

Faster SAT-Based Analysis of OO-Programs by Separation of Mutant and Non Mutant Objects

Marcelo F. Frias

Department of Computer Science
FCEyN - Universidad de Buenos Aires
and CONICET
Argentina
e-mail:mfrias@dc.uba.ar

Juan P. Galeotti

Department of Computer Science
FCEyN - Universidad de Buenos Aires
Argentina
e-mail: jgaleotti@dc.uba.ar

Abstract

In this article we present an optimization to the analysis of DynAlloy programs based on the distinction between mutant objects (those whose state is modified by the program) and non mutant objects. This allows us to consider smaller scopes and therefore to reduce the analysis time. The theoretical observation that the technique reduces the state space exponentially is supported by examples where a striking analysis time reduction can be observed.

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

Terms Languages, Design, Verification.

Keywords Alloy, DynAlloy, static analysis, software verification.

1 Introduction

DynAlloy [2] is an extension of the Alloy modeling language [5] that incorporates *actions*. Alloy specification constructs are static (i.e., they constrain a single state). Actions, on the other hand, describe evolutions of the state. They can be thought of as imperative programs whose behavior is specified by means of preconditions and postconditions written in Alloy.

While we have already developed a tool (DynJML [4]) that allows one to translate JML-annotated Java code to DynAlloy (see [10] as a reference about JML), in order to keep the article more focused we will use DynAlloy as a programming language, and Alloy instead of JML. The DynAlloy Analyzer allows one to automatically analyze Dy-

nAlloy specifications, with the aid of the Alloy Analyzer [9]. Given a DynAlloy specification to be analyzed, and an assertion presumably valid in the specification, the DynAlloy Analyzer looks for a model (in the sense of mathematical logic) of the specification, that falsifies the assertion. The existence of such model, called a *counterexample*, may expose a flaw in the specification. In effect, we can test the correctness of the specification by looking for counterexamples of assertions that should hold in it. If a counterexample is found for a given assertion (and the assertion correctly models a property that should hold in the specification), the specification must be incorrect. This analysis technique is inherited by the use we make of the Alloy Analyzer [9]. The DynAlloy Analyzer translates DynAlloy specifications to Alloy, and then the Alloy Analyzer appeals to different SAT-solvers in order to look for counterexamples.

Notice that this analysis procedure is particularly well suited for the analysis of contracts or invariants. In effect, if we want to verify if a contract is not fulfilled, then it suffices to find instances that satisfy the precondition of a method, yet falsify the postcondition. This can be easily put in the form of a DynAlloy assertion. Similarly, a formula fails to be invariant if it is not preserved by some method, i.e., if there are object instances that satisfy the formula prior to the execution of the method, but whose state is modified by the method in a way that falsifies the formula. Again, this problem can be easily modeled using DynAlloy assertions.

The analyzability of Alloy and DynAlloy models heavily depends on two factors:

1. The size of the formula that holds the assertion to be analyzed (large formulas many times cannot be compiled by the Alloy Analyzer),
2. The size of the scopes, since usually increasing a scope results in an exponential increase in the analysis time.

In this article we face these two problems, and provide techniques that allow us to obtain important improvements both in compilability and analyzability.

The paper is organized as follows. In Section 2 we introduce DynAlloy and the DynAlloy Analyzer as extensions of Alloy and the Alloy Analyzer, respectively. In Section 3 we present the optimizations to the analysis of DynAlloy models. In Section 4 we present several examples. Finally, in Section 5 we draw some conclusions.

2 The DynAlloy Modeling Language

DynAlloy is an extension of the Alloy modeling language. It allows us to define atomic actions that modify the state (much the same as atomic statements in imperative programming languages do), and build more complex actions (programs) from the atomic ones. Atomic actions are defined by means of preconditions and postconditions given as Alloy formulas. For instance, atomic actions that retrieve the first element in a list, or remove the front element from a list (usually called Head and Tail, respectively) are specified by

<pre>act Head(l : List, d : Data) pre = { l != Empty } post = { d' = l.val }</pre>	<pre>act Tail(l : List) pre = { l != Empty } post = { l' = l.next }</pre>
--	---

The primed variables d' and l' in the specification of actions Head and Tail denote the value of variables d and l in those states reached *after* the execution of the actions. 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. Thus, the atomic action *Head* modifies the value of variable d , but l keeps its initial value. This sort of frame condition allows us to use simpler formulas in preconditions and postconditions.

Equally important, DynAlloy allows us to assert properties about complex actions by means of partial correctness assertions. For instance, the following is a valid assertion:

$$\{ l \neq \text{Empty} \} \\ \text{Head}(l, d); \\ \text{Tail}(l) \\ \{ \text{Cons}(d', l', l) \}$$

The syntax of DynAlloy's formulas extends the one for Alloy formulas with the addition of the following clause for building partial correctness statements:

$$\text{formula} ::= \dots \mid \{ \text{formula} \} \text{program} \{ \text{formula} \} \\ \text{“partial correctness”}$$

Figure 1 shows how complex actions are built from atomic ones. Figure 2 describes the semantics of DynAlloy.

