A Dataflow Analysis to Improve SAT-based Bounded Program Verification

Bruno Cuervo Parrino[1], Juan Pablo Galeotti[1,2], Diego Garbervetsky[1,2], and Marcelo F. Frias[2,3]
{bcuervo,jgaleotti,diegog}@dc.uba.ar, mfrias@itba.edu.ar

[1] Departmento de Computación, FCEyN, UBA
[2] CONICET
[3] Department of Software Engineering, ITBA

Abstract. SAT-based bounded verification of programs consists of the translation of the code and its annotations into a propositional formula. The formula is then analyzed for specification violations using a SAT-solver. This technique is capable of proving the absence of errors up to a given scope. SAT is a well-known NP-complete problem, whose complexity depends on the number of propositional variables occurring in the formula. Thus, reducing the number of variables in the logical representation may have a great impact on the overall analysis. We propose a dataflow analysis which infers the set of possible values that can be assigned to each local and instance variable. Unnecessary variables at the SAT level can then be safely removed by relying on the inferred values. We implemented this approach in TACO, a SAT-based verification tool. We present an extensive empirical evaluation and discuss the benefits of the proposed approach.

1 Introduction

Bounded verification [7] is a technique in which all executions of a procedure are exhaustively examined within a finite space given by a bound (a) on the domain sizes and (b) on the number of loop unrollings. The scope of analysis is examined in order to look for an execution trace that violates the provided specification.

Several bounded verifications tools [7, 10, 12, 24] rely on appropriately translating the original piece of software, as well as the specification to be verified, to a propositional formula. The use of a SAT-Solver [3] then allows us to find a valuation for the propositional variables that encodes a failure. Theoretically, SAT-solving time grows exponentially w.r.t. the number of propositional variables. However, modern SAT solvers achieve better results on practical instance problems.

In contrast to other analyses relying on theorem provers such as SMT-solvers [16], SAT-based analyses cannot be used to prove programs are correct. They only guarantee the absence of errors within the given scope. Nevertheless, SAT-based tools are better suited for finding counterexamples. By bounding the scope of analysis, these tools are able to faithfully represent the program behavior without loosing precision (i.e., no false warnings).

The size of the propositional formula and its number of variables are highly related to the size and shape of the annotated program, the state representation (for Java: local, global variables and the heap) and the given scope of analysis. Therefore, techniques aiming at reducing any of these factors could possibly have a great impact on the overall verification cost.

Dataflow analysis [15] is a static analysis technique which is widely used for program understanding and optimization. Roughly speaking, it infers facts about the program by collecting the data flowing through its control flow graph (usually an abstraction of the concrete program state). Instances of dataflow analyses are: live variable analysis, available expressions, reachable definitions, constant propagation, etc. These analyses enable compilers to eliminate dead code, reduce unnecessary run-time checks, remove redundant computations, etc.

In this work, we present a novel dataflow analysis for inferring the set of possible values that can be assigned to local and instance variables, at each program point. The obtained information is a safe over-approximation of the actual value each variable may have. We apply this value-propagation analysis in the context of bounded verification where it is possible to use a fine-grained abstraction without compromising termination.

We introduced this analysis in TACO [10], a SAT-based tool specially aimed at verifying JML-annotated [6] Java sequential programs. TACO accurately represents all Java data types (including primitive types such as double and float) and supports nearly all JML syntax. TACO does not report false bugs but is not able to prove the absence of errors above the given scope of analysis. Among its features, TACO introduces a novel technique for removing unnecessary propositional variables at the SAT level. This is accomplished by preprocessing class invariants in order to obtain a good over-approximation of the initial state of the Java memory heap. This set of initial values can be supplied to our dataflow analysis, obtaining a more accurate set of possible values for every program variable. In turn, this information leads to a more aggressive removal of unnecessary propositional variables at the SAT level.

TACO's previous representation was implemented as a simple sequence of `if` statements. This representation introduced at least one join point per loop unrolling which impacted negatively in the precision of the overall dataflow analysis. In this work we introduce an alternative representation for loop unrollings tailored to favor precision of the dataflow analysis.

Our experiments show significant speed-ups in analysis times: about 30 times reduction in average. Surprisingly, the proposed loop encoding has a significant impact in the overall verification time.

Contributions: The technical contributions of this article include:

– A formalization of a dataflow analysis for propagating values through a program control flow graph, including a proof sketch showing that the outcome of the analysis is a sound over-approximation of the program behavior.

```
class Node { Node next; }
class List { Node header;
  /*@ invariant (\forall Node n ;\reach(this.header,Node,next).has(n);
    @   !\reach(n.next,Node,next).has(n));
    @*/
}
```

Fig. 1: A singly linked list declaration

– A proof of the fact that the propositional formula obtained by TACO relying in this analysis is equisatisfiable w.r.t. the unoptimized formula.
– TACO-Flow: an extension of TACO featuring a general dataflow framework including proper generation of control flow graphs, the (bounded) value-propagation analysis and the generation of the optimized SAT-formula.
– An empirical evaluation using benchmarks accepted by the bounded verification community [10] showing an important speed-up in verification time.

Related work: There is plenty of work aiming at improving SAT-based program verification. The most remarkable examples are the approaches implemented in F-Soft [12], Saturn [24], TACO [10] and JForge [7]. Here we will focus only on related work concerning the use of dataflow analysis to alleviate the task of the SAT-solver. For a comprehensive discussion of these tools please refer to [9].

