)\\ \\ \qquad P : \mathit{program} \rightarrow \mathcal{P}\left(\mathit{env} \times \mathit{env}\right)\\ \qquad P[\langle \mathit{pre}, \mathit{post}\rangle] = A(\langle \mathit{pre}, \mathit{post}\rangle)\\ \qquad P[\alpha?] = \{ \langle e, e' \rangle : M[\alpha]e \wedge e = e' \}\\ \qquad P[p_1 + p_2] = P[p_1] \cup P[p_2]\\ \qquad P[p_1 ; p_2] = P[p_1] ; P[p_2]\\ \qquad P[p^*] = P[p]^*\end{array}$$

Figure 3: Semantics of DynAlloy.

3.3 Specifying Properties of Executions in Alloy and DynAlloy

Suppose we want to specify that a given property P is invariant under sequences of applications of the operations "Flush" and "SysWrite", from certain initial state. A technique useful for stating the invariance of a property P consists of specifying that P holds in the initial states, and

$$
\begin{array}{lcll}
program & ::= & \langle formula, formula\rangle(\overline{x}) & \text{“atomic action”} \\
 & | & formula? & \text{“test”} \\
 & | & program + program & \text{“non-deterministic choice”} \\
 & | & program;program & \text{“sequential composition”} \\
 & | & program^* & \text{“iteration”}
\end{array}
$$

Figure 2: Grammar for composite actions in DynAlloy

that for every non initial state and every operation $O \in \{Flush, SysWrite\}$, the following holds:

$$P(s) \wedge O(s, s') \;\Rightarrow\; P(s')\ .$$

This specification is sound but incomplete, since the invariance may be violated in unreachable states. Of course it would be desirable to have a specification in which the states under consideration were exactly the reachable ones. This motivated the introduction of *traces* in Alloy [6].

The following example, extracted from [6], shows signatures for clock ticks and for traces of states. The first exclamation mark in the definition of “next”' means that this relation is total on its declared domain.

```
    sig Tick {}

    sig SystemTrace {
        ticks: set Tick,
        first, last: Tick,
        next: (ticks - last) ! → ! (ticks - first),
        state: ticks → ! System }
```

The following “fact” states that all ticks in a trace are reachable from the first tick, that a property called “Init” holds in the first state, and that the passage from one state to the next is through the application of one of the operations under consideration:

```
fact {
    first.next* = ticks
    Init(first.state)
    all t: ticks - last   |
      some s = t.state, s’ = t.next.state   |
        Flush (s,s’) ||
         some d : Data, a : Addr | SysWrite(s,s’,d,a)
}
```

If we now want to assert that P is invariant, it suffices to assert that P holds in the final state of every trace. Notice that unreachable states are no longer a burden because all states in a trace are reachable from the states that occurred before.

Even though, from a formal point of view, the use of traces is correct, from a modeling perspective it is less suitable. Traces are introduced in order to cope with the lack of real state change in Alloy. They allow us to port the primed variables used in single operations to sequences of applications of operations.

The specification of actions SysWrite and Flush in DynAlloy is done as follows:

```
{ true }

  SysWriteDA(s: System)

{ some d: Data, a: Addr |
  s’.cache = s.cache ++ (a → d) ∧
  s’.cache.dirty = s.cache.dirty + a ∧
  s’.main = s.main }


{ true }

  FlushDA(s: System)

{ some x: set s.cache.addrs |
  s’.cache.map = s.cache.map - x→Data ∧
  s’.cache.dirty = s.cache.dirty - x  ∧
  s’.main.map = s.main.map ++
     {a: x, d: Data | d = s.cache.map[a]} }
```

Notice that the previous specifications are as understandable as the ones given in Alloy. Moreover, by using partial correctness statements on the set of regular programs generated by the set of atomic actions { SysWriteDA, FlushDA }, we can assert the invariance of a property P under finite applications of functions SysWrite and Flush in a simple and elegant way, as follows:

$$\begin{gathered}\{Init(s) \wedge P(s)\} \\ (\text{SysWriteDA}(s) + \text{FlushDA}(s))^* \\ \{P(s')\}\end{gathered}$$

More generally, suppose now that we want to show that a property Q is invariant under sequences of applications of arbitrary operations $O_1, \ldots, O_k$, starting from states s described by a formula *Init*. The specification of this assertion in our setting is done via the following formula:

$$\begin{gathered}\{Init(\overline{x}) \wedge Q(\overline{x})\} \\ (O_1(\overline{x}) + \cdots + O_k(\overline{x}))^* \\ \{Q(\overline{x'})\}\end{gathered} \tag{4}$$

Notice that there is no need to mention traces in the specification of the previous properties. This is because finite traces get determined by the semantics of reflexive-transitive closure.

