

# Towards Abstraction for DynAlloy Specifications

Nazareno M. Aguirre<sup>1</sup>, Marcelo F. Frias<sup>2</sup>, Pablo Ponzio<sup>1</sup>, Brian J. Cardiff<sup>2</sup>,  
Juan P. Galeotti<sup>2</sup>, and Germán Regis<sup>1</sup>

<sup>1</sup> Department of Computer Science, FCEFQyN, Universidad Nacional de Río Cuarto  
and CONICET, Argentina

{naguirre,pponzio,gregis}@dc.exa.unrc.edu.ar

<sup>2</sup> Department of Computer Science, FCEyN, Universidad de Buenos Aires and  
CONICET, Argentina

{mfrias,bcardiff,jgaleotti}@dc.uba.ar

**Abstract.** DynAlloy is an extension of the Alloy language to better describe state change via actions and programs, in the style of dynamic logic. In this paper, we report on our experience in trying to provide abstraction based mechanisms for improving DynAlloy specifications with respect to SAT based analysis. The technique we employ is based on predicate abstraction, but due to the context in which we make use of it, is subject to the following more specific improvements: *(i)* since DynAlloy's analysis consists of checking partial correctness assertions against programs, we are only interested in the initial and final states of a computation, and therefore we can safely abstract away some intermediate states in the computation (generally, this kind of abstraction cannot be safely applied in model checking), *(ii)* since DynAlloy's analysis is inherently bounded, we can safely rely on the sole use of a SAT solver for producing the abstractions, and *(iii)* since DynAlloy's basic operational unit is the atomic action, which can be used in different parts within a program, we can reuse the abstraction of an action in different parts of a program, which can accelerate the convergence in checking valid properties.

We present the technique via a case study based on a translation of a JML annotated Java program into DynAlloy, accompanied by some preliminary experimental results showing some of the benefits of the technique.

## 1 Introduction

The increasing dependability of human activities on software systems is leading us to accept that formal methods, once thought to be worthwhile only for critical systems, are actually applicable and even necessary for a wider class of systems. Indeed, there currently exist many tools and projects attempting to bring together formal methods and widely used (less formal) software development notations and methodologies (e.g., the works reported in [17,18,20], to name a few). Two of the main limitations for using formal methods in practice are that they require mathematically trained developers, and that their application often involves the manual manipulations of mathematical expressions, both

during modelling (specification) and analysis (e.g., by theorem proving). Alloy [15] is a formal method that attempts to partly overcome these limitations. First, it is based on a simple notation, with a simple relational semantics, which resembles the modelling constructs of less formal object oriented notations, and therefore is easier to learn and use for developers without a strong mathematical background. Second, it offers a completely automated SAT based analysis mechanism, so that, in principle, no manual manipulations of mathematical expressions are necessary for analysing specifications. This is done at the expense of losing certainty, since the Alloy tool cannot guarantee the validity of a property, but only its validity in bounded (usually by a rather small bound) models [15]. Basically, given a system specification and a statement about it, the Alloy tool exhaustively searches for a counterexample of this statement (under the assumptions of the system description), by reducing the problem to the satisfiability of a propositional formula. Since the Alloy language is first-order, the exhaustive search for counterexamples has to be performed up to 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, since it cannot be used in general to guarantee the absence of counterexamples for a theory. Nevertheless, this analysis mechanism is very useful in practice, since it allows one to discover counterexamples of intended properties, and if none is found, gain confidence about our specifications. This is similar in spirit to testing, since one checks the truth of a statement for a number of cases; however, as explained in [16], the scope of the technique is much greater than that of testing, since the space of cases examined (usually in the order of billions) is beyond what is covered by testing techniques, and it does not require one to manually provide test cases.

Alloy belongs to the class of the so called *model oriented* formal methods. Specifications in Alloy are described as abstract models of software systems. These models are essentially composed of data domains and relations between these domains, much in the style of schemata for data domains and operations in  $Z$  [21]. As we and other researchers have advocated in the past, this is suitable for building *static* models of software, but it is less appropriate for the dynamics of systems, i.e., for describing executions and their intended properties [9]. This problem has inspired the definition of an extension of Alloy, called DynAlloy [10], that incorporates actions, understood as a general concept associated with state change, and covering composite as well as atomic actions. Actions can be composed as program terms in dynamic logic, i.e., via sequential composition, non deterministic choice, iteration, etc. Moreover, one can provide partial correctness assertions about actions, which the DynAlloy Analyzer then translates into Alloy for their SAT based analysis.

Abstraction is strongly related to simplicity and understandability of models, as well as to their analysability. Usually, models are driven by the first two concerns, i.e., the modeller chooses the level of abstraction in his models trying to faithfully characterise the aspects of software he is interested in, in the most simple way possible. Typically, these models are suitable according to their

understandability, but not the most appropriate with respect to analysis. Indeed, it is generally accepted that abstraction on models is crucial for the successful automated analysis of specifications [3]. In this paper, we are concerned about improving the abstraction of DynAlloy specifications for analysis. We present a technique, which has been implemented in a prototypical tool, that allows us to employ a version of predicate abstraction [12], an abstraction technique successfully used for model checking, on DynAlloy specifications. Because of the context in which we use predicate abstraction, we are able to take advantage of the following improvements:

- Since DynAlloy’s analysis is based on checking partial correctness assertions against programs, we are only interested in the initial and final states of a computation, and therefore we can safely abstract away some intermediate states. We take advantage of this situation via a particular automated use of program atomisation [11]. Notice that this kind of improvement cannot be straightforwardly applied in model checking, since the abstraction of intermediate states can lead to missing violations of safety properties.
- Since DynAlloy’s analysis is inherently bounded, we can safely rely on the sole use of a SAT solver for producing the abstractions, as well as refining these in a counterexample guided way.
- Since DynAlloy’s basic operational unit is the atomic action, which can be used in different parts within a program, we can reuse the abstraction of an action in different parts of a program, which can contribute to accelerating convergence in checking valid properties.