The idea of using dataflow analysis in the context of SAT-based program verification is not new. F-Soft [12] performs a dataflow analysis to compute ranges for values of integer-valued variables and pointers, under the hypothesis that runs have bounded lengths. Saturn [24] compresses formulas using several optimizations (e.g., program slicing) and provides means for specifying analyses aimed at producing method summaries. In [22], the authors proposed an analysis to infer method summaries and enable modular verification. JForge [7] uses a dataflow analysis to find and eliminate logically infeasible branches. In [18] the authors propose a technique based on dataflow analysis (variable-definitions) to split the SAT-problem into several simpler sub-problems.

Outline: §2 introduces the foundations of SAT-based verification and TACO, then it presents the problem we intend to tackle in the current work. §3 presents the value-propagation analysis. §4 shows how this technique is applied in the context of TACO. §5 shows our experimental results and, finally, §6 concludes and discusses future work.

2 Tight Bounds for Improved SAT-solving

Fig. 1 shows a JML declaration of a singly linked list data structure. It contains a `header` field referring to its first node. Each node links to its next node in the list by the `next` field. The List container is annotated with a JML object invariant which constraints the set of valid linked structures to those who form a finite acyclic sequence of Node elements. The construct `\reach(l,T,f)` denotes the set of objects of type T reachable from a location l using field f.

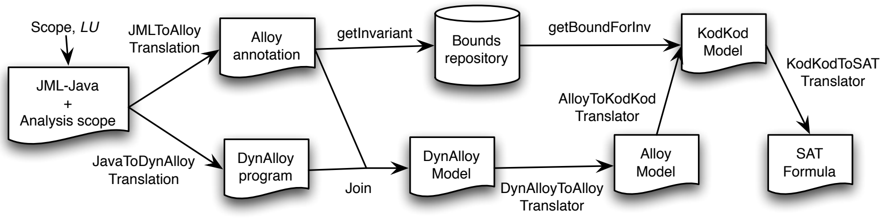


Fig. 2: Translating annotated code to SAT.

Method `removeLast` (shown in Fig. 3a) removes the last element of the list (if such an element exists). JML allows one to write a partial specification. In this example, the `ensures` clause only specifies that the returning Node element should not be reachable from the receiver list.

```
/*@ ensures
  @  !\reach(this.header,
  @   Node,next).has(\result);
  @*/
Node removeLast() {
  if (this.header!=null) {
    Node prev = null;
    Node curr = this.header;
    while (curr.next!=null) {
      prev = curr;
      curr = curr.next;
    }
    if (prev==null)
      this.header = null;
    else
      prev.next = null;

    return curr;
  } else
    return null;
}
```

(a) A `removeLast()` method

```
(this.header!=null)?;{
  prev := null;
  curr := this.header;
  {
    (curr.next!=null)?;
    prev = curr;
    curr = curr.next
  }*;
  ((curr.next==null)?;
  (prev==null)?;
    this.header := null
  +
  (prev!=null)?;
    prev.next := null;
  );
  return := curr;
}+
(this.header==null)?;
  return := null
```

(b) The DynAlloy representation

Fig. 3: Java implementation and its DynAlloy representation

As shown in Fig. 2, TACO translates the Java code annotated with the JML [6] contract into a DynAlloy specification [8]. DynAlloy is a relational specification language. In other words, every variable in DynAlloy can be seen as a relation of a fixed arity. For the `removeLast` method, the resulting DynAlloy program is shown in Fig. 3b. Signatures *List* and *Node* are introduced to model the Java classes. Also, a singleton signature *null* is defined to model the Java *null* value. **Java variables and fields** are represented using **DynAlloy variables**. Java fields are modelled in DynAlloy as functional binary relations (i.e., `S->one T`), and Java variables are modelled as unary non-empty relations (i.e., `one S`). The following DynAlloy variables are also introduced to model removeLast's Java variables and fields:

```
header: List -> one (Node+null)   prev:   one Node+null
next:   Node -> one (Node+null)   curr:   one Node+null
this:   one List                  return: one Node+null
```

If the user wants to perform a bounded verification of `removeLast`'s contract, she must limit the object domains and the number of loop iterations. Let us assume that a scope of at most 5 Node objects, 1 List object and 3 loop unrolls is chosen. The DynAlloy specification is then translated to an Alloy specification as described in [8]. In order to model state change in Alloy, the *DynAlloyToAlloy-Translator* may introduce several **Alloy relations** to represent different values (or *incarnations*) of the same DynAlloy variable in an SSA-like form [5].

In order to translate the Alloy specification into a SAT-problem, the Alloy Analyzer focuses on translating every Alloy relation into a set of **propositional variables**. Each propositional variable is intended to model that a given tuple is contained in the Alloy relation. In the example, as the $Node$ domain is restricted to 5 elements, $\{N_1, \ldots N_5\}$ is the set of 5 available Node atoms. This leads to the the following propositional variables modeling the binary relation $next_0$ (which, in turn, models the initial state of the DynAlloy variable `next`):

M_{next_0}	N_1	N_2	N_3	N_4	N_5	`null`
N_1	p_{N_1,N_1}	p_{N_1,N_2}	p_{N_1,N_3}	p_{N_1,N_4}	p_{N_1,N_5}	$p_{N_1,null}$
N_2	p_{N_2,N_1}	p_{N_2,N_2}	p_{N_2,N_3}	p_{N_2,N_4}	p_{N_2,N_5}	$p_{N_2,null}$
N_3	p_{N_3,N_1}	p_{N_3,N_2}	p_{N_3,N_3}	p_{N_3,N_4}	p_{N_3,N_5}	$p_{N_3,null}$
N_4	p_{N_4,N_1}	p_{N_4,N_2}	p_{N_4,N_3}	p_{N_4,N_4}	p_{N_4,N_5}	$p_{N_4,null}$
N_5	p_{N_5,N_1}	p_{N_5,N_2}	p_{N_5,N_3}	p_{N_5,N_4}	p_{N_5,N_5}	$p_{N_5,null}$