3.4 Analysis of DynAlloy Specifications

Alloy’s design was deeply influenced by the intention of producing an automatically analyzable language. While DynAlloy is, to our understanding, better suited than Alloy for the specification of properties of executions, the use of ticks and traces as defined in [6] has as an advantage that it allows one to automatically analyze properties of executions.

Therefore, an almost mandatory question is whether DynAlloy specifications can be automatically analyzed, and if so, how efficiently. The main rationale behind our technique is the translation of partial correctness assertions to first-order Alloy formulas, using weakest liberal preconditions [1]. The generated Alloy formulas, which may be large and quite difficult to understand, are not visible to the end user, who only accesses the declarative DynAlloy specification.

We define below a function

$$wlp : program \times formula \to formula$$

that computes the weakest liberal precondition of a formula according to a program (composite action). We will in general use names $x_1, x_2 \ldots$ for program variables, and will use names $x'_1, x'_2, \ldots$ for the value of program variables *after* action execution. We will denote by $\alpha|_x^v$ the substitution of all free occurrences of variable x by the fresh variable v in formula α.

When an atomic action a specified as $\langle pre, post\rangle(\overline{x})$ is used in a composite action, formal parameters are substituted by actual parameters. Since we assume all variables are input/output variables, actual parameters are variables, let us say, $\overline{y}$. In this situation, function wlp is defined as follows:

$$wlp[a(\overline{y}), f] = \\ pre|_{\overline{x}}^{\overline{y'}} \implies \text{all } \overline{n} \left(post|_{\overline{x'}}^{\overline{n}}|_{\overline{x}}^{\overline{y'}} \implies f|_{\overline{y'}}^{\overline{n}} \right) . \quad (5)$$

A few points need to be explained about (5). First, we assume that free variables in f are amongst $\overline{y'}$, $\overline{x_0}$. Variables in $\overline{x_0}$ are generated by translation $pcat$ given in (7). Second, $\overline{n}$ is an array of new variables, one for each variable modified by the action. Last, notice that the resulting formula has again its free variables amongst $\overline{y'}$, $\overline{x_0}$. This is also preserved in the remaining cases in the definition of function wlp.

For the remaining action constructs, the definition of function wlp is the following:

$$\begin{array}{lcl} wlp[g?, f] & = & g \implies f \\ wlp[p_1 + p_2, f] & = & wlp[p_1, f] \wedge wlp[p_2, f] \\ wlp[p_1 ; p_2, f] & = & wlp[p_1, wlp[p_2, f]] \\ wlp[p^*, f] & = & \bigwedge_{i=0}^{\infty} wlp[p^i, f] . \end{array}$$

Notice that wlp yields Alloy formulas in all these cases, except for the iteration construct, where the resulting formula may be infinitary. In order to obtain an Alloy formula, we can impose a bound on the depth of iterations. This is equivalent to fixing a maximum length for traces. A function $Bwlp$ (bounded weakest liberal precondition) is then defined exactly as wlp, except for iteration, where it is defined by:

$$Bwlp[p^*, f] = \bigwedge_{i=0}^{n} Bwlp[p^i, f] . \quad (6)$$

In (6), n is the scope set for the depth of iteration.

We now define a function $pcat$ that translates partial correctness assertions to Alloy formulas. For a partial correctness assertion $\{\alpha(\overline{y})\}\ P(\overline{y})\ \{\beta(\overline{y}, \overline{y'})\}$

$$pcat(\{\alpha\}\ P\ \{\beta\}) = \\ \forall \overline{y} \left(\alpha \implies \left(Bwlp\left[p, \beta|_{\overline{y}}^{\overline{x_0}}\right]\right)|_{\overline{y'}}^{\overline{y}}|_{\overline{x_0}}^{\overline{y}} \right) . \quad (7)$$

Of course this analysis method where iteration is restricted to a fixed depth is not complete, but clearly it is not meant to be; from the very beginning we placed restrictions on the size of domains involved in the specification to be able to turn first-order formulas into propositional formulas. This is just another step in the same direction.