$act ::=$	$p\{pre(\bar{x})\}\{post(\bar{x})\}$	“atomic action”
	$formula?$	“test”
	$act + act$	“non-det. choice”
	$act; act$	“seq. composition”
	act^*	“iteration”

Figure 1. Grammar for DynAlloy's Actions

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

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

Figure 2. Semantics of DynAlloy.

One of the important features of Alloy is the automatic analysis possibilities it provides. In effect, the Alloy Analyzer [9] allows us to automatically verify if a given assertion holds in an Alloy model. Similarly, in [3] we show how to translate DynAlloy specifications to Alloy specifications in order to achieve analyzability. We reproduce the fundamental aspects of this translation below, and refer the reader to [3] or Section 3 for optimizations.

We define below a function

$$wlp : \text{program} \times \text{formula} \rightarrow \text{formula}$$

that computes the weakest liberal precondition [1] of a formula according to a program (composite action). We will in general use names x_1, x_2, \dots for program variables, and will use names x'_1, x'_2, \dots 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

$$a\{pre(\bar{x})\}\{post(\bar{x}, \bar{x}')\}$$

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, \bar{y} . In this situation, function wlp is defined as follows:

$$\begin{aligned} wlp[a(\bar{y}), f] &= \\ pre\{\bar{y}\}_{\bar{x}} &\implies \text{all } \bar{n} \left(post\{\bar{n}\}_{\bar{x}'\bar{y}'} \implies f\{\bar{n}\}_{\bar{y}'} \right). \quad (1) \end{aligned}$$

A few points need to be explained about (1). First, we assume that free variables in f are amongst \bar{y}' , \bar{x}_0 . Variables in \bar{x}_0 are generated by the translation function $pcat$ given in (3). Second, \bar{n} is an array of new variables, one for each variable modified by the action. Last, notice that the resulting formula has again its free variables amongst \bar{y}' , \bar{x}_0 . This is also preserved in the remaining cases in the definition of function wlp .

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

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

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:

$$Bwlp[p^*, f] = \bigwedge_{i=0}^n Bwlp[p^i, f]. \quad (2)$$

In (2), 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 $\{\alpha(\bar{y})\} P(\bar{y}) \{\beta(\bar{y}, \bar{y}')\}$

$$\begin{aligned} pcat(\{\alpha\} P \{\beta\}) &= \\ \forall \bar{y} \left(\alpha \implies \left(Bwlp \left[p, \beta \Big|_{\bar{y}}^{\bar{x}_0} \right] \Big|_{\bar{y}'}^{\bar{y}} \right) \right). \quad (3) \end{aligned}$$

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

DynAlloy was introduced in [2] as an alternative to the use of traces proposed in [7] for the analysis of dynamic properties. While the proposal of [2] had methodological and practical advantages over the proposal from [7] (see [3] for more details), our proposal still had a shortcoming, namely, the lack of an adequate mechanism for handling complex objects. Consider the following DynAlloy atomic action that modifies the “next” field in a list.

$$\begin{aligned} \text{act SetNext}(l1, l2 : \text{List}) \\ \text{pre} = \{ l1 \text{ != Empty} \} \\ \text{post} = \{ l1'.\text{val} = l1.\text{val} \text{ and } l1'.\text{next} = l2 \} \end{aligned}$$

The following partial correctness assertion should be valid if we adopt a Java-like semantics:

$$\begin{aligned} \{ l1 \text{ != Empty} \text{ and } l2 \text{ != Empty} \} \\ \text{SetNext}(l1, l2); \\ \text{SetNext}(l2, l1) \\ \{ l1'.\text{next}.\text{next} = l1' \} \quad (4) \end{aligned}$$

Unfortunately, property “ $l1.\text{next}.\text{next} = l1$ ” does not hold in the final state. Therefore, an appropriate modeling for objects in DynAlloy must be found. We adopt the object model of JAlloy [8]. JAlloy translates Java programs directly to Alloy, and has been applied to finding bugs in Java programs [8, 12], and to modular analysis of code in [11].

The JAlloy model of the List signature requires just a basic signature for lists without fields

$$\text{sig List } \{ \},$$

and fields are considered as binary relations

$$\begin{aligned} \text{val} : \text{List} \rightarrow \text{lone Val}, \\ \text{next} : \text{List} \rightarrow \text{lone List}. \end{aligned}$$

These binary relations can be modified by the DynAlloy actions. We will in general distinguish between simple data that will be handled as values, and structured objects.