Following this representation, propositional variable p_{N_3,N_2} is true if and only if tuple $\langle N_3, N_2 \rangle$ is contained in the Alloy relation $next_0$. Given the selected scope of analysis, if no pre-processing is involved, the resulting SAT-problem will contain 126 propositional variables. Only 36 (28%) variables model the initial Java state, that is the representation of the receiver object instances. The remaining 90 (72%) variables are introduced to represent the intermediate and final stages for computing the `removeLast` method. That is, to model the state evolution during the execution of the method body.

Alloy uses KodKod [21] as an intermediate language, which is then translated to a CNF propositional formula (Fig. 2 sketches the translations involved). KodKod allows the prescription of bounds for Alloy relations. For each relation f, two relational instances L_f (the lower bound) and U_f (the upper bound) are attached. In any Alloy model I, f (the interpretation of relation f in model I), must satisfy $L_f \subseteq \mathsf{f} \subseteq U_f$. Therefore, pairs that are in L_f must necessarily belong to f, and pairs that are not in U_f cannot belong to f. If tuples are removed from an upper bound, the resulting upper bound is said to be *tighter* than before.

Tighter upper bounds contribute by removing propositional variables. Given an Alloy relation f, propositional variables corresponding to tuples that do not belong to U_f can be directly replaced in the translation process with the truth value *false*. This allows us to reduce the number of propositional variables.

TACO preprocesses class invariants and automatically computes a tight upper bound for the initial state of Java class fields. As shown in Fig. 2, bounds are stored in a repository. Since bounds are often reused during the analysis of different methods in a class, the cost of computing the bounds is amortized. This preprocessing allowed TACO to remove (in the presented example) over 70% of the propositional variables representing the initial state.

2.1 Problem Statement

The technique introduced in [10] limits itself to bound those propositional variables which represent the *initial* state of the program under analysis. One may argue that, as the SAT-solver is not able to recognize the order in which the program control flows, there is no guarantee that the SAT-solving process will avoid partial valuations from intermediate states that could not lead to a valid computation trace.

Dataflow analysis allows us to collect facts about the program behaviour at various points in a program. For instance, we could conclude that (under the scope of analysis previously chosen) for the following statement: `Node curr = this.header;` the set of values that are assignable to `curr` is $\{null, N_1\}$.

In this work we propose a dataflow analysis to over-approximate the set of possible values every Java variable and field may store within the provided scope of analysis. Using this conservative analysis, we can propagate the upper bounds from the initial state to the intermediate states. We believe the resulting tighter upper bounds for the intermediate Alloy relations should contribute by allowing KodKod to remove propositional variables. For instance, for the presented example we are able to remove about 58% of the propositional variables modeling intermediate stages. As we will see in §5, this technique leads to a potential improvement in the performance of the SAT-based verification.

3 Propagating Values in DynAlloy Programs

DynAlloy is based on first-order dynamic logic [11]. The aim of this specification language is to provide a formal characterization of imperative sequential programs. Fig. 4 shows a relevant fragment of DynAlloy's grammar. This fragment corresponds to the DynAlloy programs output by the *JavaToDynAlloy* translator in TACO. As shown in Fig. 3b, typical structured programming constructs can be described using these basic logical constructs. Given a DynAlloy program, our dataflow analysis computes an over approximation of all the possible variable assignments for every program location. We chose DynAlloy as a platform for the dataflow analysis because: 1) it is closer to the SAT-problem than the Java representation, and 2) it is the last intermediate representation in the TACO pipeline where a notion of control flow and state change still remains. In other words, DynAlloy contains the last imperative representation of the code under analysis.

$$\begin{array}{rlll} program ::= & v := expr & \text{“copy”} \\ | & v.f := expr & \text{“store”} \\ | & skip & \text{“skip action”} \\ | & formula? & \text{“test”} \\ | & program + program & \text{“non-deterministic choice”} \\ | & program; program & \text{“sequential composition”} \\ | & program^* & \text{“iteration”} \\ | & \langle program \rangle(\overline{x}) & \text{“invoke program”} \\ & & \\ expr ::= & \mathbf{null} \; | v \mid v.f & \end{array}$$

Fig. 4: Relevant DynAlloy fragment

Concrete Semantics: We begin by defining a concrete semantics for the execution of this DynAlloy fragment which mimics the execution of Java programs. As the DynAlloy relational semantics is interpreted in terms of atoms, *Atom* represents the set of all atoms in this interpretation. We denote by $JVar \uplus JField$ the set of DynAlloy variables. A DynAlloy variable belonging to $JVar$ corresponds to the representation of a Java variable and its concrete value is a single atom. Similarly, a variable belonging to $JField$ models a Java field whose concrete value is a mapping (functional relation) from atoms to atoms. A *concrete* state $c \in E$ maps each DynAlloy variable to a concrete value.