4. THE DYNALLOY TOOL

The Alloy tool [5] is open source. This contributes greatly toward developing extensions of the tool. DynAlloy is an extension of the Alloy tool that allows the user to write and analyze specifications involving actions. Once a DynAlloy specification (involving actions) is opened, executing the Build command first translates the DynAlloy specification to Alloy using function $pcat$, and then compiles the Alloy specification thus obtained.

In this section we discuss some modifications on the definition of function $pcat$ provided in (7) that will allow us to analyze specifications efficiently. We also describe some implementation details.

4.1 Translating Partial Correctness Assertions to Alloy

In Section 3.4 we showed how to compute the weakest liberal precondition wlp and its bounded version $Bwlp$ for an arbitrary composite action. For atomic actions a and b, $Bwlp(a;b,\alpha)$ is a formula whose shape is roughly

$$pre_a(s) \Rightarrow \forall s_1 (post_a(s, s_1) \Rightarrow \\ (pre_b(s_1) \Rightarrow \forall s_2 (post_b(s_1, s_2) \Rightarrow \alpha(s_2)))) . \quad (8)$$

When Alloy was fed with a formula like (8) for three actions, two problems arose:

1. Compilation time was almost unacceptable.
2. Analysis time was in general worse than the time obtained using traces.

Notice that the quantifiers binding variables s_1 and s_2 can be promoted to the front of the formula by simple logical manipulations, yielding

$$\forall s_1 \forall s_2 (pre_a(s) \Rightarrow (post_a(s, s_1) \Rightarrow \\ (pre_b(s_1) \Rightarrow (post_b(s_1, s_2) \Rightarrow \alpha(s_2))))) . \quad (9)$$

Feeding Alloy with a formula like (9) produced running times that were, in general, significantly better than those achieved in Alloy using traces. On the negative side, for an action of the form $(a_1 + a_2)^n$, the resulting formula was considerably large. For $n = 2$, using the definition of $Bwlp$, we obtain:

$$\begin{aligned} & Bwlp\left((a+b)^2, \alpha\right) \\ & \quad = Bwlp\left((a+b);(a+b), \alpha\right) \\ & \quad = Bwlp\left(a+b, Bwlp\left(a+b, \alpha\right)\right) \\ & \quad = Bwlp\left(a, Bwlp\left(a+b, \alpha\right)\right) \\ & \qquad \wedge Bwlp\left(b, Bwlp\left(a+b, \alpha\right)\right) \\ & \quad = pre_a \Rightarrow \left(post_a \Rightarrow Bwlp\left(a+b, \alpha\right)\right) \\ & \qquad \wedge pre_b \Rightarrow \left(post_b \Rightarrow Bwlp\left(a+b, \alpha\right)\right) . \end{aligned} \quad (10)$$

Simple logical properties allow us to rewrite (10) as

$$(pre_a \wedge post_a) \Rightarrow Bwlp\left(a+b, \alpha\right) \\ \wedge \quad (pre_b \wedge post_b) \Rightarrow Bwlp\left(a+b, \alpha\right) . \quad (11)$$

At this point, notice that the formula $Bwlp(a+b, \alpha)$ appears twice in (11). Thus computing $Bwlp\left((a+b)^n, \alpha\right)$ yields a formula whose size is exponential as a function of n. Feeding Alloy with a formula like (11) produced, for small values of n, analysis times that were significantly better that those achieved using traces. Unfortunately, compilation time grew exponentially, proving that analysis using this translation was unfeasible for reasonable values of n.

Once again, elementary properties of first-order logic allow us to transform (11) to the equivalent formula

$$((pre_a \wedge post_a) \vee (pre_b \wedge post_b)) \Rightarrow Bwlp\,(a+b, \alpha)\,. \quad (12)$$

Formula $Bwlp\,(a+b, \alpha)$ occurs only once in (12). Applying this simple transformation made the previous exponential-size formulaes become linear-size. This is the translation that is implemented in the DynAlloy tool.

