Efficient Analysis of DynAlloy Specifications

MARCELO F. FRIAS, CARLOS G. LOPEZ POMBO, and JUAN P. GALEOTTI
Universidad de Buenos Aires
and
NAZARENO M. AGUIRRE
Universidad Nacional de Río Cuarto

DynAlloy is an extension of Alloy to support the definition of actions and the specification of assertions regarding execution traces. In this article we show how we can extend the Alloy tool so that DynAlloy specifications can be automatically analyzed in an efficient way. We also demonstrate that DynAlloy's semantics allows for a sound technique that we call *program atomization*, which improves the analyzability of properties regarding execution traces by considering certain programs as atomic steps in a trace.

We present the foundations, case studies, and empirical results indicating 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

Additional Key Words and Phrases: Alloy, dynamic logic, software specification, software validation

ACM Reference Format:

Frias, M. F., Lopez Pombo, C. G., Galeotti, J. P., and Aguirre, N. M. 2007. Efficient analysis of DynAlloy specifications. ACM Trans. Softw. Eng. Methodol. 17, 1, Article 4 (December 2007), 34 pages. DOI = 10.1145/1314493.1314497 http://doi.acm.org/10.1145/1314493.1314497

1. INTRODUCTION

The main objective of specifying, or modeling, software systems is the possibility of describing software artifacts with a certain degree of abstraction, so some useful analysis tasks can be performed on these descriptions. This analysis might allow one to see, or perhaps discover, some properties of the specified artifacts, and understand the implications of our design decisions. Furthermore, the analysis tasks can help us foresee problems and anticipate possible flaws

Authors' addresses: email: {mfrias, clpombo}@dc.uba.ar, jgaleotti@dc.uba.ar, naguirre@dc.exa.unrc.edu.ar.

of the specified artifact, which is especially important when this artifact still needs to be built.

Due to their precise semantics, formal specifications of software are well suited for analysis. However, formal semantics is not necessarily enough for a specification language to be useful: appropriate specification constructs at the right level of abstraction, ease of use, simplicity, tool support (for automated analysis, for instance), and so on, play decisive roles in the take-up and utility of a formal method. Abstraction, in particular, is normally considered a good feature of formalisms for specification, since abstract descriptions usually allow us to ignore irrelevant details, and concentrate on the issues we are interested in describing and analyzing. Abstract descriptions are also usually more concise, which greatly improves the comprehension of the models, and consequently, of the relevant aspects of the specified artifacts. Analyzability is an issue related to abstraction and simplicity, since simple, but adequately expressive, models (i.e., models at the right level of abstraction) are normally easier to analyze, and reduce the complexity of tool support implementation.

Alloy [Jackson 2002a; Jackson et al. 2001] is a formal specification language, defined in terms of a simple relational semantics. It belongs to the class of the so-called *model-oriented formal methods*, but unlike other model-oriented formalisms, Alloy has been designed with the particular aim of making specifications automatically analyzable [Jackson 2002b]. Moreover, Alloy's syntax, although small, includes constructs ubiquitous in (less formal) object-oriented notations. These features make the language easy to learn and use, and have turned Alloy into an appealing formal method.

As for any other model-oriented specification language, Alloy's approach to specification consists of describing systems by building *abstract* models of them. Traditionally, model-oriented specification languages describe a system by defining its state space, and its operations as state transformations; Alloy is not an exception in this respect. Alloy specifications are defined essentially in terms of *data domains*, and *operations* among 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. These characteristics make Alloy, and other model-oriented formal notations such as Z [Spivey 1988] or VDM [Jones 1986], suitable for the specification of *static* properties of systems. However, as we have advocated in the past, and as various researchers have observed, these languages, including Alloy, are less appropriate for specifying *dynamic* properties—properties regarding execution traces—due to the static nature of their specifications.

One of the reasons why the specification of properties of executions is complicated in Alloy has to do with the semantics of operations. 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 confirm that "output" and "after" are *intentional concepts*: 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. Nevertheless, we have proposed an alternative, consisting of extending Alloy with *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. This, besides enabling us to characterize properties regarding execution traces in a convenient way, provides a significant improvement to Alloy's expressiveness and analyzability, as we will show in this article.

In order to see how actions 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:

$$\begin{array}{lcl} Oper_1;Oper_2(x,\, y) & = & \text{some } z \mid (Oper_1(x, z) \text{ and } Oper_2(z,\, y)) \\ Oper_1 + Oper_2(x,\, y) & = & Oper_1(x,\, y) \text{ or } Oper_2(x,\, y) \end{array}$$

where the first and second arguments of each operation represent its input and output parameters, respectively. However, if we aim at specifying properties of executions, then it is highly likely 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 easily define sequential composition or nondeterministic choice in Alloy, as we showed before, finite unbounded iteration of operations cannot be defined using Alloy's constructs.

Alloy's developers have acknowledged the problem of specifying properties of executions, and have proposed a solution that includes a representation of 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 [Jackson et al. 2001], 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 indeed provides 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 that 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, as used in Alloy, 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 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 Jackson et al. [2001] that we described above, 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 proposed in Frias et al. [2005a], based on actions, execution traces are only *implicitly* used. This specification can be written in a simple and elegant way, as follows:

$$\begin{gathered}\{\alpha\}\\ (Oper_1 + Oper_2)^*.\\ \{\beta\}\end{gathered}$$

This states that every terminating execution of $(Oper_1 + Oper_2)^*$ (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 [Floyd 1967; Hoare 1969]. 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 is clear then that pursuing our task of adding actions to Alloy contributes toward the usability of the language. Also, 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 the automatic analyzability of its specifications. The analysis technique principally associated with Alloy is essentially a counterexample extraction mechanism, based on SAT solving. Basically, given a system specification and a statement about it, a counterexample of this statement (under the assumptions of the system description) is exhaustively searched for. Since first-order logic is not decidable, and Alloy is based on a proper extension of first-order logic, SAT solving cannot be used in general to guarantee the validity of (or, equivalently, the absence of counterexamples for) a theory; then, the exhaustive search for examples or counterexamples has to be performed up to a certain bound k in the number of elements in the universe of the interpretations. Thus, this analysis procedure can be regarded as a *validation* mechanism, rather than a *verification* procedure. Its usefulness for validation is justified by the interesting observation that, in general, if a statement is not true, there often exists a small size counterexample of it. This has become known as the *small scope hypothesis*. The described analysis technique is implemented by the Alloy Analyzer, a tool that employs state-of-the-art SAT solvers, such as MChaff, ZChaff [Moskewicz et al. 2001] and Berkmin [Goldberg and Novikov 2002], in order to search for examples or counterexamples of specifications.

In this article, we show that our extension of Alloy with actions, referred to as DynAlloy, besides being more expressive than standard Alloy, can also be complemented with efficient automatic analysis. We modify the Alloy tool in order to allow for efficient verification of DynAlloy specifications, and show that, in our experiments based on two case studies, the resulting tool was demonstrated to be more efficient than the standard Alloy Analyzer, for validating properties of executions. We also present a mechanism called *program atomization*, which improves the analyzability of properties regarding execution traces by considering certain programs as atomic steps in a trace. This is of special interest when combined with incremental specification; once a component C together with programs regarding this component are validated (this involves the analysis of execution traces), one can build new components in terms of the defined C, and new programs, in which some of the programs corresponding to C are considered as atomic. Intuitively, this technique allows us to compress the size of the traces of new programs, leading to the exploration of longer computations during the analysis, as we will show.

The remainder of this article is organized as follows. In Section 2 we summarize the main characteristics of the Alloy specification language. In Section 3 we introduce the reader to our extension of Alloy, the DynAlloy language. In Section 4 we present the DynAlloy tool, the tool that extends the Alloy Analyzer with support for actions, partial correctness assertions, and their corresponding analysis. In Section 5 we present the program atomization technique, and its impact on incremental validation. In Section 6 we present the case studies used to compare our approach with standard Alloy, and their corresponding running times. Finally, in Section 7 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 Jackson et al. [2001]. This example serves as a means for illustrating the standard features of the language and their associated semantics, the shortcomings overcome by our alternative semantics, and will be used as a basis for the properties of traces we will analyze.