Action SetNext is now specified as follows:

$$\begin{aligned} \text{act SetNext}(l1, l2 : \text{List}, \text{next} : \text{List} \rightarrow \text{lone List}) \\ \text{pre} = \{ l1 \text{ != Empty} \} \\ \text{post} = \{ \text{next}' = \text{next} ++ (l1 \rightarrow l2) \} \end{aligned}$$

If we analyze again the state evolution for action

$$\text{SetNext}(l1, l2, \text{next}); \text{SetNext}(l2, l1, \text{next})$$

under the new specification, we get:

State Evolution	Actions
$l1 = L1, l2 = L2, \text{next} = N1$	SetNext($l1, l2, \text{next}$)
$\text{next} = N1 ++ (L1 \rightarrow L2)$ and $l1 = L1, l2 = L2$	SetNext($l2, l1, \text{next}$)
$\text{next} = (N1 ++ (L1 \rightarrow L2)) ++ (L2 \rightarrow L1)$ and $l1 = L1, l2 = L2$	

Notice that the partial correctness assertion (4) is now valid. We can now reason as follows. Assume that $L1 \neq L2$ (we will drop this assumption later).

$$l1.\text{next}.\text{next} = L1.\text{next}.\text{next} = L2.\text{next} = L1 = l1.$$

Thus, formula (4) indeed holds. If $L1 = L2$, notice that $\text{next} = N1 ++ (L2 \rightarrow L1)$ (due to the definition of $++$).

$$\begin{aligned} l1.\text{next}.\text{next} = L1.\text{next}.\text{next} = L2.\text{next}.\text{next} = L1.\text{next} = \\ L2.\text{next} = L1 = l1. \end{aligned}$$

3 Optimizations to DynAlloy for the Analysis of OO Code

While the optimizations presented in [3] can be applied to arbitrary DynAlloy specifications, specific optimizations can be added to DynAlloy actions modeling object-oriented programs. These optimizations allow us to produce Alloy models from DynAlloy models, that can be analyzed more efficiently. In this section we present two optimizations.

In our extensive experience as demanding Alloy users, we have experienced some of the limitations Alloy has. Most of these limitations are inherited from the use of SAT-solvers. The limitations can be summarized as follows:

1. Alloy models containing large formulas (as the ones obtained from the translation of DynAlloy programs coming from Java code) take a long time to compile. They can easily exhaust the resources of a modern personal computer.
2. The analysis time required by the SAT-solvers grows (usually) exponentially when the scopes are increased.

In order to cope with these limitations of Alloy, we propose optimizations for the translation of DynAlloy models to Alloy. These optimizations allow us to:

1. avoid large monolithic formulas; thus allowing us to compile much larger models within a few seconds, as opposite to the hours it took before.
2. use smaller scopes, giving rise to models that can be analyzed in a fraction of the time they previously demanded.

In the remaining part of this section we will present the optimizations. In Section 4 we will analyze the impact these optimizations have in analysis through some examples.

3.1 Handling Large Formulas

Notice that the translation of a DynAlloy program to Alloy is, so far, a single formula. In our experience, these formulas tend to grow easily if they involve closures and the number of loop unrollings grows. These long formulas pose a challenge to the Alloy compiler.

Notice also that after a number n of loop unrollings is set, a partial correctness assertion of the form

$$\begin{array}{c} \{pre(a)\} \\ A^* \\ \{post(a, a')\} \end{array}$$

becomes the DynAlloy partial correctness assertion

$$\begin{array}{c} \{pre(a)\} \\ A; \\ \vdots (n - \text{times}) \\ A \\ \{post(a, a')\} \end{array}$$

If action A has precondition $\alpha(a)$ and postcondition $\beta(a, a')$ (where a and a' are arrays of variables), the last action gets translated to Alloy as an assertion of the form

$$\begin{array}{l} \text{assert pca}\{\text{all } a_0, \dots, a_n : S \mid \\ pre(a_0) => (\alpha(a_0) => (\beta(a_0, a_1) => (\alpha(a_1) => \\ (\beta(a_1, a_2) => \dots => (\beta(a_{n-1}, a_n) \\ => post(a_0, a_n))))))\}. \end{array} \quad (5)$$

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

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

and replace the previous assertion by the following equivalent facts and assertion:

$$\begin{array}{l} \text{fact } \{pre(T.a_0)\} \\ \text{fact } \{\alpha(T.a_0)\} \\ \text{fact } \{\beta(T.a_0, T.a_1)\} \\ \text{fact } \{\alpha(T.a_1)\} \\ \text{fact } \{\beta(T.a_1, T.a_2)\} \\ \vdots \\ \text{fact } \{\beta(T.a_{n-1}, T.a_n)\} \\ \text{assert pca}\{post(T.a_0, T.a_n)\} \end{array}$$

Of course, it is seldom the case that the DynAlloy action under analysis has the exact format A^* . It should be clear nevertheless that the previous technique can be applied to actions consisting of a sequential composition of (not necessarily atomic) DynAlloy actions A_1, \dots, A_n . Again, most composite actions do not consist of a single sequential composition of actions. Programs usually have branches in order to handle different data (base cases versus composite data in recursive programs, for instance). Our strategy in this case is to automatically split the original DynAlloy assertion into several assertions. The splitting process takes as input a modified version of the control flow graph of the (unrolled) DynAlloy action. The modification consists on treating each loop iteration as an atomic action. For instance, if we are given the DynAlloy action

$$[\text{test}]? + [!\text{test}]?; (A + B)^*,$$

after unrolling the loop we obtain action

$$[\text{test}]? + [!\text{test}]?; (A + B); \dots; (A + B).$$

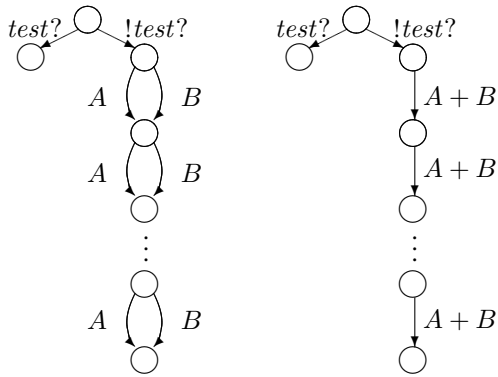


Figure 3. A control flow graph and its modified version.