Our presentation will be driven by a model resulting from a translation of Java code. As it will be made clearer later on, this kind of model will enable us to perform some of the transformations required for constructing and refining an abstraction in a more efficient way. It will also enable us to present some preliminary experimental results.

## 2 A Brief Introduction to Alloy and DynAlloy

In this section we present a brief introduction to Alloy and DynAlloy by means of an example. A thorough description of Alloy can be found in [16].

Our case-study involves a program over sets (of characters) represented as acyclic linked lists, without repeated elements. For representing these, we would need a model of lists. In order to specify lists, a data type for the data stored in the lists is necessary. We can then start by indicating the existence of a set (of atoms) for data, which in Alloy is specified using a *signature*:

```
sig Data { }
```

This is a basic signature. We do not assume any special properties regarding the structure of data. We can now specify what constitutes a list. A list consists of a head (which is a node or may be a null reference), and nodes in turn consist of a data value and an attribute `next` relating the current node to the next one in a linked list:

```

sig List {
  head : Node+NullValue
}
sig Node {
  val : Data,
  next : Node+NullValue
}

```

According to the semantics of Alloy, fields `val` and `next` are functional relations from `Node` objects to `Data` objects, and from `Node` objects to `Node` objects (or to a constant `NullValue`), respectively. `NullValue` is a signature representing a constant, namely the null reference, defined in the following way:

```

one sig NullValue { }

```

As the previous definitions show, signatures are used to define data domains and their structure. The attributes of a signature denote *relations*. The dot operator ‘.’ corresponds to relational composition, generalised to  $n$ -ary relations, and having relational image as a special case. So, for example, given a set `L` (not necessarily a singleton) of `Node` atoms, expression `L.next` denotes the relational image of `L` under the relation denoted by `next`. This leads to a relational view of the dot notation that preserves the intuitive navigational reading of dot, as in object orientation.

Using signatures and fields, it is possible to build more complex expressions denoting relations, with the aid of the Alloy operators. Operator  $\sim$  denotes relational transposition,  $*$  denotes reflexive-transitive closure, and  $\hat{\phantom{x}}$  denotes transitive closure of a binary relation. There are also binary operators. Operator  $+$  denotes union,  $\&$  denotes intersection, and dot ( $\cdot$ ) denotes, as we mentioned before, composition of relations. In all cases, the typing must be adequate. We build formulae from expressions. Binary predicate `in` checks for inclusion, while `=` checks for equality. From these (atomic) formulae we define more complex formulae using standard first-order connectives and quantifiers. Negation is denoted by `!`. Conjunction, disjunction and implication are denoted by `&&`, `||` and `=>`, respectively. Finally, quantifications have the form `some a : A |  $\alpha(a)$`  and `all a : A |  $\alpha(a)$` . Formulae can be used as axioms that constrain models, called *facts*. For example, the following fact:

```

fact AcyclicLists {
  all l:List, n:Node | n in l.head.(*next) => n !in n.(^next)
}

```

constrains lists to be acyclic. Formulae can also be used in *assertions*, which are properties to be analysed using the Alloy Analyzer. For instance, the following assertion:

```

assert NextInjective {
  all l: List, n1, n2: Node |
    n1+n2 in l.head.*next && n1 != n2 => n1.next != n2.next
}

```

asserts that, for every pair of nodes in a list, if the nodes are different, then their corresponding ‘next’ nodes are also different. One can also write parameterised formulae, called *predicates*. For example, the following predicate:

```

pred nonEmpty(l: List) {
  l.head != NullValue
}

```

characterises nonempty lists. In order to check an assertion, or ask for models of a predicate, the specifier has to provide *bounds* for the maximum number of elements to be considered for the domains. For instance, the command

```

check NextInjective for 5 but 3 Data, 3 Node

```

checks whether assertion `NextInjective` is true in all possible interpretations with at most five lists, three data items and three nodes. The command

```

run nonEmpty for 1 List but 3 Data, 3 Node

```

asks for models of predicate `nonEmpty` (i.e., nonempty lists) with at most one list, three data items and three nodes. It is also possible to check a formula for an exact number of elements in a domain.

The Alloy Analyzer receives as input an Alloy model and the selection of an assertion to be checked. Using the bounds on the data domains, a clever translation converts the Alloy model and the (negation of the) assertion into a propositional formula. A model (in the mathematical logic sense) of the resulting propositional formula is then sought for, using off-the-shelf SAT solvers. If a model is found by the SAT solver, it is converted back into a model of the Alloy specification that refutes the validity of the assertion in the specification. A similar procedure is employed for retrieving models satisfying predicates.

DynAlloy [10] is an extension of the Alloy specification language for describing state change in a more convenient way (compared to the Alloy approach, which uses predicates to specify state change). DynAlloy incorporates the notion of *atomic action* as a basic mechanism for modifying the state (atomic actions are similar to atomic statements in imperative programming languages). Atomic actions are defined by means of preconditions and postconditions, given as Alloy formulae. For instance, atomic actions for retrieving the first element in a list and for removing the front element from a list (usually called `Head` and `Tail`, respectively) may be specified as follows:

```

act Head(l:List, d:Data)
  pre = { l.head != NullValue }
  post = { d' = (l.head).val }

act Tail(l:List)
  pre = { l.head != NullValue }
  post = { l'.head = (l.head).next }

```

The primed variables `d'` and `l'` in the specification of actions `Head` and `Tail` denote the values of variables `d` and `l` in those states reached *after* the execution of the actions. There is an important 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. Thus, the atomic action `Head` modifies the value of variable `d`, but `l` keeps its initial value.