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*:

$$\text{sig } Addr\ \{\ \} \qquad \text{sig } Data\ \{\ \}.$$

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 a set of addresses, and a partial mapping from these addresses to data values. In our case, we will use memories in order to model what is common to cache and main memories, so the signature must be declared as *abstract*:

```
abstract sig Memory {
    addrs: set Addr,
    map: addrs -> one Data
}
```

The modifier "one" 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 $\text{map}(a) = d$).

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 structures. 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 Figure 1, we present the grammar and semantics of Alloy's relational logic, the core logic on top of which all of Alloy 3.0's syntax and semantics are defined [Jackson et al. 2001]. An important difference with respect to previous versions of Alloy, as for instance the one presented in [Jackson 2002a], 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

$problem$ ::= decl*form
decl ::= var : $typexpr$
$typexpr$ ::=
$type$
| $type \to type$
| $type \Rightarrow typexpr$

form ::=
expr in expr (subset)
|!form (neg)
| form and form (conj)
| form or form (disj)
| $all\ v : type$/form (univ)
| $some\ v : type$/form (exist)

expr ::=
expr + expr (union)
| expr & expr (intersection)
| expr − expr (difference)
|∼ expr (transpose)
| expr.expr (navigation)
| ∗expr (closure)
| $\{v : t$/form$\}$ (set former)
| Var

Var ::=
var (variable)
| $Var[var]$ (application)

M : form $\to env \to Boolean$
X : expr $\to env \to value$
$env = (var + type) \to value$
$value = (atom \times \cdots \times atom) +$
$\quad (atom \to value)$

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

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

Fig. 1. Grammar and semantics of Alloy.

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 Figure 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:

```
pred 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, bearing in mind that m′ is meant to denote the memory (or memory state) resulting from the application of function Write, a -> d denotes the ordered pair ⟨a, d⟩, and ++ denotes relational overriding, defined as follows[1]:

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

We have already seen a number of constructs available in Alloy, such as the dot notation and signature extension, that resemble object oriented definitions. Moreover, operations, represented by functions in Alloy, can be "attached" to signature definitions, as in traditional object-oriented approaches. However, this is just a convenient notation, and functions describe operations of the whole set of signatures, that is, 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:

```
pred 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 implicitly being "applied" to cache memories.

As explained in Jackson et al. [2001], 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 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:

```
pred SysFlush(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 this definition of function SysFlush, x->Data denotes the set of all ordered pairs whose first elements fall into the set x, and whose second elements range over Data.

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

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

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

Recall (c.f. Figure 1), that 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) [Object Management Group 1997]. 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 set intersection. Another important constraint inherent in 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) and 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

DynAlloy is an extension of the Alloy modeling language. It was first presented in Frias et al. [2005b] as a formalism suitable for dealing with properties of executions of operations specified in Alloy. In addition to the automated analysis approach we propose in this article, DynAlloy admits deductive (equational) reasoning via a complete relational calculus, as was shown in Frias et al. [2005b]. So, one can reason about DynAlloy assertions using an automated theorem prover such as PVS [Owre et al. 2001].

The reason why we proposed this extension is that we wanted to provide a setting in which, besides functions describing sets of states, actions are also available, to represent state changes (i.e., to describe relations between input and output states). 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 [Harel et al. 2000], and the well-established suitability of dynamic logic for expressing partial correctness assertions.

3.1 Alloy Functions vs. DynAlloy 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 [Harel et al. 2000] arose in the early 1970s, with the intention of faithfully reflecting state change. Motivated by dynamic logic, we propose the use of actions to model state change in Alloy.

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

$$\begin{gathered}\{\alpha\}\\ A\\ \{\beta\}\end{gathered}$$

affirms that whenever action A is executed on a state satisfying α, if it terminates, it does so in a state satisfying β (notice that we are assuming a *partial* correctness reading of this expression). This approach is particularly appropriate, since behaviors described by predicates are better viewed as the result of performing an action on an input state. Thus, the definition of predicate Write

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

Fig. 2. Grammar for composite actions in DynAlloy.

could be expressed as an action definition, of the following form:

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

At first glance it is difficult to see the differences between (1) and (3), since both expressions 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 predicates, in principle, cannot.

An immediately apparent difference between (1) and (3) is that action *Write* does not involve the parameter m′, while predicate Write uses it. This is 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. Figure 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 Figure 1 with the addition of the following clause for building partial correctness statements:

$$
\begin{array}{l}
formula \ ::= \ \ldots \mid \{formula\}\ program\ \{formula\} \\
\qquad\qquad\qquad\qquad \text{“partial correctness”}
\end{array}
$$

The syntax for programs (cf. Figure 2) is the class of regular programs defined in Harel et al. [2000], 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 Figure 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

$$M[\{\alpha\}p\{\beta\}]e \;=\; \big(M[\alpha]e \implies \forall e' \left(\langle e, e'\rangle \in P[p] \implies M[\beta]e'\right)\big)$$

$$\begin{array}{lcl}
P : \mathit{program} \to \mathcal{P}\,(\mathit{env} \times \mathit{env}) & & \\
P[\langle \mathit{pre}, \mathit{post}\rangle] & = & A(\langle \mathit{pre}, \mathit{post}\rangle) \\
P[\alpha?] & = & \{\, \langle e, e'\rangle : M[\alpha]e \wedge e = e' \,\} \\
P[p_1 + p_2] & = & P[p_1] \cup P[p_2] \\
P[p_1 ; p_2] & = & P[p_1] ; P[p_2] \\
P[p^*] & = & P[p]^*
\end{array}$$

Fig. 3. Semantics of DynAlloy.

binary relation on the environments. We define function A as follows:

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

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 corresponding input values (therefore implementing a "frame condition"). 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. Notice that, since parallel composition of actions is not an allowed action combinator, this assumption does not restrict the sound composition of actions or programs. Although parallel composition is not available, one could still define it by means of interleaving of atomic actions, via nondeterministic choice and sequential composition, as usual in concurrent programming.

3.3 Specifying Properties of Executions in Alloy and DynAlloy

Suppose that we want to specify that a given property P is invariant under sequences of applications of the operations "SysFlush" and "SysWrite", from certain initial states. A useful technique for stating the invariance of a property P consists in specifying that P holds in the initial state(s), and that for every noninitial state and every operation $O \in \{\mathit{SysFlush}, \mathit{SysWrite}\}$, the following holds:

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

This specification, although sound, is too strong, since those properties that are indeed invariants, but violate the invariance in unreachable states states, fall outside the characterization (i.e., the characterization only covers *inductive* invariants). Of course it would be desirable to have a specification in which the states under consideration were exactly the reachable ones. The need for such a characterization motivated the introduction of *traces* in Alloy [Jackson et al. 2001].

The following example, extracted from Jackson et al. [2001], shows a signature for clock ticks:

```
sig Tick {
    system: System
}
```

Traces are built by imposing a total ordering on ticks. In the example developed here this is achieved by taking advantage of Alloy's modules, importing a specification of Ordering over the above Tick signature. Notice that we refer to it below directly as "TickOrder," since it is declared globally in the Alloy module.

We will refer to total orders on ticks as *traces*. The following "fact" states that all ticks in the trace are reachable from the first tick through the application of one of the operations under consideration, and that a property called "Init" holds in the first state:

```
fact {
    Init (TickOrder/first().system)
    all t: Tick-TickOrder/last() {
        SysFlush (t.system, TickOrder/next(t).system) or
        some a: Addr, d: Data |
            SysWrite (t.system, TickOrder/next(t).system, a, d)
    }
}
```

If we now want to assert that P is invariant, it suffices to assert that if P holds in the first state then it must hold 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 appropriate. 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.

Our approach is to consider SysWrite and SysFlush as actions that perform a certain operation on the state variables. The specification of actions SysWrite and SysFlush in DynAlloy is done as follows:

```
{ true }
  SysWrite(s: System)
{ some d: Data, a: Addr |
  s'.cache.map = s.cache.map ++ (a → d) and
  s'.cache.dirty = s.cache.dirty + a and
  s'.main = s.main }


{ true }
  SysFlush(s: System)
{some x: set s.cache.addrs |
  s'.cache.map = s.cache.map − x→Data and
  s'.cache.dirty = s.cache.dirty − x and
  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 {SysWrite, SysFlush}, we can assert the invariance of a property P under finite applications of functions SysWrite and SysFlush in a simple and elegant way, as follows:

$$\begin{gathered}\{Init(s) \wedge P(s)\} \\ (\mathrm{SysWrite}(s) + \mathrm{SysFlush}(s))^* \\ \{P(s')\}\end{gathered}$$

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 so due to the fact that 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 Jackson et al. [2001], has as the advantage that it allows one to automatically analyze properties of executions. Therefore, a question is immediately raised: Can DynAlloy specifications be automatically analyzed, and if so, how efficiently?