$$E = JVar \uplus JField \rightarrow Atom \cup \mathcal{P}(Atom \times Atom)$$

We denote by $M[\phi]_c$ the truth value for formula ϕ at the concrete state c. Similarly, we denote by $X[expr]_c$ the value of expression *expr* in the concrete state c. The value of $X[expr]_c$ for the DynAlloy expressions that we will consider could be defined as follows:

$$\begin{array}{ll} X[null]_c & = \{\langle null \rangle\} \\ X[v]_c & = \{c(v)\} \\ X[v.f]_c & = c(v); c(f) \end{array}$$

where the composition of relations R (arity i) and S (arity j) is defined as follows:

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

$R \rightarrow S$ denotes the Cartesian product between relations R and S. $R + +S$ denotes the relational overriding defined as follows[4]:

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

DynAlloy's relational semantics is given in [8]. Here we present an alternative definition based on the collecting semantics [17] which is useful for proving the correctness of our proposed dataflow technique. A collecting semantics defines

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

how information flows through a program control flow graph (CFG). Given a DynAlloy program P it is possible to obtain its CFG. The CFG describes the structure of the program. In the collecting semantics, every time a new value traverses a node it is recorded. Therefore, each node keeps track of all values that passed through it.

Definition 1. *Given a DynAlloy program P and a formula ϕ representing input states, the collecting semantics of P starting with state ϕ is the least fix point (LFP) of the following equations:*

For each node n in the CFG of P:

$$in(n) = \begin{cases} \{c_0 | M[\phi]_{c_0}\} & n \textit{ is the entry of } CFG(P) \\ \bigcup\limits_{p \in pred(n)} out(p) & \textit{otherwise} \end{cases}$$
$$out(n) = \mathcal{F}_c(n, in(n))$$

where $in(n)$, $out(n)$ denote the input and output values of node n and $pred(n)$ the set of predecessors of n.

The transfer function $\mathcal{F}_c : DynAlloyProgram \times C \to C$ with $C \in \mathcal{P}(E)$ models how the concrete state changes as a DynAlloy statement is executed. The definition is the following:

$$\begin{aligned} \mathcal{F}_c(skip, cs) &= cs \\ \mathcal{F}_c(\phi?, cs) &= \{c | c \in cs \wedge M[\phi]_c\} \\ \mathcal{F}_c(v := expr, cs) &= \{c[v \mapsto X[expr]_c] | c \in cs\} \\ \mathcal{F}_c(v.f := expr, cs) &= \{c[f \mapsto c(f) + +(c(v) \to X[expr]_c)] | c \in cs\} \end{aligned}$$

Abstract Semantics: We represent an *abstract* state by mapping each DynAlloy variable to its corresponding abstract value. The abstract value of a DynAlloy variable modelling a Java field is a (probably non-functional) binary relation from atoms to atoms. A DynAlloy variable representing a Java variable maps to a set of atoms.

$$A = JVar \uplus JField \to \mathcal{P}(Atom) \cup \mathcal{P}(Atom \times Atom)$$

An abstract value represents the set of all concrete values a DynAlloy variable *may* have (i.e., an over-approximation) in a given program location. In order to operate with this abstraction we need A to be a lattice. Let $\langle A, \sqsubseteq \rangle$ such that for all $a, a' \in A$:

- $a \sqsubseteq a'$ iff $\forall x \in JVar \uplus JField(a(x) \subseteq a'(x))$,
- $a \sqcup a' = a''$ such that $\forall x \in JVar \uplus JField(a''(x) = a(x) \cup a'(x))$

The abstraction function $\alpha : E \to A$ formalizes the notion of approximation of a concrete state by an abstract state. Given a concrete state $c \in E$:

$$\alpha(c) = a \text{ s.t. } \forall v \in JVar(a(v) = \{c(v)\}) \wedge \forall f \in JField(a(f) = c(f))$$

Notice that the chosen abstract domain is indeed very similar to the concrete domain. The main difference is that an abstract value can represent several concrete values (i.e., the powerset of *Atom*). Several concrete values are merged into a single abstract value after a join point in the CFG.

The concretization function $\gamma : A \to C$ is defined as: $\gamma(a) = \{c \mid \alpha(c) \sqsubseteq a\}$. Let X_α be the abstract evaluation function which is identical to X except that $a(v)$ directly returns a set. The abstract transfer function $\mathcal{F} : \mathit{DynAlloyProgram} \times A \to A$ is defined by the following set of rules:

```
– $\mathcal{F}(\mathit{skip}, a) = a$
– $\mathcal{F}(\phi?, a) = a$
– $\mathcal{F}(v := \mathit{expr}, a) = a[v \mapsto X_\alpha[\mathit{expr}]_a]$
– $\mathcal{F}(v.f := \mathit{expr}, a) = \textbf{let } \mathit{from} = a(v),\ \mathit{to} = X_\alpha[\mathit{expr}]_a \textbf{ in}$
        $\textbf{if } |\mathit{from}| = 1$
          $\textbf{then } a[f \mapsto a(f) + +(\mathit{from} \to \mathit{to})]$ (strong update)
          $\textbf{else } a[f \mapsto a(f) \cup (\mathit{from} \to \mathit{to})]$ (weak update)
```

Notice that the semantics of the store operation distinguishes two cases: 1) the abstraction is precise enough to perform an update of a unique source, 2) an over-approximated step must be taken. Due to space limitations we do not include the dataflow equations for the analysis. It is essentially equal to the collecting semantics but using the $\sqcup$ operator to merge states.

Correctness: Here we show that the abstraction is a sound over-approximation of the collecting semantics.