From atomic actions we can build complex actions, also called *programs*, as follows. If  $\alpha$  is an Alloy formula, then  $\alpha?$  is a test action (akin to the “assert” construct in the Java programming language). The nondeterministic choice between two (not necessarily atomic) actions  $a_1$  and  $a_2$  is denoted by  $a_1 + a_2$ , while their sequential composition is denoted by  $a_1; a_2$ . Finally,  $*$  iterates actions. As is customary, a partial correctness assertion of the form  $\{\alpha\} p \{\beta\}$  is satisfied if, for every state  $e$  that satisfies  $\alpha$ , all the states reachable from  $e$  through program  $p$  satisfy  $\beta$ . For instance, the following is a valid partial correctness assertion for our case study:

```

{ l.head != NullValue }
  Head(l, d) ; Tail(l)
{ (l.head).val = d' and (l.head).next = l'.head }

```

One of the important characteristics of Alloy is that its specifications can be automatically analysed using the Alloy Analyzer. As we explained before, the Alloy Analyzer allows one to automatically verify if a given assertion holds in all interpretations associated with an Alloy model, with the domain sizes being bounded by user provided bounds. DynAlloy specifications are also subject to automated analysis. In [10], we show how DynAlloy specifications can be translated into Alloy specifications, so that we can indirectly analyse DynAlloy specifications using the Alloy Analyzer. In order to do this, the specifier only needs to provide an extra bound, one to be associated with the maximum number of iterations to be considered.

The case study model employed in this article originates from Java code. Our translation from Java to DynAlloy adopts the object model of JAlloy [14] in order to handle complex data. JAlloy translates Java programs directly to Alloy models. The JAlloy model of signatures `List` and `Node` requires just basic signatures (without fields)

```

sig List { }      sig Node { }

```

and fields are defined as binary relations

```

head : List -> one (Node+NullValue)
val   : Node -> one Data
next  : Node -> one (Node+NullValue)

```

The modifier “one” forces these relations to be total functions. They can be modified by the DynAlloy actions. An action `SetNext`, modelling the update of the value of attribute `next` for a given node, can now be specified as follows:

```

act SetNext(n1,n2:Node+NullValue,next:Node->one(Node+NullValue))
  pre = { n1 != NullValue }
  post = { next' = next ++ (n1 -> n2) }

```

where `++` is relational overriding.

```

/*@ private invariant
  @ (\forall Node n; \reach(this.head).has(n); !\reach(n.next).has(n));
  @*/

/*@ public normal_behavior
  @ assignable theSet;
  @ ensures this.theSet.equals(\old(this.theSet).difference(s.theSet));
  @ also
  @ private normal_behavior
  @ assignable head;
  @*/
public void removeAll(CharSet s) {
  if (this.head != null) {
    Node current = this.head;
    Node prev = null;
    while (current!=null) {
      if (s.isMember(current.value)) {
        if (prev!=null) prev.next = current.next;
        else this.head = current.next;
      } else prev = current;
      current = current.next;
    } } }

```

Fig. 1. JML-Annotated code to be analysed in this article

### 3 From JML-Annotated Java Code to DynAlloy

Finding the right case study for analysing our technique is a difficult task. Seeking for such an appropriate case study, which would allow for a controlled increase of code size, and to check some non trivial properties of the code under consideration, we decided our case study to be a Java program solving the following simple problem:

Given a linked list  $l$  (holding characters as information) and a set  $S$  of characters as input, remove from  $l$  all nodes holding elements in  $S$ .

The actual code, including the corresponding JML annotations, is provided in Fig. 1. This program, although simple, is in our opinion fairly adequate, since the size of code can be increased in a controlled way by unrolling the loop required for traversing the list as many times as deemed appropriate. Moreover, it also allows us to check properties such as that the representation invariant, saying that lists are acyclic, is preserved by this program, as well as checking other related properties, such as that the elements removed are no longer part of the list. Notice that expressing the former (see Fig. 1) requires quantification and reachability predicates, which are constructs hard to analyse for most analysis techniques.

In this section we will provide some details regarding how the translation from annotated Java code to DynAlloy is performed, using parts of our case

study. A more thorough description can be found in [8,13]. Our program involves statements for assignment and attribute modification. These are modelled as atomic DynAlloy actions, in the following way:

```

act assign(l:A, r:A)      act SetF(l:A, r:B, F:A->one B)
  pre { true }           pre { l != NullValue }
  post { l' = r }        post { F'=F++(l->r) }

```

In the definition of action `SetF`, the binary relation `F` gets modified by the action. Complex programs are translated as follows:

```

P1 ; P2                ->   T(P1);T(P2)
if (C) then P1 else P2 ->   (C'?;(P1))+(!C'?;(P2))
while (C) P             ->   (C'?;P)*;!C'?

```

where `C'` is the Alloy translation of predicate `C`. JML assertions are mapped to Alloy formulae. For instance, the first representation invariant in our case study, which constrains lists to be acyclic structures, is translated into the following Alloy formula:

$$\text{all } n : \text{Node } n \mid n \text{ in this.head.}(*\text{next}) \Rightarrow !(n \text{ in } n.^{\wedge}\text{next}). \quad (1)$$

This translation is completely automated.