The main rationale behind our technique for the analysis of DynAlloy specifications is the translation of partial correctness assertions to first-order Alloy formulas, using weakest liberal preconditions [Dijkstra and Scholten 1990]. 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 a function:

$$wlp : program \times formula \rightarrow formula$$

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:

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

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

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

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:

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

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

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

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

Of course, this analysis mechanism, 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.

3.5 Expressiveness of DynAlloy Specifications

DynAlloy provides, to our understanding, a language that is more convenient than standard Alloy for specifying properties of traces (implicitly determined by program definitions), by means of partial correctness assertions. Moreover, as we discussed in the previous section, DynAlloy specifications can also be automatically analyzed, and as we will discuss later on, more efficiently than their corresponding equivalent specifications in standard Alloy.

However, DynAlloy has an important limitation in expressiveness, when compared with the standard Alloy's approach for dealing with properties of executions. DynAlloy allows one to validate an important class of properties of traces, the so called *invariance properties*. But, in its current form, DynAlloy cannot deal with more general properties of traces. In particular, one cannot express *liveness* properties by means of partial correctness assertions in DynAlloy, whereas in Alloy, with its explicit characterization of traces, such assertions are easily specified. Consider, for instance, the following assertion:

> In every execution of successive applications of SysWrite and SysFlush, any dirty address eventually becomes nondirty.

Using standard Alloy's explicit characterization of traces, we can express this assertion as follows[2]:

```
assert {
    all t: Tick | all a: Addr |
        a in t.system.cache.dirty implies
            (some t' : Tick | TickOrder/lt(t,t') and !(a in t'.system.cache.dirty))
}
```

This assertion is not expressible in DynAlloy by means of actions, implicit traces and partial correctness assertions.

Although liveness properties are expressible in the standard Alloy's approach to the characterization of traces, this does not necessarily mean that the Alloy Analyzer can be employed in order to validate these assertions. It is well known in the area of concurrent systems that only infinite traces can be actual counterexamples of liveness properties (liveness properties do not exclude finite prefixes of runs) [Alpern and Schneider 1985]; in other words, given a liveness assertion P, any finite trace can be extended to an infinite trace satisfying P. Since the analysis mechanisms associated with the Alloy and DynAlloy Analyzers cannot handle infinite runs, neither of the tools can be employed in order to *validate* liveness properties. One might, nevertheless, consider more complex mechanisms for detecting the violation of liveness properties, such as discovering loops in traces, which makes it possible to avoid certain desirable states in some infinite runs. However, Alloy employs an explicit state style of specification, which typically leads to relatively long loops exhibiting violations of liveness properties, usually longer than what Alloy's analysis mechanism can handle in practice.

This fact provides an important justification for studying deductive reasoning over Alloy specifications. Since the analysis mechanism associated with Alloy cannot handle liveness assertions, one could take advantage of the Alloy characterization of properties of traces and attempt to use a proof calculus, such as that presented in Frias et al. [2004], to perform deductive reasoning regarding liveness properties.

[2]Predicate TickOrder/lt reflects the *less-than* relation imposed by the ordering.

4. THE DYNALLOY TOOL

The Alloy tool [Jackson 2002b] is open source. This encourages and facilitates the development of 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 into Alloy using the translation function *pcat*, and then compiles the Alloy specification thus obtained.

In this section we discuss some modifications on the definition of the translation 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 into 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 the following:

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

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

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

So, we started looking for ways of improving the analysis of these formulas.

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 the following:

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

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 actions of the form $(a+b)^n$, the resulting formula was quite large. For $n = 2$, using the definition of *Bwlp*, we obtain:

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

Simple logical properties allow us to rewrite (10) as

$$((pre_a \wedge post_a) \Rightarrow Bwlp(a+b,\alpha)) \;\wedge\; ((pre_b \wedge post_b) \Rightarrow Bwlp(a+b,\alpha)) \,. \qquad (11)$$

At this point, notice that the formula $Bwlp(a+b,\alpha)$ appears twice in (11). Thus, computing $Bwlp((a+b)^n,\alpha)$ 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 the analysis using this translation was unfeasible for reasonable values of n.

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

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

Formula $Bwlp\,(a+b, \alpha)$ occurs only once in (12). Applying this simple transformation made the previous exponential-size formulas become linear-size.

Finally, Alloy's parser seems to perform quite badly on formulas involving many parentheses and implications; so, we desugarized the resulting formulas, by replacing the implications by their Boolean equivalent formulas in terms of conjunction and negation. We then applied the De Morgan rule in order to push the negations down to the atoms of the formulas. This is essentially the translation that is implemented in the DynAlloy tool.

Notice that this translation produces a single monolithic formula to be checked. The size of this formula still has an impact on compilation time. If we choose a number n of loop unfoldings to perform, a partial correctness assertion of the form

$$\begin{array}{c}\{pre(a)\}\\ (P_1 + \cdots + P_k)*\\ \{post(a, a')\}\end{array}$$

gets translated by DynAlloy into an Alloy assertion of the form

$$\begin{array}{l}\text{assert } pca\{\text{all } a_0, \ldots, a_n : S \mid pre(a_0) \Rightarrow\\ \quad \big(pre_{P_1}(a_0) \wedge post_{P_1}(a_0, a_1)\big) \vee \cdots \vee \big(pre_{P_k}(a_0) \wedge post_{P_k}(a_0, a_1)\big) \Rightarrow\\ \qquad\qquad \vdots\\ \quad \big(pre_{P_1}(a_{n-1}) \wedge post_{P_1}(a_{n-1}, a_n)\big) \vee \cdots \vee \big(pre_{P_k}(a_{n-1}) \wedge post_{P_k}(a_{n-1}, a_n)\big) \Rightarrow\\ \qquad\qquad\qquad\qquad\qquad\qquad post(a_0, a_n)\}\end{array}$$

In order to reduce the size of the previous assertion, we automatically create a new signature T

$$\begin{array}{l}\text{one sig } T\{\\ \quad a_0, \ldots, a_n : S\\ \}\end{array}$$

and replace the previous assertion by the following equivalent (c.f. (12)) facts and assertion:

$$\begin{array}{l}\text{fact } \{pre(T.a_0)\}\\ \text{fact } \{\big(pre_{P_1}(T.a_0) \wedge post_{P_1}(T.a_0, T.a_1)\big) \vee \cdots\\ \qquad\qquad \vee \big(pre_{P_k}(T.a_0) \wedge post_{P_k}(T.a_0, T.a_1)\big)\}\\ \vdots\\ \text{fact } \{\big(pre_{P_1}(T.a_{n-1}) \wedge post_{P_1}(T.a_{n-1}, T.a_n)\big) \vee \cdots\\ \qquad\qquad \vee \big(pre_{P_k}(T.a_{n-1}) \wedge post_{P_k}(T.a_{n-1}, T.a_n)\big)\}\\ \text{assert pca}\{post(T.a_0, T.a_n)\}\end{array}$$

In this way, compilation time is no longer a concern.

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 is the source code for the Alloy tool publicly available, but all the necessary software and tools required in order to generate the source code are freely available, also. 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 3.0 of the Alloy Analyzer. We changed the original Alloy "Build" command so that it now first translates a DynAlloy specification into Alloy, and then compiles the resulting model, in the same way standard Alloy does.

5. PROGRAM ATOMIZATION AND INCREMENTAL VALIDATION

When specifying software systems in Alloy, modelers typically modularize their specifications by using the notions of signature and signature extension. Thus, predicates, as operation descriptions, are often associated with particular signatures, and one typically defines operations corresponding to complex signatures out of operations corresponding to simpler signatures. Various examples of this situation are given in Jackson et al. [2001], and we reproduced some previously in this article. For instance, operation SysWrite (associated with signature System) is defined in terms of operation Write (associated with signature Memory); similarly, an operation SysRead (again, associated with signature System) is defined by operation Read (associated with signature Memory) in Jackson et al. [2001]. This practice of defining operations of complex modules out of the operations defined in the simpler modules is widely accepted as methodologically correct, since it favors encapsulation and reuse.