The control flow graph and its modified version for this action are given in Fig. 3.

We then look for a path from the root that covers the loop unrollings (in the example this path corresponds to the action $[!test?]; (A + B); \dots; (A + B)$) and build an assertion from it. After removing this path from the graph, the procedure continues until no more paths exist containing loop unrollings. The action corresponding to the graph that remains in the end is translated to a new assertion (in the example, the remaining action is $[test?]$). Notice that the paths containing the loop unrollings have the appropriate shape to be translated to Alloy using “facts”.

Our experience is that using this technique much larger models can be compiled. In Section 4 we present several examples; for these examples we could not compile models with more than 4 loop unrollings. Using “facts” it is possible to easily compile the same models but with 20 loop unrollings. In most examples, splitting the assertions even improved the overall analysis time. Reducing nondeterminism helps the SAT-solvers perform more efficiently. Also, notice that the resulting assertions can be analyzed in parallel. Since the number of paths in the modified control flow graphs is usually small, parallel analysis demands few resources.

3.2 Deriving Appropriate Scopes

Another source of complexity in the analysis process is the scope of domains. Usually, for some domains it is necessary to have at least as many objects as loop unrollings are performed in the DynAlloy code. If we aim at performing several loop unrollings as a mean to gain confidence about the correctness of the source code, scopes end up making the analysis unfeasible. Therefore, reducing the

search space is necessary in order to gain analyzability. In this section we present a technique to this end.

Recall that object fields are (as in JAlloy [8]) modeled as binary relations. Actions that modify fields make these relations evolve. Thus, the Alloy assertion resulting from the translation of a DynAlloy program that makes a field f to evolve k times, will contain universally quantified relation variables f_0, f_1, \dots, f_k . Notice that if at most k evolutions of fields introduced in signature S occur, at most k objects from S will be modified. This means that if the scope for signature S is n (larger than k), there will be $n - k$ objects whose fields will not be modified. We call these objects *non mutant*, and those that are modified, *mutant*. We replace signature S by two new signatures:

$$\text{sig mutS}\{ \}, \text{ and } \text{sig nonmutS}\{ \} .$$

Also, the DynAlloy model is automatically processed in order to re-type relational variables. For instance, an action that modified a field $f : S \rightarrow T$, now receives a parameter $df : \text{mutS} \rightarrow T$. Notice that this simple process gives rise (upon translation to Alloy) to an Alloy assertion that rather than quantifying over relations f_0, f_1, \dots, f_k , now quantifies over relations

$$df_0, \dots, df_k : \text{mutS} \rightarrow T, \quad sf_0 : \text{nonmutS} \rightarrow T .$$

The gain in using this technique comes from realizing that while the original signature S had scope n , the new signatures will have scopes:

$$\text{mutS} : k, \quad \text{and} \quad \text{nonmutS} : n - k .$$

If we analyze the size of the potential search space in each case, if signature T has scope m , we obtain:

1. $f_0, \dots, f_k \mapsto 2^{(k+1) \times n \times m}$.
2. $df_0, \dots, df_k, sf_0 \mapsto 2^{(k+1) \times k \times m} \times 2^{(n-k) \times m}$.

We compare in Table 3.2 both values for $n = m = 10$ and varying values of k .

k	0	1	2	3	4
S	2^{100}	2^{200}	2^{300}	2^{400}	2^{500}
mutS and nonmutS	2^{100}	2^{110}	2^{140}	2^{190}	2^{260}

k	5	6	7	8	9	10
S	2^{600}	2^{700}	2^{800}	2^{900}	2^{1000}	2^{1100}
mutS and nonmutS	2^{350}	2^{460}	2^{590}	2^{740}	2^{910}	2^{1100}

Table 1. Comparison of state space sizes.

Both when $k = 1$ (no evolution of field f) or when $k = n$ (all elements can be modified), no improvement is reported (and even an overhead might be generated). In all other cases there is a clear advantage in using this technique.