If formula (1) is denoted by `NoCycle(this, head, next, val)`, and `P` is the DynAlloy program obtained from our case study (see Fig. 2), the problem to solve is expressed as the following partial correctness assertion, that we will call `NoCyclePreserved`:

$$\{ \text{NoCycle}(\text{this}, \text{head}, \text{next}, \text{val}) \} \\ P \\ \{ \text{NoCycle}(\text{this}', \text{head}', \text{next}', \text{val}') \}$$

In order to make the analysis simpler, we will make use of the following DynAlloy atomic action (whose correctness should be checked at a later stage against some implementation):

```

act isMember(result:boolean, s:set Char, c:Char)
  pre { true }
  post { result' = true <=> c in s }

```

## 4 SAT-Based Predicate Abstraction for DynAlloy Models

We now present the mechanism employed in order to abstract DynAlloy specifications. As we mentioned, the mechanism is based on predicate abstraction and counterexample guided abstraction refinement. We will assume that the reader has some basic acquaintance with the subject as presented in [12,6]. Briefly, standard predicate abstraction works as follows. Given a transition system  $P = \langle S, \text{Init}, \tau \rangle$ , where  $S$  is the set of states,  $\text{Init}$  a formula characterising

```

act RemoveAll(this: List, curr, prev: Node+NullValue, S: set Char,
              value: Node -> one Char, next: Node -> one (Node+NullValue),
              head: List -> one (Node+NullValue))
01. (this.head != NullValue)?;
02. ( assign(prev, NullValue);
03.   assign(curr, this.head);
04.   ( (curr != NullValue)?;
05.     ( ( (
06.         (curr.value in S)?;
07.         ( (prev != NullValue)?;
08.           setNext(prev, curr.next, next)
09.           +
10.           (prev = NullValue)?;
11.           setHead(thisValue, curr.next, head)
12.         ) )
13.         +
14.         ( (curr.value !in S)?;
15.           assign(prev, curr)
16.         ) )
17.       assign(curr, curr.next)
18.     )
19.   )*)
20. (curr = NullValue)?
21. )
22. +
23. ( (this.head = NullValue)?;
24.   skip
25. )

```

**Fig. 2.** DynAlloy specification corresponding to program `removeAll`

the set of initial states, and  $\tau$  a set of transitions (i.e., binary relations over  $S$ ), one starts by providing some predicates  $\phi_1, \phi_2, \dots, \phi_n$  over  $S$ . The main idea is to consider an abstraction  $Q^A$  of the lattice  $\wp(S)$  of state properties over  $S$ , together with two functions  $\alpha : \wp(S) \rightarrow Q^A$  and  $\gamma : Q^A \rightarrow \wp(S)$ , relating  $Q^A$  and  $\wp(S)$  in such a way that  $\alpha(\gamma(Q^A)) = Q^A$  and, for every  $s \in \wp(S)$ ,  $s \subseteq \gamma(\alpha(s))$ . That is, the pair  $\langle \alpha, \gamma \rangle$  forms a *Galois connection* between  $Q^A$  and  $\wp(S)$ . In predicate abstraction,  $Q^A$  has a particular form, it is composed by the monomials over  $n$  boolean variables  $B_1, B_2, \dots, B_n$  representing the truth values of  $\phi_1, \phi_2, \dots, \phi_n$ , respectively; a monomial is either *true* or *false*, or a conjunction of literals ( $B_i$  or  $\neg B_i$ ) in which each  $B_i$  appears at most once (positively or negatively). This set clearly forms a lattice, where the atoms (which represent the abstract states) are the canonical monomials, i.e., the monomials in which each  $B_i$  appears exactly once. The concretisation function  $\gamma : Q^A \rightarrow \wp(S)$  is simply defined as  $\gamma(s^A) = \{s \in S \mid s \models s^A[\phi_i/B_i]\}$  whereas the abstraction function is given by:

$$\alpha(s) = \bigwedge_{i \in 1..n} \{B_i \mid s \models \phi_i\} \wedge \bigwedge_{i \in 1..n} \{\neg B_i \mid s \models \neg \phi_i\}$$

As explained in [12], this results in a more efficient way of calculating the abstract model from a concrete one.

We would like to provide the above described abstraction mechanism for DynAlloy specifications. As we mentioned, atomic actions are the basic mechanism for characterising state change in DynAlloy, and typically have the following form:

```
act a(s: State)
  pre { pre(s) }
  post { post(s,s') }
```

for some designated state signature **State**. In order to apply predicate abstraction, we need a number of predicates  $\phi_1(s), \phi_2(s), \dots, \phi_n(s)$  over the state signature **State**. In our case, we consider as an initial set of abstraction predicates the individual conjuncts in the postcondition of the assertion, and the conditions extracted from the source code of the program (which appear in test actions in the DynAlloy translation):

```
this.head != NullValue   curr != NullValue
curr.value in S           prev != NullValue
```

Now let us describe how the abstract DynAlloy program is represented. Since in this case we have five predicates (the above four plus the postcondition of the assertion), we can characterise the abstract state space by the following **AState** Alloy signature:

```
sig AState { p0, p1, p2, p3, p4 : Boolean }
```

The idea behind our characterisation of the abstract DynAlloy program is, as the reader might expect, that a particular atom of signature **AState** will represent exactly one abstract state. It is easy to see what are the concrete states associated with an abstract state **s**: those that satisfy exactly those  $\phi_i$ 's for which **s.pi** is **true**.

We now need to compute abstract actions characterising the abstract behaviour of each of the concrete atomic actions. Let us first consider atomic actions in isolation. Abstracting the (concrete) precondition *pre* of a given action *a* is not difficult. We need to decide which are the corresponding elements of the abstract lattice  $Q^A$ , i.e., the monomials, better characterising *pre*. This can be done simply by checking which of the  $\phi_i$ 's and  $\neg\phi_i$ 's are implied by *pre*. For postconditions, on the other hand, the process is slightly more complicated. The reason is that, as it is shown in the actions for our DynAlloy program, the postconditions are not state formulae, but *relations* describing how the states previous to the execution of the actions are related to the corresponding states after the execution of the actions. Thus, what we actually need to check, for an atomic action with precondition **pre(s)** and postcondition **post(s,s')**, is the abstract state corresponding to the *strongest postcondition* of **pre(s)** according to **post(s,s')**. We use the Alloy Analyzer in order to check these assertions.