It is rather natural to think that modelers would take advantage of the modularization of the system specification in terms of signatures and their corresponding operations, and try to perform the analysis tasks modularly. When validating static properties, however, the validity of a particular property is *independent* of the validity of other intended properties, since it only depends on the facts (axioms of the specification) and the structure of the specification being validated. Nevertheless, one might take advantage of module definitions, and separate a system specification in modules. In this way, when validating static properties corresponding to the structure and operations of a signature S, one can consider only the axioms corresponding to S, and therefore reduce the size of the formulas to be analyzed. Still, for the top level module, validation will take into account the whole system specification, and therefore one cannot benefit from the validation tasks performed in simpler modules.

The situation is different when we look at properties regarding executions. As we said before, it is common, and considered methodologically correct, to define operations of a complex module M (in our case, a complex signature) in terms of the operations of the simpler modules M is built out from. So, when defining traces for validating a particular property of executions of M, one would be implicitly, using properties of traces of operations of the modules M is built out from.

Consider, as an initial example to illustrate incremental validation in DynAlloy, a complex system encapsulating an instance of a slightly modified version of our System specification and an external Cache. First we introduce the modifications to our System specification. They simply consist of making the SysWrite action take the address and data to write as parameters, giving rise to the following new specification:

$$\{true\}$$

$$\text{SysWrite(s: } \mathit{System}\text{, a: } \mathit{Addr}\text{, d: } \mathit{Data}\text{)}$$

$$\{ \text{ s'.cache} = \text{s.cache} ++ (\text{a} \to \text{d}) \wedge \text{s'.cache.dirty} = \text{s.cache.dirty} + \text{a} \wedge \text{s'.main} = \text{s.main } \}$$

We will also consider the specification of the Write action, as presented in (3), and a new action, called Input, specified as follows:

$$\{true\}$$

$$\text{Input(a: } \mathit{Addr}\text{, d: } \mathit{Data}\text{)}$$

$$\{\text{some ai: } \mathit{Addr}\text{, di: } \mathit{Data} \mid \text{a'} = \text{ai and d'} = \text{di}\}$$

Finally, the specification of a complex system is the following:

```
sig ComplexSystem {
    s: System,
    c: Cache
}
```

Consider also two actions that ComplexSystem comes equipped with, one for writing on the encapsulated system and one for flushing a block of cache addresses, defined respectively as follows:

$$\begin{aligned}\text{CSysWrite(c: ComplexSystem, a: Addr, d: Data)} =&\\ \text{Input(a, d);SysWrite(c.s, a, d);Write(c.c, a, d)}&\end{aligned}$$

$$\text{CSysFlush(c: ComplexSystem)} = (\text{SysFlush(c.s)})^{*}$$

Note that these actions are programs defined in terms of the more basic actions, Input, Write, SysWrite, and SysFlush, corresponding to signatures Memory and System. Now, suppose that we want to check whether the system underlying a complex system retains the property DirtyInv, under sequences of applications of CSysWrite and CSysFlush. This can easily be asserted by using a partial correctness assertion, in the following way:

$$\begin{array}{c}\{\mathit{DirtyInv}(c.s)\}\\ (\text{CSysWrite}(c, a, d) + \text{CSysFlush}(c))^{*}\\ \{\mathit{DirtyInv}(c'.s)\}\end{array} \tag{13}$$

Here, we have a choice for performing the analysis of this property. The easiest, but not necessarily the best, would be to *unfold* the definitions of CSysWrite and CSysFlush, obtaining the following assertion regarding ComplexSystem:

$$\begin{array}{c}\{\mathit{DirtyInv}(c.s)\}\\ (\mathit{Input}(a, d); \mathit{SysWrite}(c.s, a, d); \mathit{Write}(c.c, a, d)\ +\ \text{SysFlush}(c.s)^{*})^{*}\\ \{\mathit{DirtyInv}(c'.s)\}\end{array}$$

This alternative does not correspond to incremental validation, since we do not carry out the analysis based on already validated properties of simpler actions; instead, we simply expand their definitions. Notice that we have nested stars in the above program. From the point of view of the automated validation of traces (in the style of standard Alloy), if the bound imposed on iteration is k, we will need to explore traces whose length goes up to $k \times k$. In the style of validation of traces of DynAlloy, this has an equivalent impact, but in the size of the formula to be validated. This, even for small values of k, will lead us to scopes beyond what is currently analyzable in practice, using both approaches.

On the other hand, we might first consider the validity of the following assertion as a property of systems:

$$\begin{array}{c}\{\mathit{DirtyInv}(s)\}\\ \text{SysFlush(s)}^{*}\\ \{\mathit{DirtyInv}(s')\}\end{array}$$

Indeed, the iteration of action SysFlush preserves DirtyInv, and therefore we will be able to validate this assertion. In this case, if k is the bound imposed on the number of iterations, the formula corresponding to the above partial correctness assertion will only be linearly proportional to k.

Assuming that we have validated this assertion, and gained confidence about its validity, we can *atomize* it. Basically, the atomization consists of considering a new atomic action on systems, let us call it BlockSysFlush, whose pre- and post-conditions are the ones corresponding to the already validated partial correctness assertion. After the atomization of SysFlush$(c.s)^{*}$ the specification of CSysFlush can be rewritten as:

$$\text{CSysFlush(c: ComplexSystem)} = \text{BloskSysFlush(c.s)}$$

and consequently, the assertion presented in (13) can be equivalently stated as

$$\begin{array}{c}\{\mathit{DirtyInv}(c.s)\}\\ (\text{CSysWrite}(c) + \text{BlockSysFlush}(c.s))^{*}\\ \{\mathit{DirtyInv}(c'.s)\}\end{array}$$

becoming an unbounded iteration of the nondeterministic choice between program CSysWrite (which becomes the sequential composition of three atomic actions after the unfolding) and the atomic action BlockSysFlush (resulting from the described atomization). Therefore, validating it will require the generation of a formula which depends *linearly* on the bound k imposed on the number of iterations.

This reasoning corresponds to incremental validation in the sense of using already validated properties of smaller modules as a technique to reduce the complexity of validating properties of bigger ones. The same reasoning can be applied to the action CSysWrite, but in this case as a property of the complex systems, by validating the following assertion:

$$\begin{gathered}\{\mathit{DirtyInv}(c.s)\}\\ \mathit{Input}(a,d);\mathit{SysWrite}(c.s,a,d);\mathit{Write}(c.c,a,d)\\ \{\mathit{DirtyInv}(c'.s)\}\end{gathered}$$

This can be seen as the decomposition of the validation of the partial correctness assertion presented in (13) into three less complex ones. Basically, in the first case we have modularized the validation of two nested iterations as the validation of two separate iterations; and in the second case we have abstracted a program by validating the sequential composition separately.

Notice that in standard Alloy, this would be very difficult to achieve, since the explicit definition of traces would force us to define different sorts of traces for different programs, and then explicitly indicate how the elements in these sorts are related. It is not clear whether this might favor the analyzability of specifications.

Let us describe in more detail the general procedure for incremental validation via atomization in DynAlloy, and also justify more precisely its soundness. Suppose that we need to validate an assertion of the form:

$$\begin{gathered}\{\alpha\}\\ P\\ \{\beta\}\end{gathered}$$

where the program term P contains some subprogram or subterm P_s. Moreover, suppose that in a previous validation activity, we needed to check for the validity of

$$\begin{gathered}\{\alpha_s\}\\ P_s\\ \{\beta_s\}\end{gathered}$$

that is, the partial correctness of P_s with respect to the pre- and post-condition specification $\{\alpha_s\}\{\beta_s\}$. Let us suppose further, that for a bound k on the number of iterations, no counterexamples were found for the partial correctness assertion of P_s. Then we can assure that, in all traces of P_s with iterations of at most k steps, if the trace starts in a state satisfying α_s then it ends in a state satisfying β_s. Then we might consider extending our set of atomic actions with a new atomic action a_{P_s}, whose specification is the following:

$$\{\alpha_s\}$$
$$a_{P_s}$$
$$\{\beta_s\}$$

We can then *replace* every occurrence of P_s in P by a_{P_s}, obtaining a variant P_v of P. This process is what we call program atomization. Notice that, if we analyze P_v with k as a bound on the number of iterations and no counterexamples are found, then necessarily the original P has no counterexamples under the same circumstances (i.e., with the same bound k on the number of iterations, and the same bounds on the numbers of atoms of the domains of the specification). This is justified by Theorem 5.1.