The process of re-typing a DynAlloy program according to the previous technique is straightforward. Those actions that modify a field will now be declared as:

```
act Setf(o : S, v : T, f : S → T)
  pre = { true }
  post = { f' = f++(o → v) }
  ↦
  act Setf(o : mutS, v : T, f : mutS → T)
    pre = { true }
    post = { f' = f++(o → v) }
```

Notice that we can deduce now that variable o can be typed as $\text{mut}S$. The problem still remains about how to type variables occurring in pre and post-conditions. This is not complex; if we are given a variable of signature S , it gets type $\text{nonmut}S + \text{mut}S$. Similarly, if we are given a field variable $f : S \rightarrow T$, it now gets typed as $f : \text{nonmut}S + \text{mut}S \rightarrow T$. Therefore, if we are analyzing a partial correctness assertion

$$\begin{aligned} & \{ \text{pre}(f) \} \\ & \text{act } A(f : S \rightarrow T) \\ & \{ \text{post}(f) \} \end{aligned}$$

and action A makes field f to evolve, we analyze instead the partial correctness assertion

$$\begin{aligned} & \{ \text{pre}(df + sf) \} \\ & \text{act } A(sf : \text{nonmut}S \rightarrow T, df : \text{mut}S \rightarrow T) \\ & \{ \text{post}(df' + sf) \} \end{aligned}$$

Notice that both in the precondition and the postcondition we use relation sf . This is because it only contains objects that do not evolve.

In general, if we are given an Alloy “check” of the form

check *assertion* for m but n S ,

and there are k objects that evolve for signature S , the check is replaced by the following:

check *assertion* for m but k $\text{mut}S$, $n - k$ $\text{nonmut}S$.

Determining the amount of evolving objects is not always obvious. In some cases it is possible to determine the exact amount automatically. This is the case, for instance, for the methods analyzed in Section 4.1. In some other cases, it is not always clear whether different variables refer to different objects, and therefore counting the number of evolutions of different variables may produce an upper bound. In case there is enough knowledge about the given problem, the computed upper bound can be manually refined.

4 Examples

In Section 4.1 we present an implementation of sets based on single linked lists, and analyze the correctness of the implementations. We present running time displayed in the format “mm:ss”. Experiments were performed in a personal computer with an AMD 3200, 64 bits processor; 2GB, dual channel memory, and Linux Mandriva 10.2, 64 bits. We employed Alloy Analyzer (version 3.0 Beta, February 9th., 2006), and used the BerkMin SAT-solver.

Usually, when presenting case-studies of analysis of code using Alloy, examples do not consider more than 3 loop unrollings, and small scopes for data domains (seldom more than 4) are considered. In this paper, and using the optimizations presented in Section 3, we go one step further by considering larger numbers of loop unrollings and data domain scopes.

4.1 Analysis of Sets Implemented as Single Linked Lists

In Fig. 4 we present a fragment of a DynAlloy specification showing: In lines 4–7, the signature hierarchy. In lines 8–18, two auxiliary atomic actions for assigning values to variables. In lines 19–25, an auxiliary predicate to be used in the setter for field “isEmpty”. In lines 26–33, the setter for field “isEmpty”. In lines 34–53, the invariants for this class. In lines 54–67, a predicate characterizing the abstraction function. In lines 68–85, an auxiliary program that models the body of the loop inside method “isMember”. In lines 86–105, the DynAlloy translation of method “isMember”. In lines 106–126, two auxiliary predicates to be used in the precondition and postcondition of the partial correctness assertion for method “isMember”. In lines 127–142, the partial correctness assertion to be analyzed.

In Section 3.1 we mentioned that the Alloy compiler’s performance was undermined when compiling large formulas. In Table 2 we compare compilation times for method “insert” with and without “facts”. A similar progression for compilation times has been observed for the remaining methods.

unrollings	0-1	0-2	0-3	0-4	0-5	0-6
no facts	00:54	13:22	3:50:32	>10h	>10h	>10h
facts	00:04	00:04	00:04	00:05	00:06	00:06

Table 2. Compilation times with and without “facts” for action insert.

In Fig. 5 we present the DynAlloy model for method “insert”. We removed what is common with Fig. 4.

According to the optimization technique presented in

Section 3.1, we split the DynAlloy action “insert” into three different DynAlloy programs, namely,

1. “insertToEmpty”, the action branch that inserts the integer in a (previously) empty list,
2. “noInsert”, the action branch that finds the object to be inserted already in the list (and therefore does not insert the integer),
3. “insertToNonEmpty”, the action branch that inserts the integer at the end of the given list.

In Fig. ?? we present a fragment of the DynAlloy model for action “insertToNonEmpty” distinguishing between mutant and non mutant objects.

In the case of “insertToEmpty” and “insertToNonEmpty” there are two object evolutions. There are no evolutions for action “noInsert”, and therefore we do not distinguish between non mutant and mutant elements. Therefore, a check of the form

check *assertion* for m but n IntListSet,

is transformed to a check

check *assertion* for m but $n - 2$ nonmutIntListSet, 2
mutIntListSet

in case we distinguish between non mutant and mutant objects.

In Table 3 we report analysis times for action “insertToEmpty”, not distinguishing between non mutant and mutant objects. In Table 4 we report analysis times for action “insertToEmpty”, but this time distinguishing between non mutant and mutant objects. We use boldface in order to mark the better entries between the tables.

scope	3	4	5	6	7	8
time	00:01	00:01	00:01	00:01	00:01	00:02

Table 3. Analysis times for action “insertToEmpty” (no distinction of non mutant and mutant objects).

scope	3	4	5	6	7	8
time	00:08	00:08	00:08	00:11	00:14	00:17

Table 4. Analysis times for action “insertToEmpty” (distinguishing non mutant and mutant objects).