The process just described for computing the abstraction of atomic actions, although correct, generally leaves us with too coarse abstractions, which would produce an important number of spurious counterexamples when checking the abstract program, even when the property being checked is invalid. The reason for this is that this kind of abstraction only takes into account the precondition of the action, and not the information regarding the context in which the action is used. We will use these abstractions as a starting point, and will compute the abstractions using the following more sophisticated approach.

Suppose that we have to check a DynAlloy assertion of the form:

$$\{ \text{pre\_a} \} \quad P \quad \{ \text{post\_a} \}$$

Notice that every DynAlloy assertion check needs two different bounds, one limiting the size of the domains, and another one bounding loops. Let us consider these to be  $k_d$  and  $k_l$ , respectively. We will start by unrolling the loops in  $P$  according to bound  $k_l$ , thus obtaining a sequential program  $P_S$ , without loops. Our abstraction process will consist of computing an abstract version of  $P_S$ 's control graph. We will consider, initially, basic abstractions for all atomic actions computed as described above, in terms of their corresponding pre and post conditions and using  $k_d$  as a bound. Also, we will compute the abstraction of the precondition  $\text{pre\_a}$  of the assertion, also using  $k_d$  as a bound. We will then start visiting  $P_S$ 's computation tree in a depth-first fashion; so, in each step we will choose either a test action (which can be abstracted straightforwardly, since its associated condition is among the abstraction predicates) or an atomic action  $a$ . For this, we check whether, for its current abstract precondition (which is not necessarily a canonical monomial), we have already computed its corresponding abstract postcondition. If not, we concretise the current precondition, and compute the abstraction of the corresponding concrete strongest postcondition (for these checks, we also use  $k_d$  as a bound). Since  $P_S$ 's computation tree is finite and acyclic (due to the absence of loops, which we have previously unrolled), this process is guaranteed to terminate. If we reach a final abstract state (a leaf in the computation tree for the abstract version of  $P_S$ ) in which the abstraction of the postcondition is not satisfied, then we found an abstract counterexample. Notice that the postcondition of the assertion is precisely characterised in the abstract program, since it is included in the set of abstraction predicates. If no abstract counterexample trace is found, then the property has been checked valid within the established bounds.

If an abstract counterexample is found, then we have to check whether it is a spurious one or not. If it is not spurious, the property is invalid, and we find a counterexample as the concretisation of the obtained abstract trace. If, on the other hand, it is spurious, we employ a traditional counterexample guided abstraction refinement as presented in [7]. We find the abstract state in which the spurious counterexample “breaks” (i.e., where it cannot be further concretised), and employ the predicate discovery approach as described in [7].

Let us summarise this process of abstraction. It is composed of the following steps:

1. Take program  $P$  and unroll the loops in it according to bound  $k_l$ , obtaining as a result a sequential program  $P_S$  (both  $P$  and  $P_S$  are concrete).
2. Compute the abstractions for `pre_a` and `post_a`, and for each of the atomic actions, using the set of abstraction predicates available (initially, these are the conjuncts of the postcondition in the concrete assertion and the conditions in the program).
3. Start the verification process by visiting the computation tree for  $P_S$  and computing the corresponding abstract states along the traversal. Here, the abstractions of the actions are used, and more detailed abstract pre- and post-conditions are computed for their definitions when not previously considered abstract preconditions are found).
4. When a final abstract state (a leaf in the computation tree) is reached, there are two possibilities.
  - (a) The abstract state satisfies the abstraction of the postcondition. In this case, we continue the visit of the tree. If the whole computation tree has been visited, then the property has been successfully checked (for the given bounds).
  - (b) The abstract state violates the abstract postcondition. In this case, we have found an abstract counterexample. We concretise the corresponding trace, and if it corresponds to a concrete counterexample, the property being checked is invalid. If not, the counterexample is spurious, and is used to calculate a new abstraction predicate. We incorporate the new predicate to our set of abstraction predicates, and go back to step 2.

It is important to notice how the above described abstraction mechanism allows us to improve analysability. In the straightforward approach (without abstraction), two variables make the size of the propositional formula resulting from the DynAlloy assertion to be checked grow, namely  $k_d$  and  $k_l$ . Usually, these two variables in combination make the formula too big to be handled with the available resources. Using the above described abstraction mechanism, the analysis is split into various checks in order to build the abstraction, each of which is only affected by the  $k_d$  bound. The only checkings affected by both bounds are the ones corresponding to the concretisation of abstract counterexamples. However, these are generally much simpler than checking the original program, since the trace corresponding to an abstract counterexample is only a sequential program with no branching nor loops (branching has an important negative impact in checking DynAlloy's assertions).

We present below some experimental results regarding the application of the abstraction mechanism just described, in comparison with the straightforward (no abstraction) SAT analysis. However, the described (traditional) abstraction mechanism is not sufficient, and we will need to perform some optimisations in order to gain an acceptable performance for the abstraction based analyses. These optimisations are described in the next section.

## 5 Improving the Abstraction Based Analysis

In this section we present a few optimisations that we applied to the above described traditional abstraction approach. The contribution, in terms of performance in the analysis, that these optimisations provided are reported later on, in the section on experimental results.