THEOREM 5.1. *Let $\{\alpha\}P\{\beta\}$ be a DynAlloy partial correctness assertion. Let P_s, be a subterm of P, and $\{\alpha_s\}P_s\{\beta_s\}$, a partial correctness assertion for it. Let a_{P_s} be an atomic action with the same specification as P_s. Further, suppose that, for given bounds for domains in the specification and for a bound k on the number of iterations, neither $\{\alpha_s\}P_s\{\beta_s\}$ nor $\{\alpha\}P[a_{P_s}/P_s]\{\beta\}$ (where $P[a_{P_s}/P_s]$ denotes the program resulting from replacing all occurrences of P_s in P by a_{P_s}) have any counterexamples. Then, under the same given bounds for domains on the specification and for the bound k on the number of iterations, $\{\alpha\}P\{\beta\}$ does not have any counterexamples.*

PROOF. Let $\{\alpha\}P\{\beta\}$ be a DynAlloy partial correctness assertion, P_s, a subterm of P, and $\{\alpha_s\}P_s\{\beta_s\}$, a partial correctness assertion for it. Let a_{P_s} be an atomic action with the same specification as P_s. Suppose that under certain bounds on the domains of the specification and a bound k on the number of iterations, $\{\alpha_s\}P_s\{\beta_s\}$ has no counterexamples. Let us suppose also that under the same bounds on the domains of the specification and the bound k on the number of iterations, $\{\alpha\}P[a_{P_s}/P_s]\{\beta\}$ does not have any counterexamples, but $\{\alpha\}P\{\beta\}$ does have a counterexample. If this is the case, then there exists a run $\sigma = s_1, \ldots, s_n$ of P such that s_1 satisfies the initial condition α but s_n does not satisfy β. Notice that since $\{\alpha\}P[a_{P_s}/P_s]\{\beta\}$ does not have any counterexamples, necessarily P_s must be involved in σ (otherwise, σ would also be a run of P after the atomization, and it would be a counterexample of its specification). So σ has the form:

$$\sigma = s_1, \ldots, s_{b_1}, \ldots, s_{e_1}, \ldots, s_{b_2}, \ldots, s_{e_2}, \ldots, s_{b_m}, \ldots, s_{e_m}, \ldots, s_n$$

where $s_{b_i}, \ldots, s_{e_i}$ corresponds to runs of P_s. Since we have k as the limit on iteration, and we know that $\{\alpha_s\}P_s\{\beta_s\}$ has no counterexamples with that bound, we know that if each s_{b_i} satisfies α_s, then each s_{e_i} satisfies β_s. Let us suppose that, for some x, s_{b_x} does not satisfy α_s. We can replace all the previous $s_{b_i}, \ldots, s_{e_i}$, with $i < x$, by s_{b_i}, s_{e_i}, obtaining the following:

$$\sigma_x = s_1, \ldots, s_{b_1}, s_{e_1}, \ldots, s_{b_2}, s_{e_2}, \ldots, s_{b_x}$$

which is a prefix of a trace of $P[a_{P_s}/P_s]$. Moreover, s_1 satisfies α. Thus this prefix is a counterexample of $\{\alpha\}P[a_{P_s}/P_s]\{\beta\}$, since it corresponds to a run starting in a state satisfying the initial condition α and which, before reaching the final state, violates the precondition of an atomic action, not allowing us to assert the validity of the postcondition β in the final state. Notice also that this trace is a run bounded by k on the number of iterations. Since, as we assumed, $\{\alpha\}P[a_{P_s}/P_s]\{\beta\}$ does not have counterexamples with iteration bounded by k, this cannot be possible. Therefore, all s_{b_i}'s must satisfy α_s, and consequently all s_{e_i}'s must satisfy β_s. Then the trace:

$$s_1, \ldots, s_{b_1}, s_{e_1}, \ldots, s_{b_2}, s_{e_2}, \ldots, s_n$$

is a trace of $P[a_{P_s}/P_s]$ (with iteration bounded by k), with s_1 satisfying α and s_n not satisfying β; again, this contradicts our hypothesis that $\{\alpha\}P[a_{P_s}/P_s]\{\beta\}$ does not have counterexamples bounded by k. Therefore, we arrived at a contradiction, proving that, if for certain bounds $\{\alpha\}P[a_{P_s}/P_s]\{\beta\}$ does not have counterexamples (for particular bounds on the domains and the length of iterations), then $\{\alpha\}P\{\beta\}$ does not have counterexamples either, for the same bounds on the domains and the length of iterations. $\square$

Notice that the implication in the other direction does not necessarily hold. Consider as a trivial, and somehow extreme, example the following partial correctness assertions:

$$\begin{gathered}\{\mathit{DirtyInv}(s)\}\\ \mathit{SysWrite}(s,a,d);\mathit{SysWrite}(s,a,d)\\ \{\text{some } s'.\mathit{cache}.\mathit{dirty}\}\end{gathered}$$

$$\begin{gathered}\{\mathit{DirtyInv}(s) \ \&\& \ \text{no } s.\mathit{cache}\}\\ (\mathit{SysWrite}(s,a,d);\mathit{SysWrite}(s,a,d)+\mathit{SysFlush}(s))*\\ \{\mathit{DirtyInv}(s')\}\end{gathered}$$

Notice that the first of these specifications is valid, so we will not find counterexamples of any size (moreover, it is not sensible to the bound on iteration, since the program does not even involve iteration). The second specification is also valid, so we will not find any counterexamples of it. Now suppose we apply program atomization, replacing the subterm $\mathit{SysWrite}(s,a,d);\mathit{SysWrite}(s,a,d)$ in the program of the second specification, and considering the first specification as its definition, then the resulting program will indeed have counterexamples. The problem here has to do with the fact that the postcondition of the first specification is too weak for our embedding into the program of the second specification. This fact leads us to a "false negative". Theorem 5.1 guarantees that we will not have any false positives when using program atomization.

6. CASE STUDIES

In this section we analyze three case studies. The first is an assertion whose validity follows from the specification, and therefore has no counterexamples. It will serve as a stress test for Alloy and DynAlloy. The assertion of the second case study has counterexamples, and is useful for verifying how efficiently can these be found using DynAlloy. The third is an example of the improvement

Table I. Verification Times for the Assertion *DirtyInv* Using the SAT Solver *MChaff*

#elems →	4		5		6		7	
Tr. length ↓	DynAlloy	Alloy	DynAlloy	Alloy	DynAlloy	Alloy	DynAlloy	Alloy
6	**0′05″**	0′25″	**0′05″**	2′16″	**0′23″**	4′19″	**0′11″**	6′41″
7	**0′04″**	1′37″	**0′06″**	10′03″	**0′48″**	3′54″	**0′29″**	15′01″
8	**0′10″**	3′48″	**0′25″**	5′39″	**1′07″**	8′55″	**2′26″**	22′51″
9	**0′12″**	8′21″	**0′24″**	6′12″	**1′32″**	11′13″	**1′49″**	53′34″
10	**0′36″**	16′17″	**0′16″**	32′30″	**1′34″**	16′04″	**1′05″**	> 60′
11	**1′10″**	36′13″	**0′15″**	12′39″	**4′55″**	28′59″	**2′05″**	> 60′
12	**0′38″**	14′21″	**1′53″**	20′16″	**3′19″**	46′19″	**2′07″**	> 60′

in the verification times of the first case study when program atomization is carried out on the specification.

The analysis was carried out using a 64-bit AMD Athlon 3200 with 2 GB of RAM running on a dual channel architecture. For the analysis we imposed a limit of 60 minutes. Those runs that did not finish within 60 minutes were stopped and marked in the tables as "> 60′".