Theorem 1. *Let $cs \in C$, $a \in A$, $n \in CFG(P)$,*

$$\alpha(cs) \sqsubseteq a \Rightarrow \alpha(\mathcal{F}_c(n, cs)) \sqsubseteq \mathcal{F}(n, a)$$

Proof sketch: It can be proved for each statement separately. The proof for skip and test actions is trivial. The only case that requires some care is the store operation. It follows directly from the definitions of α, $\mathcal{F}_c(n, cs)$, and $\mathcal{F}(n, \alpha)$ (see a complete proof in the companion report [4]).

Corollary 1. *For each node $n \in CFG(P)$, the LFP of the dataflow analysis equations is an over approximation of the LFP of the corresponding equations in the collecting semantics.*

Termination: It follows trivially from the fact that a finite *Atom* set leads to a finite lattice (both the concrete and abstract domains are finite).

4 Effective Removal of Variables Using Dataflow Analysis

We now present the mechanism to effectively remove propositional variables in the SAT-formula. As previously mentioned, TACO removes propositional variables by introducing *tighter* upper bounds for those Alloy relations representing the initial Java memory heap. KodKod allows one to prescribe bounds for Alloy relations of any arity (unary relations included). The *DynAlloyToAlloy* translator introduces several versions of the same DynAlloy variable in order to model state change in Alloy. Our goal is to compute a tighter upper bound for each Alloy relation modelling different versions of the same DynAlloy variable.

The execution of the *DynAlloyToAlloy* translator is separated into several phases. Each phase performs a semantic preserving transformation of the DynAlloy specification. The following phases are executed in an orderly fashion:

1. **Unroll**: Removes loops by unrolling them up to the provided limit.
2. **Inline**: Replaces program invocation with the corresponding method bodies.
3. **SSA**: Applies an SSA-like transformation of the DynAlloy program.
4. **NoLocals**: Promotes local variables to program parameters.

The resulting DynAlloy specification is then translated into an Alloy representation following the rules presented in [8]. Once the single static assignment (SSA) transformation is applied, DynAlloy variables and Alloy relations match. Therefore, if we perform our value-propagation analysis on this final DynAlloy representation, we will obtain an over approximation of all possible values an Alloy relation may have.

By default, Alloy associates a conservative upper bound for each relation which was not explicitly bounded. If an upper bound is found in the repository, TACO instruments the Alloy representation by including the stored upper bound, refining the initial state. It is worth mentioning that, as the upper bounds stored in the repository can be seen as an over approximation of the values of the initial Java memory heap, we use them as a refined entry abstraction for our dataflow analysis.

The refined entry abstraction is then completed for the remaining variables depending on the intended meaning of the DynAlloy variable. For those variables modelling Java parameters, all tuples that satisfy the type definition within the scope of analysis are associated. For any other DynAlloy variable x, no tuple is associated (i.e., $a(x) = \emptyset$).

In order to properly introduce tighter bounds for all Alloy relations, we inspect the abstract value of each DynAlloy variable at the *exit* location. Given a DynAlloy variable $x \in JVar \uplus JField$, the abstract value for x at this location could be written as an upper bound of (the Alloy relation) x and fed to the KodKod input, leading to the removal of unnecessary propositional variables. For cases where x's abstract value maps to an empty set, a special measure has to be taken in order to enforce Alloy's relational constraints.

Definition 2. *Let a_{exit} be the computed abstract value for the exit of the CFG.*

$$U_x^{DF} = \begin{cases} a_{exit}(x) & \textit{if } a_{exit}(x) \neq \emptyset \\ defVal(x) & \textit{otherwise} \end{cases}$$

where $defVal(x)$ returns the default Java values (e.g, 0 for `Integer`, false for `Bool`, etc) in case x models a Java variable, and a total function whose range contains default values in case x represents a Java field.

We call U_x^{TACO} to the upper bound fed by TACO to KodKod when no dataflow analysis is performed. The following theorem ensures that the technique is safe (i.e., it does not miss faults).

Theorem 2. *Let θ be the Alloy formula output by the* DynAlloyToAlloy *translator. Given an Alloy model I such that $M[\theta \wedge \bigwedge_x (x \subseteq U_x^{TACO})]_I = \textbf{\textit{true}}$*

Then, there is an Alloy model I' such that $M[\theta \wedge \bigwedge_x (x \subseteq U_x^{DF})]_{I'} = \textbf{\textit{true}}$

Proof. Let I be an Alloy model satisfying the hypothesis. We know by definition that $defVal(x) \subseteq U_x^{DF}$. Let x be an Alloy relation such that $I(x) \subseteq U_x^{TACO} \setminus U_x^{DF}$. Due to Corollary 1, x must match a DynAlloy variable which simultaneously satisfies that: 1) x represents a local variable, and 2) no assignment nor access to x occurs within the set of traces codified by I. Therefore, x's value has no effect on the set of traces codified by I. This means that this set remains unchanged if we replace x's value with any other value (e.g. $defVal(x)$). Due to the fact that θ encodes a partial correctness assertion [8], the satisfiability of θ does not depend on x's value. Thus, $M[\theta]_I = M[\theta]_{I[x \mapsto defVal(x)]}$ by substitution. Therefore, we can define I' as:

$$I'(x) = \begin{cases} I(x) & \text{if } I(x) \subseteq U_x^{DF} \\ defVal(x) & \text{otherwise} \end{cases}$$

where $M[\theta]_I = M[\theta]_{I'}$ and $I'(x) \subseteq U_x^{DF}\Box$.