In Table 5 we report analysis times for action “insertToNonEmpty”, not distinguishing between non mutant and

mutant objects. In Table 6 we report analysis times for action “insertToNonEmpty”, but this time distinguishing between non mutant and mutant objects. We use boldface in order to mark the better entries between the tables. Notice that the code for action “insertToNonEmpty” does not include any loops. Therefore we only vary the scope.

unrollings [→] scope ↓	0-1	0-2	0-3	0-4	0-5	0-6	0-7
3	00:02	00:02	00:02	00:02	00:02	00:02	00:03
4	00:01	00:02	00:02	00:02	00:03	00:03	00:03
5	00:02	00:03	00:06	00:12	00:18	00:23	00:28
6	00:02	00:08	00:27	01:43	03:02	04:50	06:50
7	00:04	00:14	01:22	11:41	36:38	>60'	>60'
8	00:06	00:28	05:07	>60'	>60'	>60'	>60'

Table 5. Analysis times for action “insertToNonEmpty” (no distinction of non mutant and mutant objects).

unrollings [→] scope ↓	0-1	0-2	0-3	0-4	0-5	0-6	0-7
3	00:09	00:09	00:09	00:09	00:09	00:10	00:10
4	00:09	00:09	00:10	00:09	00:09	00:10	00:10
5	00:09	00:10	00:09	00:10	00:11	00:12	00:12
6	00:11	00:11	00:12	00:12	00:14	00:15	00:18
7	00:12	00:13	00:15	00:15	00:17	00:24	00:27
8	00:16	00:17	00:18	00:20	00:23	00:27	00:41

Table 6. Analysis times for action “insertToNonEmpty” (distinguishing non mutant and mutant objects).

In Table 7 we present analysis times for action “noInsert”. Notice that since no field evolutions occur in this action, we do not separate between non mutant and mutant objects.

unrollings [→] scope ↓	0-1	0-2	0-3	0-4	0-5	0-6	0-7
3	00:01	00:01	00:01	00:01	00:02	00:02	00:02
4	00:01	00:01	00:01	00:01	00:01	00:02	00:02
5	00:01	00:02	00:03	00:03	00:05	00:05	00:06
6	00:02	00:04	00:07	00:16	00:25	01:10	01:27
7	00:04	00:09	00:20	00:50	01:15	03:43	10:38
8	00:10	00:23	00:58	02:31	05:33	08:44	19:46

Table 7. Analysis times for action “noInsert”.

Finally, in Table 8 we compare the accumulated analysis times for both approaches, denoted as “I” (for “insert”) and “IM” (referring to action “insert” under the separation between non mutant and mutant objects).

unrollings ↑ scope ↓	0-5		0-6		0-7	
	I	IM	I	IM	I	IM
3	00:05	00:19	00:05	00:20	00:06	00:20
4	00:05	00:18	00:06	00:20	00:06	00:20
5	00:24	00:24	00:29	00:25	00:35	00:26
6	03:28	00:50	06:01	01:36	08:18	01:56
7	37:54	01:46	>60'	04:21	>60'	11:19
8	>60'	06:13	>60'	09:28	>60'	20:44

Table 8. Accumulated analysis times for action “insert”.

5 Conclusions

In this article we have presented optimizations to the analysis of DynAlloy specifications. These optimizations produce dramatic improvements in the analysis time for those programs whose analysis time is SAT-bound. This is particularly evident when comparing Tables 5 and 6. In this case, the time for up to 7 loop unrollings and 8 lists reduces from many hours (according to the time progression we see in the table), to just 41 seconds. Observing the accumulated analysis times for the list’s method “insert” (c.f. Table 8) the time reduction appears to be exponential, which is coherent with Table 1. Notice also that the overall analysis time when distinguishing non mutant and mutant elements gets most of its weight from actions for which it is not possible to separate into non mutant and mutant objects. In the case of method “insert” with up to 7 loop unrollings and 8 lists, 19:46 out of the total 20:44 come from action “noInsert”, for which our technique cannot profit from the distinction between non mutant and mutant objects.

It is then clear that new optimizations as the ones presented in this paper are necessary for the situations in which the current optimizations do not apply, and this is indeed further work we plan to carry on in the near future.

There are cases in which distinguishing non mutant and mutant objects yields worse analysis times. Examples can be seen in Tables 3 and 4. In all these cases it is clear that the analysis time is mainly spent in pre-processing the model and not in SAT-solving. The difference against our technique comes from the overhead introduced to the pre-processing due to the splitting of types. In all cases the difference is small and is measured in seconds.