Running times very much depend on the chosen SAT solver, and while our translation works well in all of them, different optimizations can be applied depending on the particular SAT solver chosen.

4.2 Implementation Details

Not only the source code for Alloy is publicly available. All the necessary software and tools required in order to generate the source code are freely available, too. For instance, the Alloy grammar specification, as required by JavaCC (a parser generator for Java), is also supplied. We extended this grammar specification to a specification of DynAlloy's grammar. Combining the use of the tools JJTree and JavaCC, we built a parser and abstract syntax tree generator for DynAlloy. Given a tree for a DynAlloy model, we apply transformations leading to an Alloy specification.

In order to make this process invisible to the end user, we modified distribution 2.0 of the Alloy Analyzer.We changed the original Alloy Build command so that it now first translates a DynAlloy specification to Alloy, and then compiles the resulting model, in the way standard Alloy does.

5. CASE-STUDIES

In this section we analyze two case-studies. The first one is an assertion whose validity follows from the specification, and, therefore, has no counterexamples. It will serve us as a stress test for Alloy and DynAlloy. The second assertion has counterexamples, and is useful for verifying how efficiently can these be found using DynAlloy. The analysis was carried out using a Sun Sunblade 2000, with two 1GHz processors, and 2 GB of RAM. For the analysis we will impose a limit of 60'. Those runs that did not finish within 60' were stopped and marked in the tables as "> 60′".

5.1 Case-Study 1: DirtyInv

The problem we will analyze is whether function DirtyInv, defined in (2), is an invariant with respect to finite applications of operations SysWrite and Flush. Its Alloy specification is the following:

```
assert DirtyInvAssertionAlloy {
all tr: SystemTrace |
    DirtyInv(tr.state[tr.first]) =>
        DirtyInv (tr.state[tr.last])}
```

The corresponding DynAlloy specification is:

```
assert DirtyInvAssertionDynAlloy {
   {DirtyInv(s)}
      (SysWriteDA(s) + FlushDA(s))*
   {DirtyInv(s')}
}
```

Notice that these specifications are quite similar, in the sense that both predicate *only* about the initial and final states. In Tables 1–3 we compare running CPU times for the analysis of both specifications for different trace lengths, domain sizes, and the available SAT solvers.

The "check" condition used in the Alloy specification for traces of length n and domains of size k, is:

```
check DirtyInvAssertionAlloy for k but n+1
Tick, 1 SystemTrace.
```

For the DynAlloy specification, we use:

```
check DirtyInvAssertionDynAlloy for k.
```

Tr. length →	≤ 2		≤ 3		≤ 4	
# elems ↓	Alloy	DAlloy	Alloy	DAlloy	Alloy	DAlloy
3	0′01″	0′01″	0′04″	**0′02″**	0′20″	**0′07″**
4	0′13″	**0′01″**	3′04″	**0′11″**	45′25″	**1′48″**
5	1′40″	**0′03″**	34′40″	**0′59″**	> 60′	**31′17″**
6	5′52″	**0′06″**	> 60′	**2′24″**	> 60′	> 60′

Table 1: Verification time for the assertion *DirtyInv* using the SAT solver *MChaff*.

Tr. length →	≤ 2		≤ 3		≤ 4	
# elems ↓	Alloy	DAlloy	Alloy	DAlloy	Alloy	DAlloy
3	**0′02″**	0′06″	**0′07″**	2′17″	**0′24″**	31′53″
4	**0′16″**	0′18″	6′48″	**1′57″**	> 60′	> 60′
5	5′31″	**0′40″**	> 60′	**9′53″**	> 60′	> 60′
6	> 60′	**0′58″**	> 60′	> 60′	> 60′	> 60′

Table 2: Verification time for the assertion *DirtyInv* using the SAT solver *ZChaff*.

Tr. length →	≤ 2		≤ 3		≤ 4	
# elems ↓	Alloy	DAlloy	Alloy	DAlloy	Alloy	DAlloy
3	0′01″	0′01″	0′01′	0′01″	0′06″	**0′01″**
4	0′05″	**0′01″**	0′29″	**0′02″**	4′35″	**0′09″**
5	0′16″	**0′01″**	2′01″	**0′10″**	20′18″	**0′47″**
6	0′55″	**0′04″**	8′15″	**0′32″**	> 60′	**5′29″**

Table 3: Verification time for the assertion *DirtyInv* using the SAT solver *Berkmin*.