4.1 Loop Optimization

The Java while construct `while` B `do` P `od` can be expressed in DynAlloy as $(B?; P)^*; (\neg B)?$. Given a loop limit of k, the **Unroll** phase transforms the loop into:

$$\underbrace{((B?; P) + skip); \ldots; ((B?; P) + skip)}_{k-times}; (\neg B)?$$

Although semantically correct, this representation of the while construct permits several permutations. For instance: the program trace $B?; P; skip$ is equivalent to $skip; B?; P$. This apparently harmless symmetry has a tremendous impact since our dataflow analysis is branch insensitive. Due to this, the computed over approximation becomes too coarse.

This observation led us to modify the **Unroll** phase. The new nested unrolling encodes `while` B `do` P `od` into $T_k(B, P); (\neg B)?$, where T is recursively defined as:

$$T_0(B, P) = skip$$
$$T_n(B, P) = (((B?; P); T_{n-1}(B, P)) + skip)$$

5 Empirical Evaluation

In this section we present the experimental evaluation we performed in order to validate our approach. We aim at answering the following two research questions:

RQ1: Is our approach capable of outperforming the current SAT-based analyses?
RQ2: Where do the performance gains come from?

Method	Analysis Times (secs.)				Variables	
	TACO	T-Flow	Dataflow	Speed-up	# TACO	Reduction
SList.contains.s20	8.25	5.34	0.13	1.54	1743	2.70%
SList.insert.s20	9.91	11.30	0.41	0.88	3892	11%
SList.remove.s20	18.11	8.20	0.12	2.21	3749	15.92%
AList.contains.s20	29.20	8.43	0.19	3.46	3573	34.73
AList.insert.s20	10.66	9.04	0.14	1.18	4732	0.97%
AList.remove.s20	144.35	11.94	0.17	12.09	4580	11.55%
CList.contains.s20	109.7	47.53	0.16	2.31	2530	28.74%
CList.insert.s20	59.9	45.82	0.18	1.31	4512	30.78%
CList.remove.s20	1649.47	353.66	0.33	4.66	5365	11.39%
AvlTree.find s20	25.82	25.28	0.14	1.02	1412	5.95%
AvlTree.findMax.s20	1439.32	119.03	1.96	12.09	2505	2.95%
AvlTree.insert.s17	32018.14	20744.99	98.78	1.54	204799	30.33%
BinHeap.findMin.s20	109.86	10.45	0.85	10.51	2224	9.80%
BinHeap.decK.s18	18216.79	335.42	2.63	54.31	9469	1.16%
BinHeap.insert.s17	25254.66	122.22	8.52	206.63	33477	28.89%
BinHeap.extMin.s20	1149.71	451.19	21.71	2.55	55188	27.55%
TreeSet.find.s20	14475.49	769.64	1.74	18.81	3032	2.51%
TreeSet.insert.s13	26447.76	719.78	35.12	36.74	38822	1.26%
BSTree.contains.s13	28602.33	9190.95	0.19	3.11	648	0.15%
BSTree.insert.s12	7251.39	14322.00	0.44	0.51	2396	13.28%
BSTree.remove.s09	18344.38	1214.49	1.79	15.10	13369	5.42%

Table 1: Analysis times (in seconds) for TACO and TACO-Flow, speed-up and variables in the obtained propositional formula.

In order to answer these questions we implemented TACO-Flow. TACO-Flow[5] is an extension of TACO which implements the approach described in §4. That is, a new encoding of loop unrolls, a generic dataflow framework for DynAlloy programs, our value-propagation analysis as an instance of this framework, and finally, its application as a means to remove propositional variables in the Alloy intermediate representation.

We considered the benchmarks presented in [10] and compare the analysis times of TACO and TACO-Flow. We analized the following case studies: **LList**: An implementation of sequences based on singly linked lists; **AList**: The implementation `AbstractLinkedList` of interface `List` from the Apache package commons.collections, based on circular doubly-linked lists; **CList**: The implementation `NodeCachingLinkedList` of interface `List` from the Apache package commons.collections; **BSTree:** A binary search tree implementation from Visser et al. [23]; **TreeSet**: The implementation of class `TreeSet` from package `java.util`, based on red-black trees; **AVLTree**: An implementation of AVL trees obtained from the case study used in [2]; **BHeap**: An implementation of binomial heaps used as part of a benchmark in [23]; For each class we consider the most representative set of methods featuring insertion, deletion, and look-up. All methods are correct with respect to their contracts except for BHeap.extMin which contains an actual fault discovered in [10].

We analyzed mainly correct implementations since we are interested in measuring the worst case scenario for bounded-verification (i.e., search space exhaustion). The case of BHeap.extMin is included to show the analysis does not miss bugs that are catchable in the given scope.

Both TACO and TACO-Flow were fed with an initial set of upper-bounds. These upper-bounds were discovered using a cluster of computers as reported in

[5] TACO-Flow and the benchmarks are available at `http://www.dc.uba.ar/tacoflow`.

our previous work [10]. TACO-Flows used them to produce an entry abstraction for the value-propagation analysis.

We were interested in assessing the impact of the techniques in terms of analysis time and in seeing if the overhead introduced by the dataflow analysis can be compensated by the obtained performance gains.

Hardware and Software platform: All experiments were run on an Intel Core i5-570 processor running at 2.67GHz and 8GB DDR3 total main memory, on a Debian's GNU/Linux v6 operating system.