References

- [1] Dijkstra, E., and Scholten, C. 1990. *Predicate calculus and program semantics*, Springer-Verlag.
- [2] Frias M., Lopez Pombo C., Baum G., Aguirre N., and Maibaum T. 2005. *Reasoning About Static and Dynamic Properties in Alloy: A Purely Relational Approach*, in ACM-Transactions on Software Engineering and Methodology (TOSEM), 14(4), 478–526.
- [3] Frias, M., Galeotti, J., Lopez Pombo, C., and Aguirre, N. 2005. *DynAlloy: Upgrading Alloy with Actions*, in Proceedings of the 27th. ACM-IEEE International Conference on Software Engineering (ICSE) 2005, St. Louis, USA, May 2005, ACM Press, 442–450.
- [4] Frias, M., and Galeotti, J. 2006. *Efficient Analysis of JML–Annotated Java Programs Using DynAlloy*, submitted.
- [5] Jackson, D. 2002a. *Alloy: a lightweight object modelling notation*. ACM Transactions on Software Engineering and Methodology 11, 2, 256–290.
- [6] Jackson, D. 2002b, *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*, MIT Laboratory for Computer Science, Cambridge, MA.
- [7] 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, 62–73, Vienna, Austria, Association for the Computer Machinery, ACM Press.
- [8] Jackson, D. and Vaziri, M. 2000, *Finding Bugs with a Constraint Solver*, in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), August 21-24, 2000, Portland, OR, USA. ACM, pp. 14–25.
- [9] Jackson, D. 2002b, *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*, MIT Laboratory for Computer Science, Cambridge, MA.
- [10] Leavens, G., Baker, A., sc and Ruby, C. 1999. *JML: a Notation for Detailed Design*. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175–188.
- [11] Taghdiri, M. 2004. *Inferring Specifications to Detect Errors in Code*. 19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria. IEEE Computer Society. pp. 144–153.
- [12] Vaziri, M. and Jackson, D. 2003. *Checking Properties of Heap-Manipulating Procedures with a Constraint Solver*, in Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Warsaw, Poland, Lecture Notes in Computer Science 2619 Springer, pp. 505–520.

```

1. module IntListSet
2. open util/boolean
3. open util/integer
4. one sig null {}
5. sig Object {}
6. abstract sig IntSet extends Object {}
7. sig IntListSet extends IntSet {}
8. action changeBoolValue(r: Bool, l: Bool) {
9   pre {}
10  post {r'=l}
11 }
12 action changeListSetValue(
13   r: IntListSet+null,
14   l: IntListSet+null)
15 {
16   pre {}
17   post {r'=l}
18 }
19 pred setIsEmptyPost(
20   l: IntListSet,
21   b: Bool, isEmpty0,
22   isEmpty1: IntListSet -> one Bool)
23 {
24   isEmpty1=isEmpty0++l->b
25 }
26 action setIsEmpty(
27   l: IntListSet,
28   b: Bool,
29   isEmpty: IntListSet -> one Bool)
30 {
31   pre {}
32   post { setIsEmptyPost(l,b,isEmpty,isEmpty') }
33 }
34 pred invarIntListSet(
35   l: IntListSet+null,
36   isEmpty: IntListSet -> one Bool,
37   value: IntListSet -> one Int,
38   next: IntListSet -> one (IntListSet+null)
39 ) {
40   (l!=null) &&
41   (l.next=null ||
42    (all l1: IntListSet |
43     l1 in l.(next) implies l1.isEmpty=False)
44   ) &&
45   (all l1: IntListSet |
46    l1 in l.(next) implies
47     (l1 !in l1.next.(next))) &&
48   (all l1, l2: IntListSet |
49    (l1 in l.next && l2 in l.next &&
50     l1.isEmpty = False && l2.isEmpty = False &&
51     l1.value = l2.value) implies
52     l1 = l2)
53 }
54 pred fAbs(
55   estr: IntListSet,
56   absData: set Int,
57   isEmpty: IntListSet -> one Bool,
58   value: IntListSet -> one Int,
59   next: IntListSet -> one (IntListSet+null)
60 ){
61   all i: Int |
62   i in absData iff
63     (some l: IntListSet |
64      l in estr.(next) &&
65      l.isEmpty=False &&
66      l.value=i)
67 }
68 program whileIsMember(
69   current: IntListSet+null,
70   i: Int,
71   result: Bool,
72   value: IntListSet -> one Int,
73   next: IntListSet -> one (IntListSet+null))
74 {
75   [(current!=null && result=False)]?;
76   (
77     [current.value=i]?;changeBoolValue(result, True)
78     +
79     [current.value!=i]?;skip
80   );
81   changeListSetValue(current, current.next)
82 +
83   [(current=null || result=True)]?;
84   skip
85 }
86 program isMember(
87   thisObject: IntListSet,
88   i: Int,
89   result: Bool,
90   current: IntListSet+null,
91   isEmpty: IntListSet -> one Bool,
92   value: IntListSet -> one Int,
93   next: IntListSet -> one (IntListSet+null))
94 {
95   [result=False]?;
96   (
97     [thisObject.isEmpty=False]?;
98     [current=thisObject]?;
99     (whileIsMember(current, i, result, value, next))*;
100    [(current=null || result=True)]?
101  +
102    [thisObject.isEmpty=True]?;
103    skip
104  )
105 }
106 pred isMemberObeysSpecPre(
107   l: IntListSet,
108   absPre: set Int,
109   isEmpty: IntListSet -> one Bool,
110   value: IntListSet -> one Int,
111   next: IntListSet -> one (IntListSet+null)
112 ) {
113   invarIntListSet(l,isEmpty,value,next) &&
114   fAbs(l,absPre,isEmpty,value,next)
115 }
116 pred isMemberObeysSpecPost(
117   l: IntListSet,
118   i: Int,
119   result: Bool,
120   absPre: set Int,
121   isEmpty: IntListSet -> one Bool,
122   value: IntListSet -> one Int,
123   next: IntListSet -> one (IntListSet+null)
124 ) {
125   invarIntListSet(l,isEmpty,value,next)
126   && (result=True iff (i in absPre))
127 }
128 assert isMemberObeysSpec {
129   assertCorrectness(
130     l: IntListSet,
131     i: Int,
132     result: Bool,
133     current: IntListSet+null,
134     absPre: set Int,
135     isEmpty: IntListSet -> one Bool,
136     value: IntListSet -> one Int,
137     next: IntListSet -> one (IntListSet+null)
138   ){
139     pre = { isMemberObeysSpecPre(l,absPre, isEmpty,value,next) }
140     program = { isMember(l,i,result,current,isEmpty,value,next) }
141     post = { isMemberObeysSpecPost(l,i,result',absPre,
142                                     isEmpty',value',next') }
143   }
144 }