5.2 Case-Study 2: FreshDir

Given an initially empty CacheSystem whose set of addresses has size k, we will assert that every sequence of applications of the operations SysWrite and Flush still leaves a "fresh" address, that is, an address that has never been written into. This is a flawed assertion. In order to write the assertion, we require a function specifying that a CacheSystem is empty, and another describing the fresh address property. They are given next.

```
fun Init (s: System) {
    s.cache.dirty = none[s.cache.dirty]
    s.cache.map = none[s.cache.map]
    s.main.map = none[s.main.map]
}

fun FreshDir (s: System) {
    some a: Addr { all d: Data {
      ! ((a -> d) in s.main.map) &&
      ! ((a -> d) in s.cache.map) } }
}
```

Once Init and FreshDir were specified, we need to specify our assertion, both in Alloy and DynAlloy.

```
assert FreshDirAssertionAlloy {
all tr: SystemTrace |
Init(tr.state[tr.first]) =>
  FreshDir(tr.state[tr.last])
}

assert FreshDirAssertionDynAlloy {
   {Init(s)}
      (SysWriteDA(s) + FlushDA(s))*
   {FreshDir(s')}
}
```

In order to guarantee that there are n addresses, we check the assertion imposing a scope of n for signature Addr and include as part of the model a fact asserting that there are n distinct elements for this signature. So, we verify the Alloy assertion using the command

```
check FreshDirAssertionAlloy for 3 but n Addr,
n+1 Memory, n+1 System, n+1 Tick, 1 SystemTrace.
```

For DynAlloy we use

```
check FreshDirAssertionDynAlloy for 3 but n
Addr, n+1 Memory, n+1 System.
```

Table 4 shows a comparison of analysis running times for these assertions, under MChaff and Berkmin. ZChaff presented in general worse analysis times.

	MChaff		Berkmin	
Tr. length ↓	Alloy	DAlloy	Alloy	DAlloy
3	0′01″	0′01″	0′01″	0′01″
4	0′11″	**0′02″**	0′05″	**0′02″**
5	17′12″	**0′43″**	**0′05″**	0′19″
6	> 60′	**2′55″**	10′30″	**8′09″**
7	> 60′	**59′18″**	> 60′	> 60′

Table 4: Verification times for *FreshDir*.

6. CONCLUSIONS AND FURTHER WORK

We believe that using actions within Alloy in order to represent state change is a methodological improvement. Effectively, using actions favors a better separation of concerns, since models do not need to be reworked in order to describe the adequate notion of trace modeling the desired behavior. Using actions the problem reduces to describing how actions are to be composed. This methodological improvement is supported by empirical results evidencing that analysis can be done more efficiently than resorting to traces.

The shape of the formulas obtained during the translation of partial correctness assertions into Alloy gives us the opportunity of parallelizing their analysis process, allowing for the analysis of larger models.

Different SAT solvers react differently to the formulas resulting from the translation. While all of them behave satisfactorily, we can still generate different translations depending on the chosen SAT solver, in order to improve the analysis time.

Finally, a new version of Alloy (Alloy 3.0) has been made recently available. All our developments will be ported to this new version as soon as its source code is released.

7. REFERENCES

[1] E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag, 1990.

[2] E. Goldberg and Y. Novikov. BerkMin: A fast and robust sat-solver. In *Proceedings of the conference on Design, automation and test in Europe*, pages 142–149. IEEE Computer Society, 2002.

[3] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic logic*. Foundations of Computing. MIT Press, 2000.

[4] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 2002.

[5] D. Jackson. *A micromodels of software: Lightweight modelling and analysis with Alloy*. MIT Laboratory for Computer Science, Cambridge, MA, 2002.

[6] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proceedings of the 8th European software engineering conference held together with the 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 62–73, Vienna, Austria, 2001. Association for the Computer Machinery, ACM Press.

[7] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In J. Rabaey, editor, *Proceedings of the 38th conference on Design automation*, pages 530–535, Las Vegas, Nevada, United States, 2001. ACM Press.

[8] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.