For every case study we checked that their class invariants are preserved and their method contracts are satisfied. For each method we selected the greatest scope that TACO could verify within a given time threshold (10 hours). The maximum scope is restricted to at most 20 node elements for each experiment. This is due to the fact that this is the greatest scope used for evaluating TACO in our most recent work. If loops are found, they were unrolled up to 10 times. Table 1 shows the end-to-end analysis times using both TACO and TACO-Flow, the cost of the dataflow analysis in TACO-Flow and its speed-up (the ratio TACO/TACO-Flow). The last two columns show the number of propositional variables of the SAT-formula produced by TACO and the percentage of reduction introduced by TACO-Flow.

Notice that the overall speed-up was very significant in almost all cases. More specifically, it was approximately 20 times faster in average. Only two methods exhibited a loss in performance. The required time to compute the dataflow analysis was, in general, negligible and compensated by the speed-up obtained in the end-to-end execution.

We strongly believe that the exhibited speed-up could also lead to an increase in scalability. For instance, the maximum scope that TACO successfully analyzed within the time threshold for method insert in class BinHeap was 17 node elements. In contrast, TACO-Flow was able to analyze the same scope with a speed-up of 8.52x and it easily analyzed the same method up to the maximum scope for which we had initial upper-bounds. More experimental validation is required in order to justify this claim.

Our research was driven by the hypothesis that verification times were sensitive to a decrease in the number of propositional variables. To validate that hypothesis, we collected the number of propositional variables generated by both TACO and TACO-Flow. TACO-Flow indeed reduced the number of propositional variables and the obtained performance gains appeared to confirm the hypothesis. As the reduction percentage did not seem to be directly related to the gain proportion, a further investigation of this matter is necessary.

TACO-Flow differs from TACO in the introduction of a new encoding for loop unrollings (hereinafter denoted as TACO$^+$) and the removal of propositional variables based on the dataflow analysis output. We decided to measure each contribution separately (Tables 2 and 3).

Surprisingly, TACO$^+$ showed an impressive improvement in the analysis time. We conjecture this is because the new encoding avoids a significant number of paths in the CFG leading to isomorphic valuations in the SAT-formula. For

Method	TACO vs $TACO^+$
SList.contains	1.64
SList.insert	0.64
SList.remove	1.80
AList.contains	2.86
AList.insert	1.15
AList.remove	14.10
CList.contains	1.47
CList.insert	0.69
CList.remove	3.89
AvlTree.find	0.97
AvlTree.findMax	12.72
AvlTree.insert	1.09
BinHeap.findMin	13.99
BinHeap.decK	76.94
BinHeap.insert	178.49
BinHeap.extMin	2.35
TreeSet.find	13.05
TreeSet.insert	31.24
BSTree.contains	4.12
BSTree.insert	0.60
BSTree.remove	18.26

Table 2: $TACO^+$ speed-up

Method	$TACO^+$vs TACO-Flow
SList.contains	0.94
SList.insert	1.38
SList.remove	1.23
AList.contains	1.21
AList.insert	1.03
AList.remove	0.86
CList.contains	1.57
CList.insert	1.89
CList.remove	1.20
AvlTree.find	1.06
AvlTree.findMax	0.95
AvlTree.insert	1.42
BinHeap.findMin	0.75
BinHeap.decK	1.09
BinHeap.insert	1.15
BinHeap.extMin	1.08
TreeSet.find	1.44
TreeSet.insert	0.90
BSTree.contains	0.76
BSTree.insert	0.84
BSTree.remove	1.33

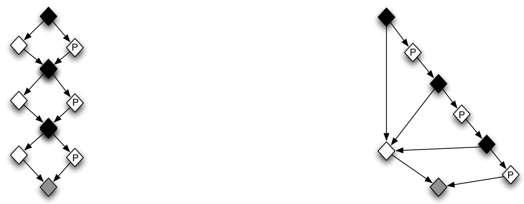
Table 3: TACO-Flow speed-up

instance, for a CFG of only one loop, the application of n loop unrollings in TACO leads to 2^n paths whereas the same application in $TACO^+$ leads to $2(n-1)$ potential paths (see Fig. 5). Even tough this result was not initially expected, it is actually a consequence of the introduction of the dataflow analysis in TACO-Flow which needs a better encoding of loop unrolls to mitigate precision loss.

Now, we focus on Table 3. For every method we took the maximum scope that $TACO^+$ could analyze and ran TACO-Flow on the same setting. It is worth noticing that, even when the improvements are less impressive than those shown in Table 2, this rather simple dataflow analysis is able to obtain significant gains. For instance, in some cases it is about 90% with approximately 16% on average.

Unfortunately, there are a few cases where some performance loss is reported. At first glance, these cases contradict our initial hypothesis accounting that a reduction in the number of propositional variables leads to performance improvement. Tighter upper-bounds are not only used by KodKod to remove propositional variables. KodKod computes a symmetry breaking predicate (SBP) based on the provided lower and upper-bounds. This SBP reduces many (but often not all) isomorphic valuations (i.e., symmetries). Our guess is that the introduction of our tighter upper bounds for the intermediate states is degrading the SBP that KodKod produces (this reasoning may also apply to the two benchmarks with no speed-ups in Table 1). This effect could be avoided by injecting the tighter upper-bounds directly at the SAT level without affecting KodKod. Although this is a clear research direction to follow, this goes beyond the proposed scope of this article.

Threats to validity: A first concern is related with the fact that we are empirically comparing the proposed approach only against our own previous work. Even though this concern is valid, we would like to point out that TACO was recently compared against several state-of-the-art SAT-based, model checkers and SMT-based verification tools [10].



(a) TACO's loop unroll encoding (b) New encoding of loop unrollings

Fig. 5: Loop unroll encodings in TACO and TACO-Flow.