All case studies are based on the same Alloy model, namely the model of cache memories introduced before. We think these case studies are good representatives of typical Alloy/DynAlloy models and assertions. Nevertheless, we plan to carry out experimental analyses on other models, which due to a lack of time we have been unable to develop and present in this article.

6.1 Case Study 1: DirtyInv

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

```
assert DirtyInvAssertion {
   DirtyInv (TickOrder/first().system) =>
   DirtyInv (TickOrder/last().system)
}
```

The corresponding DynAlloy specification is:

```
assert DirtyInvAssertionDynAlloy {
   {DirtyInv(s)}
      (SysWrite(s) + SysFlush(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 I and II 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:

Table II. Verification Times for the Assertion *DirtyInv* Using the SAT Solver *Berkmin*

#elems →	4		5		6		7	
Tr. length ↓	DynAlloy	Alloy	DynAlloy	Alloy	DynAlloy	Alloy	DynAlloy	Alloy
6	**0′02″**	0′09″	**0′04″**	0′15″	**0′05″**	0′54″	**0′10″**	3′49″
7	**0′03″**	0′11″	**0′06″**	0′20″	**0′09″**	1′15″	**0′14″**	6′16″
8	**0′05″**	0′23″	**0′07″**	0′39″	**0′13″**	1′52″	**0′15″**	11′23″
9	**0′11″**	0′27″	**0′12″**	0′48″	**0′13″**	3′00″	**0′23″**	13′26″
10	**0′21″**	0′29″	**0′20″**	1′13″	**0′51″**	4′19″	**1′00″**	20′26″
11	**0′22″**	0′30″	**0′32″**	1′31″	**1′14″**	7′29″	**1′40″**	33′30″
12	**0′28″**	0′44″	**1′20″**	1′58″	**1′36″**	12′00″	**2′53″**	44′19″

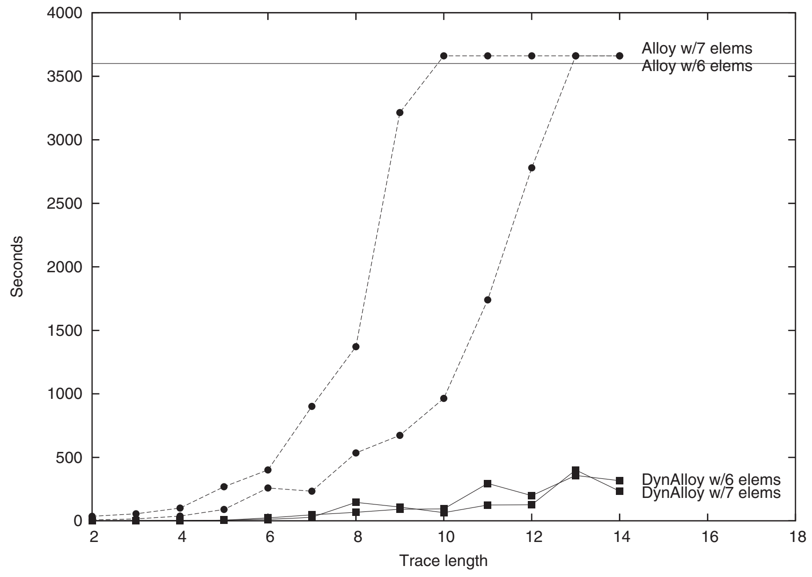

Fig. 4. Evolution graph for the assertion *DirtyInv* using the SAT solver *MChaff* with domain sizes 6 and 7.

```
check DirtyInvAssertionAlloy for k but n+1 Memory, n+1
Cache, n+1 MainMemory, n+1 System, n+1 Tick.
```

For the DynAlloy specification, we use:

```
check DirtyInvAssertionDynAlloy for k but n+1 Memory, n+1
Cache, n+1 MainMemory, n+1 System.
```

Figure 4 shows how both approaches evolve in time as the trace length grows. The results are shown for two domain sizes ($k = 6$ and $k = 7$). Analogously, Figure 5 shows how both approaches evolve in time as the domain size grows. The results are shown for two trace lengths ($n = 9$ and $n = 14$).

As we did for *MChaff*, Figure 6 shows how both approaches evolve in time as the trace length grows. The results are shown for two domain sizes ($k = 6$ and $k = 7$). Analogously, Figure 7 shows how both approaches evolve in time as the domain size grows. The results are shown for two trace lengths ($n = 9$ and $n = 14$).

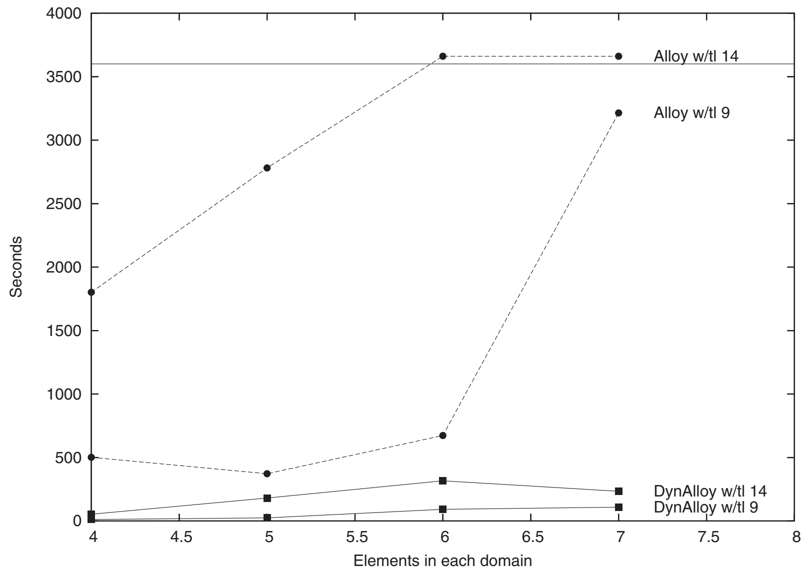

Fig. 5. Evolution graph for the assertion *DirtyInv* using the SAT solver *MChaff* with trace lengths 9 and 14.

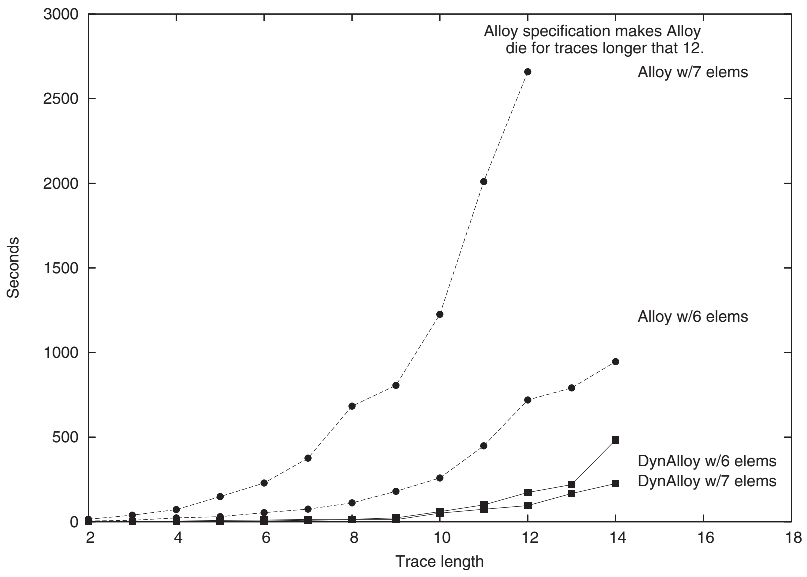

Fig. 6. Evolution graph for the assertion *DirtyInv* using the SAT solver *Berkmin* with domain sizes 6 and 7.

Running times for RelSat will not be presented because they exhibited worse behavior, compared to MChaff and Berkmin, for out case studies. As an example, the verification time for traces of length 2 and 6 elements was $10'17''$ for DynAlloy and $> 60'$ for Alloy.

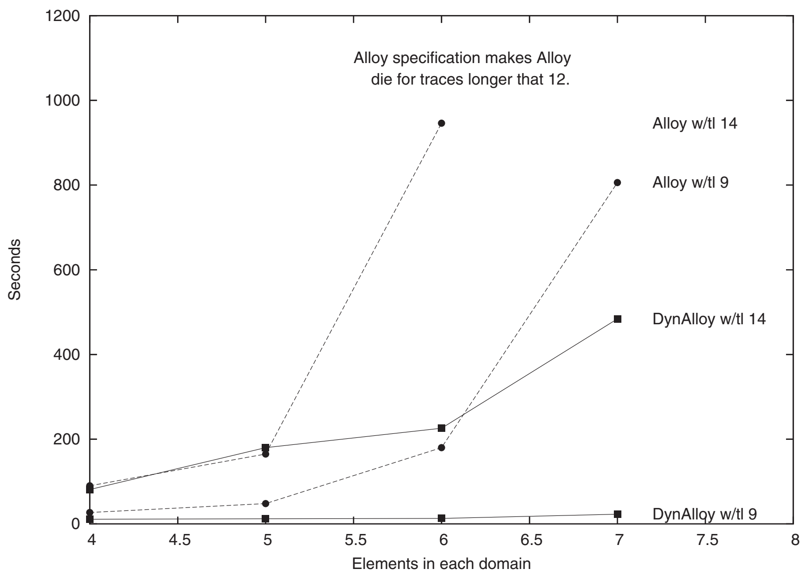

Fig. 7. Evolution graph for the assertion *DirtyInv* using the SAT solver *Berkmin* with trace length 9 and 14.

The noticeable difference in running times between our DynAlloy specifications and the standard Alloy specifications can be explained by our discussion at the end of Section 4.1. Due to the Alloy characterization of traces, it is relatively straightforward to realize that, for executions of programs of the form $(p_1 + \cdots + p_n)^*$, the number of traces to be checked depends exponentially on the length of traces. In DynAlloy, on the other hand, for programs of the form $(p_1 + \cdots + p_n)^*$, the size of the formula to be validated depends linearly on the bound k imposed on iteration.

6.2 Case Study 2: FreshDir

Let us continue with our second case study. 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 SysFlush 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 predicate specifying that a CacheSystem is empty, and another describing the fresh address property. They are given next.