```

Figure 4. Fragment of DynAlloy model. Implementation of sets by single linked lists.

```

/*-----*/
/* Actions on fields and object constructor */
/*-----*/
pred setNextPost(l: IntListSet, n: IntListSet+null,
  next0, next1: IntListSet -> one (IntListSet+null)) {
  next1=next0+1->n }

action setNext(l: IntListSet, n: IntListSet+null,
  next: IntListSet -> one (IntListSet+null)) {
  pre {}
  post { setNextPost(l,n,next,next') } }

program newIntListSet(result: IntListSet+null,
  alive: set IntListSet, isEmpty: IntListSet -> one Bool,
  value: IntListSet -> one Int,
  next: IntListSet -> one (IntListSet+null)) {
  [result !in alive]?;
  setIsEmpty(result,True,isEmpty);
  setNext(result,null,next);
  setValue(result,Int(0),value) }

/*-----*/
/* action insert */
/*-----*/
program whileInsert(i: Int, found: Bool,
  prev: IntListSet+null, current: IntListSet+null,
  isEmpty: IntListSet -> one Bool,
  value: IntListSet -> one Int,
  next: IntListSet -> one (IntListSet+null)) {
  [(current!=null && found=False)]?;
  ( [current.value=1]?;changeBoolValue(found,True)
  +
  [current.value=1]?;skip
  );
  changeListSetValue(prev, current);
  changeListSetValue(current, current.next)
  +
  [(current=null || found=True)]?;skip }

program insert(thisObject: IntListSet,
  i: Int, found: Bool, prev: IntListSet+null,
  current: IntListSet+null, newNode: IntListSet+null,
  isEmpty: IntListSet -> one Bool,
  value: IntListSet -> one Int,
  next: IntListSet -> one (IntListSet+null)) {
  [found=False]?;
  ( [thisObject.isEmpty=True]?;
  setValue(thisObject,i,value);
  setIsEmpty(thisObject,False,isEmpty)
  +
  [thisObject.isEmpty=False]?;
  [prev=thisObject]?;
  [prev=null]?;
  (whileInsert(i, found, prev, current,
    isEmpty, value,next))*;
  [(current=null || found=True)]?;
  ( [found=False]?;
  newIntListSet(newNode,thisObject+univ.next,
    isEmpty,value,next);
  setIsEmpty(newNode,False,isEmpty);
  setValue(newNode,i,value);
  setNext(prev,newNode,next)
  +
  [found=True]?;
  skip
  )
  )
}

/*-----*/
/* Assertions */
/*-----*/
pred insertObeysSpecPre(
  l: IntListSet,
  absPre: set Int,
  isEmpty: IntListSet -> one Bool,
  value: IntListSet -> one Int,
  next: IntListSet -> one (IntListSet+null)) {
  invarIntListSet(l,isEmpty,value,next) &&
  fAbs(l,absPre,isEmpty,value,next) }

pred insertObeysSpecPost(
  l: IntListSet,
  i: Int,
  absPre: set Int,
  absPost: set Int,
  isEmpty: IntListSet -> one Bool,
  value: IntListSet -> one Int,
  next: IntListSet -> one (IntListSet+null)) {
  fAbs(l,absPost,isEmpty,value,next) implies
  (invarIntListSet(l,isEmpty,value,next) and
  (absPost=absPre+i) ) }

assert insertObeysSpec {
  assertCorrectness(
  l: IntListSet,
  i: Int,
  found: Bool,
  prev: IntListSet+null,
  current: IntListSet+null,
  newNode: IntListSet+null,
  absPre: set Int,
  absPost: set Int,
  isEmpty: IntListSet -> one Bool,
  value: IntListSet -> one Int,
  next: IntListSet -> one (IntListSet+null)) {
  pre = { insertObeysSpecPre(l,absPre,isEmpty,value,next) }
  program = { insert(l,i,found,prev,current,newNode,
    isEmpty,value,next) }
  post = { insertObeysSpecPost(l,i,absPre,absPost,
    isEmpty',value',next') } }
}

```

Figure 5. DynAlloy model for method “insert”.