A second concern is about how representative the benchmarks are. In this regard, the benchmarks we have chosen appear recurrently in case studies used by the bounded verification community [1, 7, 14, 19, 23]. In addition, the algorithms found in the case studies are commonplace. They recurrently appear in many applications [20] ranging from container classes to XML parsers. Therefore, even if it is not possible to perform general claims about all applications, they can be used as a relative measure of how well the proposed approach performs compared with other tools aiming at verifying heap manipulating algorithms.

Another threat to validity is the length of these benchmarks. They target code manipulating rather complex data structures, working at the intraprocedural level. In the presence of contracts for methods, modular SAT-based analysis could be applied by replacing method calls by their corresponding contracts and then analyzing the resulting code. This approach is followed for instance in [7].

Finally, TACO-Flow relies on having a pre-computed set of initial upper-bounds. The distributed computation cost of this artifact is significant with respect to the sequential analysis time. Nevertheless, as already mentioned, this computational cost can be amortized along time.

6 Conclusions and Further Work

In this article we presented a value-propagation analysis aiming at reducing SAT-solving verification costs. Applying this technique required the implementation of a dataflow framework in TACO. As a means to mitigate precision loss we introduced a new encoding for loops. This had an unexpected positive impact in the overall performance.

In summary, the whole approach led to an important increase of performance in the whole verification process. More experimentation is required to assess with confidence whereas the approach is capable of increasing the scope of analysis beyond the current state-of-the-art.

We strongly believe there is still room for reducing verification cost by relying on dataflow analyses. For instance, an alias analysis can be used to rule-out infeasible valuations. We are currently implementing this analysis using our framework. Initial results seems to be promising. We also want to check our conjecture about KodKod's symmetry breaking predicate. The current TACO prototype removes variables at the Alloy level. We plan to develop a new prototype that could remove them at the SAT-level. We believe that this will mitigate the observed performance loss.

References

1. Boyapati C., Khurshid S., Marinov D., *Korat: automated testing based on Java predicates*, in ISSTA 2002, pp. 123–133.
2. Belt J., Robby and Deng X., *Sireum/Topi LDP: A Lightweight Semi-Decision Procedure for Optimizing Symbolic Execution-based Analyses*, FSE'09, pp. 355–364.
3. Biere, A., Heule M., van Maaren H., *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, vol. 185, 2009.
4. Cuervo Parrino, B., Galeotti, J.P., Garbervetsky, D., Frias, M. *A dataflow analysis to improve SAT-based program verification* Technical Report. `http://www.dc.uba.ar/tacoflow/techrep.pdf`. May 2011.
5. Cytron, R. Ferrante, J., Rosen, B. K., Wegman, M. N. and Zadeck, F. K., *Efficiently computing static single assignment form and the control dependence graph*, ACM TOPLAS, 13(4), pp. 451–490.
6. Chalin P., Kiniry J.R., Leavens G.T., Poll E. *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2.* FMCO'05: pp. 342-363.
7. Dennis, G., Yessenov, K., Jackson D., *Bounded Verification of Voting Software.* In VSTTE '08, pp. 130–145. 2008.
8. Frias, M. F., Galeotti, J. P., Lopez Pombo, C. G., Aguirre, N., *DynAlloy: Upgrading Alloy with Actions*, in ICSE'05, pp. 442–450, 2005.
9. Galeotti J., *Software Verification Using Alloy.* PhD. Thesis, UBA, 2011.
10. Galeotti J.P., Rosner N., López Pombo C.G., and Frias M.F., *Analysis of invariants for efficient bounded verification.* In Proceedings of ISSTA'10, pp. 25–36.
11. Harel D., Kozen D., and Tiuryn J. *Dynamic logic. Foundations of Computing.* MIT Press, 2000
12. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P., *F-Soft: Software Verification Platform.* In CAV'05, pp. 301–306, 2005.
13. Jackson, D., *Software Abstractions.* MIT Press, 2006.
14. Jackson, D., Vaziri, M., *Finding bugs with a constraint solver*, in ISSTA'00, pp. 14–25, 2000
15. Kindall, Garia A. *A unified approach to global program optimization.* in POPL '73. pp. 194–206.
16. Mendonça de Moura L., Bjørner N. *Z3: An Efficient SMT Solver.* TACAS'08, pp. 337–340.
17. Nielson, F. *A denotational framework for data flow analysis.* Acta Inf. 18 (1982), pp. 265–287.
18. Shao D., Gopinath D., Khurshid S. and Perry D.E. *Optimizing Incremental Scope-Bounded Checking with Data-Flow Analysis.* In ISSRE'10, pp. 408–417.
19. Sharma R., Gligoric M., Arcuri A., Fraser G., Marinov D.*Testing Container Classes: Random or Systematic?*, in FASE 2011. March 2011.
20. Siddiqui, J. H., Khurshid, S., *An Empirical Study of Structural Constraint Solving Techniques*, in ICFEM'09, LNCS 5885, 88–106, 2009.
21. Torlak E., Jackson, D., *Kodkod: A Relational Model Finder.* in TACAS '07, LNCS 4425, pp. 632–647.
22. Taghdiri, M., Seater, R., Jackson D. *Lightweight extraction of Syntactic Specifications.* In FSE '06, pp. 276–286.
23. Visser W., Păsăreanu C. S., Pelánek R., *Test Input Generation for Java Containers using State Matching*, in ISSTA'06, pp. 37–48.
24. Xie, Y., Aiken, A., *Saturn: A scalable framework for error detection using Boolean satisfiability.* in ACM TOPLAS, 29(3): (2007)