### 5.1 Program Atomisation

Program atomisation is an abstraction technique for DynAlloy programs that allows us to contribute to scaling analysability up by replacing (arbitrary) complex programs by atomic actions with the same behaviour. The analysis improves because the SAT-solver does not need to look for intermediate valid states matching the program behaviour. Generally, the use of program atomisation is not fully automated, although the way in which we will use it here, a restricted form, enables us to fully automate it. In the context of this article, atomisation will have a great impact, because the removal of intermediate states favours abstraction: intermediate states (when these are temporary) typically cause breaks in spurious counterexamples, and possibly the introduction of further abstraction predicates characterising the corresponding intermediate (temporary) state situations. We will automatically perform atomisation in order to abstract away certain intermediate states, via the following atomisation policy:

Consider, as a program to be atomised, any maximal sub-path of the control flow graph that does not involve tests.

Notice that program atomisation is applied *before* the program is unrolled according to the bound on iteration.

We still have to provide a method for automatically building the atomisations. Let us consider atomic actions **A1** and **A2** specified as follows:

$$\begin{array}{cc} \{ Q(s) \} & \{ S(s) \} \\ \text{A1}(s) & \text{A2}(s) \\ \{ R(s, s') \} & \{ T(s, s') \} \end{array}$$

The atomisation of the sequential composition **A1** ; **A2** is defined as the atomic action **A** specified by:

$$\begin{array}{c} \{ Q(s) \ \&\& \ (\text{all } st \mid R(s, st) \Rightarrow S(st)) \} \\ \text{A} \\ \{ \text{some } i \mid R(s, i) \ \&\& \ T(i, s') \} \end{array}$$

Notice that **A**'s precondition characterises exactly those states that satisfy the precondition of **A1**, and upon execution of **A1** lead only to states satisfying **A2**'s precondition. On the other hand, the postcondition clearly models the sequential composition of the behaviours of **A1** and **A2**. We are only defining the atomisation for the sequential composition of two actions. If more actions are to be atomised, the process can be iterated. It is straightforward to prove that given actions

```

act atomAssignPrevCurr(prev,prevVal,curr,currVal:Node+NullValue)
  pre = { true }
  post = { prev' = prevVal && curr' = currVal }

act atomSetNextAssCurr(l,r: Node+NullValue,next: Node->one(Node+NullValue),
  curr,currVal: Node+NullValue)
  pre = { l != NullValue }
  post = { next' = next++(l->r) && curr' = currVal }

```

**Fig. 3.** Atomisations for program removeAll

A1, A2, A3, atomising first A1 and A2 (and the result with A3) yields an action equivalent to the one resulting from first atomising A2 and A3. Moreover, in order to simplify the resulting atomisation, notice that:

1. Whenever A2's precondition is **true**, A's precondition reduces to  $Q(s)$ .
2. If A1 and A2 modify disjoint sets of state variables, we can proceed as follows. Let the state variables  $s = S_0 \cup S_1 \cup S_2$ , with  $S_0, S_1, S_2$  disjoint, and such that A1 modifies  $S_1$  and A2 modifies  $S_2$ . The postcondition then simplifies to

$$R(S_0 \cup S_1 \cup S_2, S_0 \cup S'_1 \cup S_2) \ \&\& \ T(S_0 \cup S'_1 \cup S_2, S_0 \cup S'_1 \cup S'_2).$$

We perform atomisation at the very beginning, and include simplifications as the ones mentioned to be applied on the resulting atomisations. For our case-study, the original DynAlloy program (see Fig. 2) is atomised using the actions in Fig. 3.

## 5.2 Detection of Induction

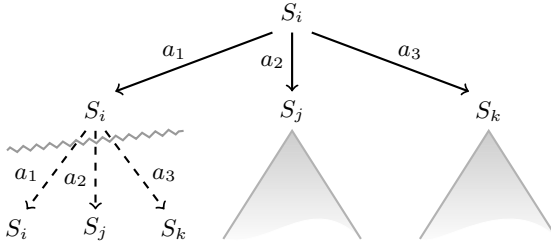
Consider cases in which the program we are trying to check is of the form:

$$P = \text{Init}; (a_0 + \dots + a_n)*$$

with each  $a_i$  not necessarily atomic. Moreover, suppose that we check the property under consideration *incrementally*, i.e., we check the property for  $k_l$  only after we checked it for all loop bounds smaller than  $k_l$ . In these cases, we can consider the following rule in order to “prune” the construction of the computation tree corresponding to  $P_S$  in the verification and abstraction process:

if  $s$  is the current abstract state (at some point after initialisation), and the abstraction of  $a_i$  leaves us again in the abstract state  $s$ , then we can stop building the abstract graph for  $P_S$  after the last  $s$ , because the abstract states resulting from the last  $s$  will necessarily be visited as branches of the first  $s$ .

The idea behind this rule is graphically depicted in Figure 4. The fact that this rule is applied only in incremental verifications is crucial, since it corresponds



**Fig. 4.** Graphical description of the pruning rule for detecting convergence in invariant checks.

essentially to performing a kind of iterative deepening visit of the abstract graph, looking for violations of the property (then we only advance in the process if we have not found shorter violations). Also, the fact that atomic actions are repeated in the graph, so that we can use the abstractions produced for these in other parts of the graph, is relevant for this rule. Notice also that, given that we apply program atomisation prior to performing the loop unrolls, it is guaranteed that we are not abstracting away final states, only states which are intermediate within loops.

The case in which we are checking whether the program preserves the representation invariant of sets over linked lists, i.e., whether it leaves the resulting list acyclic and with no repeated elements, corresponds to the above described schema. As we describe below, in the section regarding experimental results, this rule allowed us to find a convergence of the representation invariant property after four loop unrolls. The reason for this is that all abstract final states after four loop unrolls are “already visited” states in the computation tree, enabling us to infer that further loop unrolls cannot lead to violations of the property.