```
pred Init (s: System) {
     no s.cache.dirty
     no s.cache.map
     no s.main.map
}

pred FreshDir (s: System) {
     some a: Addr { all d: Data {
```

Table III. Verification times for *FreshDir*

	MChaff		Berkmin	
Tr. length ↓	Alloy	DAlloy	Alloy	DAlloy
3	0′03″	**0′00″**	0′02″	**0′01″**
4	0′21″	**0′01″**	0′16″	**0′02″**
5	12′20″	**4′21″**	2′07″	**0′04″**
6	> 60′	**55′43″**	28′44″	**0′09″**
7	> 60′	> 60′	> 60′	**0′35″**
8	> 60′	> 60′	> 60′	**2′15″**
9	> 60′	> 60′	> 60′	**5′57″**
10	> 60′	> 60′	> 60′	**54′27″**

```
        ! ((a -> d) in s.main.map) &&
        ! ((a -> d) in s.cache.map) } }
}
```

With Init and FreshDir already specified, we can easily specify our assertion, both in Alloy and DynAlloy, as follows:

```
assert FreshDirAssertion {
   Init (TickOrder/first ().system) =>
      FreshDir (TickOrder/last ().system)
}

assert FreshDirAssertionDynAlloy {
  {Init(s)}
    (SysWrite(s) + SysFlush(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 including 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 k but n Addr, n+1 Memory,
n+1 Cache, n+1 MainMemory, n+1 System, n+1 Tick.
```

For DynAlloy we use

```
check FreshDirAssertionDynAlloy for k but n Addr, n+1
Memory, n+1 Cache, n+1 MainMemory, n+1 System.
```

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

Finally, Figures 8 and 9 show how both approaches evolve in time as the trace length grows. The results are shown only for domain size 3.

6.3 Case Study 3: Incremental Validation of DirtyInv

We finish this section by showing the impact of program atomization in the analysis of DynAlloy specifications. We extend the specification of memories with caches by defining signature ComplexSystem, as shown in Section 5.

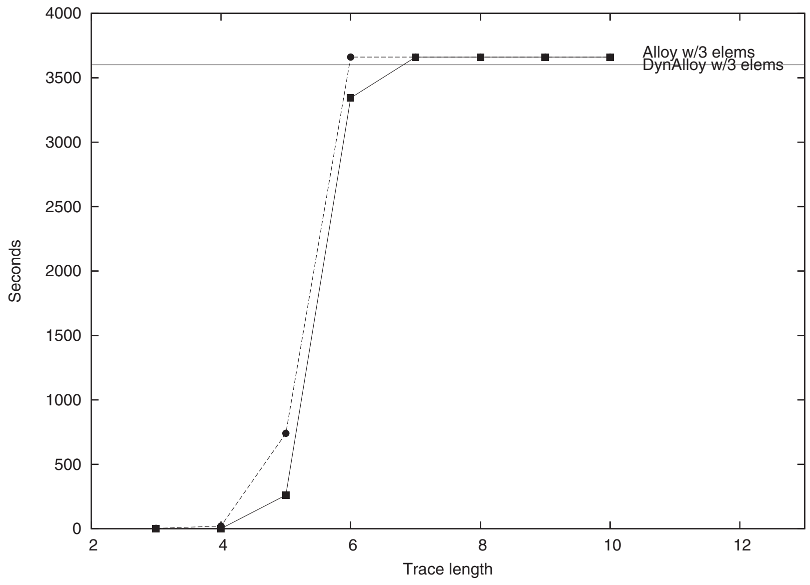

Fig. 8. Evolution graph for the assertion *FreshDir* using *MChaff* with domain size 3.

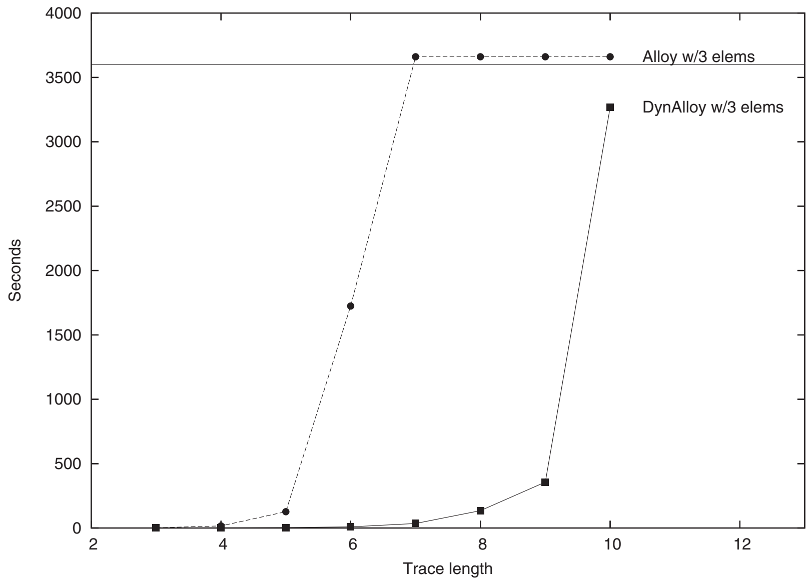

Fig. 9. Evolution graph for the assertion *FreshDir* using the SAT solver *Berkmin* with domain size 3.

We first check whether DirtyInv(c.s) is invariant under sequences of applications of CSysWrite and CSysFlush, by unfolding their definitions.