## 6 Experimental Results

For running the experiments we used a personal computer with a 2Ghz Core 2 Duo processor, with 2GB of RAM, running the Alloy Analyzer 4.1.5 under Ubuntu GNU/Linux 2.6.22, 32 bits.

We attempted to verify the following assertions regarding our case study:

`NoCyclePreserved:`

```
all n: Node |
    n in thisV.headV.(*nextV) => n !in n.nextV.(*nextV)
```

`ElementsRemoved:`

```
all n: Node |
    n in (thisV.headV.(*nextV) - (currV.(*nextV))) =>
        n.valueV !in (c1+c2+c3)
```

The first of these, `NoCyclePreserved`, affirms that the list is acyclic, while the second one, `ElementsRemoved`, in combination with the fact that `curr` is null at the end of the algorithm, ensures that the characters in the set to be removed, passed as a parameter, are indeed removed from the list.

Without the use of abstraction, if we unroll the only loop in the program 23 times the analysis of assertion `NoCyclePreserved` exhausts the available memory, causing a run time crash of the Alloy Analyzer (the corresponding CNF formula had 620841 variables, 30768 of which are primary, and a total of 1339236 clauses). In this case, the scopes for signatures `Char` and `Node` were set to 24. Also, if we unroll the loop 51 times, a runtime exception is thrown by the Alloy Analyzer due to insufficient memory to construct the CNF formula to be analysed.

Using the traditional predicate abstraction mechanism described, without the further optimisations, 44 seconds of SAT analysis time were required for our tool to check the property `NoCyclePreserved`. Two new predicates were introduced during the process. Thanks to the fact that our method can take advantage of already calculated abstract actions, performing SAT Solving was only necessary for the first four loop unrolls. For the same assertion, but after applying program atomisation to the original specification, the SAT time was reduced by almost 50%: 25 seconds. The number of predicates and loop unrolls required to discover new abstract actions, though, remained the same. The above described induction detection mechanism allows us to reduce the number of nodes in the abstract execution tree to be visited. Notice that, given a certain number  $k$  of loop unrolls, the size of the abstract execution tree for our program is exponential with respect to  $k$ . More precisely, in the worst case (when the whole tree has to be visited) we need to visit  $1 + \sum_{i=1}^n 2^i + \sum_{i=1}^{n-1} 2^i$  nodes for the atomised program and  $1 + \sum_{i=1}^n 2^i + 2 * \sum_{i=1}^{n-1} 2^i$  nodes for the non atomised case. For instance, if we unroll the loop in the program four times for the atomised case, the abstract tree will be composed of 45 nodes. On the other hand, by employing induction detection the procedure can be terminated after visiting 19 nodes, for the assertion `NoCyclePreserved`. On the other hand, the abstract tree for the nonatomised program, with the loop unrolled four times, will have 59 nodes, and using induction detection our algorithm visited only 30 of them. Furthermore, induction detection allows us to conclude that we can stop the search for abstract counterexamples (of any size), since, as it was discussed in section 5.2, no new states violating the property can appear. As it is shown by this example, this technique can reduce the search state space considerably.

On the other hand, if we want to verify the property `ElementsRemoved` using again the traditional abstraction mechanism, the abstraction process diverges due to excessive introduction of abstraction predicates. Nevertheless, using program atomisation on the model, and introducing the right auxiliary invariants (`prev.next` equals `curr`, `prev` and `curr` are reachable from the head of the list, and the list is acyclic) as abstraction predicates, the process converges again very quickly (9 seconds of SAT time). New atomic actions appeared when verifying the property up to the third iteration, and 13 nodes of the (atomised) abstract computation tree were visited by our algorithm. A total of 8 abstraction

predicates were necessary. It is worth mentioning that `NoCyclesPreserved` was used as an abstract invariant (i.e., a predicate that is valid before and after execution of each abstract action). We did so safely because `NoCyclesPreserved` was previously verified to be a concrete invariant. Notice also that, since the program that we are considering is the same as for the previous case, the number of nodes of the execution tree can be calculated using the same formula (there are 18 nodes in the graph for 3 loop unrolls).

Of course, obtaining the above mentioned auxiliary predicates automatically would require the use of invariant generation techniques. We used the “global invariants” generation mechanism of the STeP tool [2], but in order to do this we had to produce an ad hoc representation of the program in STeP’s language SPL. Clearly, this requires further investigation, and is part of our future work.

An important fact to mention regarding the experiments is that, since the DynAlloy specification originates from code, atomic actions essentially correspond to assignments, and therefore these are “invertible”, making the calculation of weakest preconditions in sequential programs (when examining abstract counterexamples) very efficient. It is expected that this efficiency will not be preserved when considering other kinds of DynAlloy specifications.

## 7 Related Work

There exist many tools and approaches applying ideas of predicate abstraction for formal verification, such as for instance the work reported in [6,7]. While our approach is strongly based on Das and Dill’s technique for predicate abstraction, it differs from existing tools in that ours is tailored to SAT-solving through Alloy and DynAlloy, as opposed to most other tools, whose associated verification technique is model checking. In [4] SAT-solving is used to construct the abstraction, but the more conventional techniques based on symbolic model checking are used in the remaining parts of the process. With respect to abstraction in the context of Alloy and DynAlloy specifications, the most closely related approaches we know of are the work of Taghdiri [19] on the use of predicate abstraction in JAlloy analysis, and the work of some of the authors of this paper on program atomisation [11]. The work of Taghdiri is different from ours since her abstraction mechanism faithfully represents the code of a Java program, except for the method calls, where abstraction is applied. The abstractions obtained for procedure calls are somehow “reusable” (they can be used for other places in the program where the procedure is also called), as in our approach; however, her mechanism for discovering new predicates is completely different, since in her case the spurious counterexamples are not really abstract runs, but concrete ones where the abstraction is present only in the form of underspecified effects for procedure calls. SATURN [22] is, as our tool, completely based on SAT-solving. The techniques it uses to improve analysability are: program slicing (at the intra-procedural level) and a kind of abstraction called *function summaries* to modularise the analysis at the inter-procedural level. SATURN models programs faithfully (no abstraction is performed).