```
assert ComplexDirtyInvUnfolding {
{DirtyInv(c.s)}
  (Input(a, b);SysWrite(c.s, a, d);Write(c.c, a, d) +
      (FlushDA(c.s))*)*
  {DirtyInv(c'.s)} }
```

Then, we perform incremental validation via program atomization. First, we check whether CSysFlush preserves DirtyInv. This is specified in DynAlloy by the following assertion on systems:

```
assert ComplexSystem_CSysFlush_DirtyInv {
    {DirtyInv(s)}
        SysFlush(s)*
    {DirtyInv(s')}
}
```

Having validated this assertion, and not finding any counterexamples, we can now atomize this program. This corresponds to considering a new atomic action CSysFlush associated with signature System, defined as follows:

```
{DirtyInv(s)}
  CSysFlush(s)
{DirtyInv(s')}
```

Then, we can validate the preservation of DirtyInv under the application of the operation CSysWrite. This assertion can be stated as follows:

```
assert ComplexSystem_CSysWrite_DirtyInv {
    {DirtyInv(c.s)}
        Input(a, b);SysWrite(c.s, a, d);Write(c.c, a, d)
    {DirtyInv(c'.s)}
}
```

Once this assertion is verified, we can proceed by atomizing the program CSysWrite (but in this case associated with signature ComplexSystem).

Using these two results, we can validate a new assertion, ComplexSystem_DirtyInv, defined as follows:

```
assert ComplexSystem_DirtyInv {
    {DirtyInv(c.s)}
        (CSysWrite(c) + CSysFlush(c.s))*
    {DirtyInv(c'.s)}
}
```

The comparison of the running times required to validate the preservation of DirtyInv, both for incremental and nonincremental approaches, are presented in Table IV. The first column of the table shows the bound imposed to both iterations, while the second shows the size of each of the domains involved in the property.

7. CONCLUSIONS AND FURTHER WORK

We believe that using actions within Alloy in order to represent state change is a methodological improvement. Using actions favors a better separation of concerns, since models do not need to be reworked in order to describe the

Table IV. Verification Times for *DirtyInv* Under Incremental and Nonincremental Approaches

		MChaff		Berkmin	
Bound	# elements	Nonincremental	Incremental	Nonincremental	Incremental
2	4	$0'32''$	$\mathbf{0'02''}$	$0'04''$	$\mathbf{0'03''}$
	5	$1'10''$	$\mathbf{0'03''}$	$0'03''$	$\mathbf{0'03''}$
	6	$0'37''$	$\mathbf{0'03''}$	$0'08''$	$\mathbf{0'03''}$
	7	$2'17''$	$\mathbf{0'04''}$	$0'11''$	$\mathbf{0'04''}$
	8	$4'57''$	$\mathbf{0'08''}$	$0'11''$	$\mathbf{0'06''}$
	9	$1'49''$	$\mathbf{0'09''}$	$0'24''$	$\mathbf{0'09''}$
3	4	$> 60'$	$\mathbf{0'04''}$	$0'51''$	$\mathbf{0'03''}$
	5	$> 60'$	$\mathbf{0'03''}$	$0'52''$	$\mathbf{0'03''}$
	6	$> 60'$	$\mathbf{0'04''}$	$2'09''$	$\mathbf{0'05''}$
	7	$> 60'$	$\mathbf{0'08''}$	$6'26''$	$\mathbf{0'06''}$
	8	$> 60'$	$\mathbf{0'12''}$	$2'38''$	$\mathbf{0'09''}$
	9	$> 60'$	$\mathbf{0'22''}$	$13'27''$	$\mathbf{0'13''}$
4	4	$> 60'$	$\mathbf{0'05''}$	$16'23''$	$\mathbf{0'03''}$
	5	$> 60'$	$\mathbf{0'06''}$	$15'22''$	$\mathbf{0'04''}$
	6	$> 60'$	$\mathbf{0'07''}$	$25'52''$	$\mathbf{0'05''}$
	7	$> 60'$	$\mathbf{0'80''}$	$22'33'$	$\mathbf{0'08''}$
	8	$> 60'$	$\mathbf{0'31''}$	$> 60'$	$\mathbf{0'13''}$
	9	$> 60'$	$\mathbf{3'51''}$	$> 60'$	$\mathbf{0'18''}$
5	4	$> 60'$	$\mathbf{0'09''}$	$> 60'$	$\mathbf{0'05''}$
	5	$> 60'$	$\mathbf{0'08''}$	$> 60'$	$\mathbf{0'05''}$
	6	$> 60'$	$\mathbf{0'17''}$	$> 60'$	$\mathbf{0'09''}$
	7	$> 60'$	$\mathbf{1'09''}$	$> 60'$	$\mathbf{0'12''}$
	8	$> 60'$	$\mathbf{21'41''}$	$> 60'$	$\mathbf{0'19''}$
	9	$> 60'$	$> 60'$	$> 60'$	$\mathbf{0'18''}$

adequate notion of trace modeling of 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. Furthermore, thanks to this standardized and automated characterization of executions, we are able to perform incremental validation of properties of executions. The technique associated with this possibility, called program atomization, enables us to explore longer computations during the analysis activities.

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 in the future, 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.

We also observed that DynAlloy has an important limitation in expressive power, since in its current form, it only allows us to express invariance properties. The standard Alloy approach to the modeling of traces, on the other hand, allows one to specify other kinds of properties, particularly liveness properties. However, neither the DynAlloy tool (due to expressiveness restrictions) nor the Alloy Analyzer (due to intrinsic properties of liveness assertions) can be employed for analyzing liveness properties. This is due to the fact that liveness properties cannot have finite traces as counterexamples (liveness properties do not exclude finite prefixes of runs) [Alpern and Schneider 1985]. Since the analysis mechanisms associated with Alloy and DynAlloy cannot handle infinite runs, we are currently exploring the use of a complete proof calculus, presented in Frias et al. [2004], to perform deductive verification of assertions, particularly liveness assertions, in Alloy specifications. We are examining a characterization of a similar proof calculus for a closely related language in PVS, presented in Lopez Pombo et al. [2002], for this task.

ACKNOWLEDGMENTS

The authors wish to thank Emina Torlak, who proposed an alternative to our previous way of translating partial correctness assertions into Alloy. Her alternative translation is the one that DynAlloy currently implements, and resulted in much better compilation times. The authors also wish to thank the anonymous reviewers for their helpful comments.

REFERENCES

Alpern, B. and Schneider, F. B. 1985. Defining liveness. *Inform. Proc. Lett. 21*, 4, 181–185.

Dijkstra, E. W. and Scholten, C. S. 1990. *Predicate Calculus and Program Semantics*. Springer-Verlag, New York, NY.

Floyd, R. W. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, Providence, Rhode Island, pp. 19–32.

Frias, M. F., Galeotti, J. P., Lopez Pombo, C. G., and Aguirre, N. M. 2005a. DynAlloy: upgrading alloy with actions. In *Proceedings of the 27th International Conference on Software Engineering*, G.-C. Roman, Ed. Association for Computing Machinery and IEEE Computer Society, ACM Press, St. Louis, Missouri, USA, 442–450.

Frias, M. F., Lopez Pombo, C. G., and Aguirre, N. M. 2004. An equational calculus for Alloy. In *Proceedings of the Sixth International Conference on Formal Engineering Methods (ICFEM)*, J. Davies, W. Schulte, and M. Barnett, Eds. Lecture Notes in Computer Science, vol. 3308. Springer-Verlag, Seattle, Washington, 162–175.

Frias, M. F., Lopez Pombo, C. G., Baum, G. A., Aguirre, N., and Maibaum, T. S. E. 2005b. Reasoning about static and dynamic properties in alloy: A purely relational approach. *ACM Trans. Softw. Eng. Meth. 14*, 4, 478–526.

Goldberg, E. and Novikov, Y. 2002. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and, Test in Europe*, C. D. Kloos and J. da Franca, Eds. IEEE Computer Society, Paris, France, 142–149.

Harel, D., Kozen, D., and Tiuryn, J. 2000. Dynamic logic. *Foundations of Computing*. MIT Press, Cambridge, MA.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Comm. ACM 12*, 10, 576–583.

Jackson, D. 2002a. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Meth. 11*, 2, 256–290.

Jackson, D. 2002b. *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*. MIT Laboratory for Computer Science, Cambridge, MA.

Jackson, D., Shlyakhter, I., and Sridharan, M. 2001. 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*. ACM Press, Vienna, Austria, 62–73.

Jones, C. 1986. *Systematic Software Development Using VDM*. Prentice Hall, Hertfordshire, UK.

Lopez Pombo, C. G., Owre, S., and Shankar, N. 2002. A Semantic Embedding of the $\mathbf{A_g}$ Dynamic Logic in PVS. Tech. Rep. SRI-CSL-02-04, Computer Science Laboratory, SRI International. July.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Conference on Design Automation*, J. Rabaey, Ed. ACM Press, Las Vegas, Nevada, 530–535.

Object Management Group. 1997. *Object Constraint Language Specification*. Object Management Group, Needham, MA. version 1.1.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. 2001. *PVS Language Reference*, Version 2.4 ed. SRI International.

Spivey, J. M. 1988. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, New York, NY.

Received August 2005; revised July 2006, January 2007; accepted January 2007