## 8 Conclusions and Future Work

We have investigated the application of predicate abstraction for improving the analysis of DynAlloy specifications. We have concentrated in a particular kind of DynAlloy specifications, namely those resulting from a translation from annotated Java code. We plan to exploit the experiences gained in providing abstraction for this kind of DynAlloy specifications in order to provide a mechanism applicable to a wider class of these. We exploited the principal analysis technique associated with Alloy and DynAlloy specifications, SAT based analysis, in order to build, analyse and improve the abstractions in an automated way, based on Das & Dill's algorithm for abstraction refinement [6], complemented primarily with an automated program atomisation mechanism. We have also provided a mechanism for detecting convergence in the unrolling process for a certain schema of DynAlloy programs.

The results of the experiments conducted, based on a case study over a model originated from a JML annotated Java program, show that the improvements performed to the traditional predicate abstraction mechanism had an important impact for one of the properties we checked. The reason is that these constituted an economy in abstraction predicates to be introduced during verification, as well as in the size of the formulae analysed when examining abstract counterexamples for abstraction refinements. However, this is just an initial attempt, and it is clear that we need to develop more case studies.

There are several directions for future work. We are planning to experiment with the use of automated theorem provers to attempt to simplify the formulae introduced in the process of improving abstractions, for eliminating unnecessary quantifiers, amongst other things. Invariant generation techniques, integrated in the approach, will have in our opinion an important positive impact, so this is one of the directions we want to explore. We are also planning to incorporate a differentiated treatment for mutant and non mutant objects in the DynAlloy specifications, in order to make the construction of the abstractions more efficient.

It is also important to mention that our approach corresponds only to intra-procedural analysis. We plan to study the combination of the presented approach with Taghdiri's abstraction for Alloy [19].

## References

1. Ball, T., Cook, B., Das, S., Rajamani, S.: Refining Approximations in Software Predicate Abstraction. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988. Springer, Heidelberg (2004)
2. Bjorner, N., Browne, A., Colon, M., Finkbeiner, B., Manna, Z., Sipma, H., Uribe, T.: Verifying Temporal Properties of Reactive Systems: a STeP Tutorial. Formal Methods in System Design 16 (2000)
3. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems (TOPLAS) 16(5) (1994)

4. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate Abstraction of ANSIC Programs using SAT, Technical Report CMU-CS-03-186. Carnegie Mellon University (2003)
5. Cousot, P.: Abstract interpretation. *ACM Computing Surveys* 28(2) (1996)
6. Das, S., Dill, D.: Successive Approximation of Abstract Transition Relations. In: *Proceedings of the IEEE Symposium on Logic in Computer Science LICS 2001*. IEEE Computer Society Press, Los Alamitos (2001)
7. Das, S., Dill, D.: Counterexample Based Predicate Discovery in Predicate Abstraction. In: Aagaard, M.D., O’Leary, J.W. (eds.) *FMCAD 2002*. LNCS, vol. 2517. Springer, Heidelberg (2002)
8. Dennis, G., Chang, F., Jackson, D.: Modular Verification of Code with SAT. In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, pp. 109–120 (2006)*
9. Frias, M., López Pombo, C., Baum, G., Aguirre, N., Maibaum, T.: Reasoning about static and dynamic properties in alloy: A purely relational approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14(4) (2005)
10. Frias, M., Galeotti, J.P., López Pombo, C., Aguirre, N.: DynAlloy: upgrading alloy with actions. In: *Proceedings of the 27th International Conference on Software Engineering ICSE 2005*. ACM Press, New York (2005)
11. Frias, M., Galeotti, J.P., López Pombo, C., Aguirre, N.: Efficient Analysis of DynAlloy Specifications. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*. ACM Press, New York
12. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) *CAV 1997*. LNCS, vol. 1254. Springer, Heidelberg (1997)
13. Galeotti, J.P., Frias, M.F.: DynAlloy as a Formal Method for the Analysis of Java Programs. In: *Proceedings of IFIP Working Conference on Software Engineering Techniques (SET 2006)*, Warsaw. Springer, Heidelberg (2006)
14. Jackson, D., Vaziri, M.: Finding Bugs with a Constraint Solver. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, USA, August 21-24, pp. 14–25. ACM Press, New York (2000)
15. Jackson, D.: Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (ACM TOSEM)* 11(2) (2002)
16. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge (2006)
17. Kim, S.-K., Carrington, D.: Formalizing the UML Class Diagram Using Object-Z. In: France, R.B., Rumpe, B. (eds.) *UML 1999*. LNCS, vol. 1723. Springer, Heidelberg (1999)
18. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15(1) (2006)
19. Taghdiri, M.: Inferring Specifications to Detect Errors in Code. In: *Proceedings of the 19th International Conference on Automated Software Engineering ASE 2004, Austria (September 2004)*
20. Uchitel, S., Chatley, R., Kramer, J., Magee, J.: LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios. In: Gavel, H., Hatcliff, J. (eds.) *TACAS 2003*. LNCS, vol. 2619. Springer, Heidelberg (2003)
21. Woodcock, J., Davies, J.: *Using Z: Specification, Refinement and Proof*. Prentice-Hall, Englewood Cliffs (1996)
22. Xie, Y., Aiken, A.: Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability. *ACM-Transactions on Programming Languages and Systems (TOPLAS)* (to appear)