Noname manuscript No.
(will be inserted by the editor)

TacoFlow: Optimizing SAT Program Verification Using Dataflow Analysis

Bruno Cuervo Parrino[1], **Juan Pablo Galeotti**[2], **Diego Garbervetsky**[1,3], **Marcelo F. Frias**[3,4]
e-mail: `bcuervo@dc.uba.ar, galeotti@cs.uni-saarland.de, diegog@dc.uba.ar, mfrias@itba.edu.ar`

[1] Departmento de Computación, FCEyN, UBA
[2] Saarland University
[3] CONICET
[4] Department of Software Engineering, Instituto Tecnológico de Buenos Aires.

The date of receipt and acceptance will be inserted by the editor

Abstract In previous work we presented TACO, a tool for efficient bounded verification. TACO translates programs annotated with contracts to a SAT problem which is then solved resorting to off-the-shelf SAT-solvers. TACO may deem propositional variables used in the description of a program initial states as being unnecessary. Since the worst-case complexity of SAT (a known NP problem) depends on the number of variables, most times this allows us to obtain significant speed ups. In this article we present TacoFlow, an improvement over TACO that uses dataflow analysis in order to also discard propositional variables that describe intermediate program states. We present an extensive empirical evaluation that considers the effect of removing those variables at different levels of abstraction, and a discussion on the benefits of the proposed approach.

1 Introduction

Bounded verification [12] is a highly automatic verification 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 verification tools [12, 17, 19, 33] 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 one to find a valuation for the propositional variables that encodes a failure. Theoretically, SAT-solving time grows exponentially with respect to the number of propositional variables. However, modern SAT solvers achieve better results on practical instance problems.

The number of clauses and propositional variables in the resulting propositional formulas 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 significant impact on the overall verification cost.

Dataflow analysis [23] is a well-know and widely spread static analysis technique used for program understanding and optimization. Essentially, it is used to infer 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 and constants propagation. These analyses enable compilers to eliminate dead code, reduce unnecessary runtime checks and remove redundant computations, among other things.

Dataflow analysis can also be applied to the code under analysis as a means to obtain an optimized version before applying the (SAT-based) bounded verification. In this work we analyze this hypothesis by introducing a dataflow analysis into the TACO [17] verifier tool chain.

TACO is a SAT-based tool specially aimed at verifying Java sequential programs. The current TACO distribution supports specifications written in Java Modeling Language (JML) [6] or in the JForge Specification Language (JFSL) [34]. TACO accurately represents all Java data types (including primitive types such as double and float) and supports nearly all JML and JFSL syntax. TACO does not report false positives, but it isn't able to prove the absence of errors above the given scope of analysis. Among its features, TACO introduces a novel technique for reducing the number of propositional variables without harming precision. 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 SAT propositional variables.

We introduced a dataflow framework and implemented 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.

TACO's original representation for program loops was implemented as a simple sequence of `if` statements. This representation introduced at least one join point per loop unrolling, and impacted negatively in the performance of the overall dataflow analysis. Thus, 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. We also analyzed the effect of removing the unnecessary variables at different stages of the process of translating the program into a propositional formula, in order to identify the best place where to apply the optimization and understand whether it has any kind of impact in the other optimizations involved during the verification process.

Contributions: The technical contributions of this article include:

- A formalization of a dataflow analysis for propagating values through a program control flow graph, including the proof showing that the outcome of the analysis is a sound over approximation of the program behavior.
- A proof of the fact that the propositional formula obtained by TACO relying on this analysis is equisatisfiable with respect to the unoptimized formula.
- TacoFlow: 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.
- The application of the optimization at different stages of the verification, requiring the implementation of proper mappings between program variables and their logical representations both at the relational level as well as the propositional logic level.
- An empirical evaluation using benchmarks accepted by the bounded verification community [17] showing an important speed-up in verification time.

Outline: §2 introduces the foundations of SAT-based verification and TACO, then it presents the problem we intend to address in the current work. §3 presents the syntax and semantics of DynAlloy, an intermediate language used by TACO that is required for the formalization and proof of the the value-propagation analysis which is presented in §4. §5 shows how this technique is applied in the context of TACO. §6 shows our experimental results. In §7 we analyze some related work. Finally, §8 concludes and discusses future work.

2 Tight Bounds for Improved SAT-solving

TACO [17] is a tool for Java code verification based on SAT-solving. Given a *verification scope* bounding the sizes of the object domains, and a limit for the number of loop iterations, TACO translates a program annotated with JML or JFSL specifications into a propositional formula.

Figure 1 shows a JFSL declaration of a singly linked list data structure. JFSL has its roots in the Alloy specification language [21]; it resembles JML but with a relational flavor. The declared list contains a `header` field referring to its first node. Each node is linked to its next node in the list by the `next` field. The List container is annotated with a JFSL object invariant which constrains the set of valid linked structures to those that form a finite acyclic sequence of Node elements. JFSL allows quantifiers (e.g., `all`, `some`). The construct `*f` denotes the relational transitive closure of field `f`. In other words, the JFSL expression `expr.*f` comprises the set of all references that are reachable from a location `expr` using field `f`. Since the user may be interested in denoting the set of references that are not equal to the value `null`, the minus (`-`) operator can be used to remove elements from this set. Similarly, membership in a given set can be tested through the construct `in`.

Method `removeLast` (shown in Fig. 3a) removes the last element of the list (if such an element exists). Both JML and JFSL allow one to write partial specifications. In this example, the JFSL `@Ensures` annotation only specifies that the returned Node element should not be reachable from the receiver list.

As shown in Fig. 2, TACO translates the Java code annotated with the JML or JFSL contract into a DynAlloy specification [15]. DynAlloy is a relational specification language. This means that every variable in DynAlloy can be seen as a relation of a fixed arity. For the `removeLast`

```
class Node {
  Node next;
}
class List {
  Node header;

  @Invariant("all n: Node | n in this.header.*next - null
      implies not (n in n.*next)")
}
```

Figure 1: A singly linked list declaration

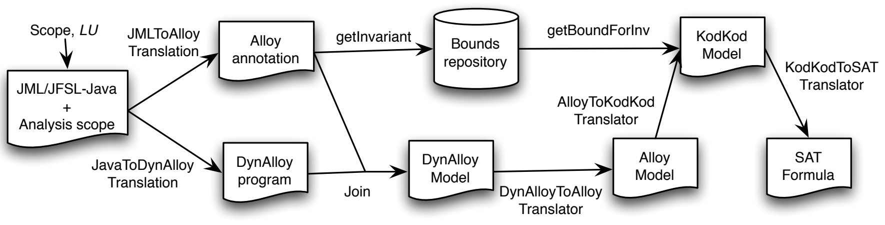


Figure 2: Translating annotated code to SAT.

```
@Ensures("return !in
     this.header.*next - null")
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

Figure 3: Java implementation and its DynAlloy representation

method, the resulting DynAlloy program is shown in Fig. 3b. Signatures *List* and *Node* are introduced to model the corresponding 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 modeled as unary noempty relations (i.e., `one S`). The following DynAlloy variables are also introduced to model removeLast's Java variables and fields:

```
// DynAlloy variables modeling Java fields
header: List -> one (Node+null)
next:   Node -> one (Node+null)
// DynAlloy variables modeling arguments and local variables
this:   one List
prev:   one Node+null
curr:   one Node+null
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 [15]. Alloy is defined in terms of a simple relational semantics, its syntax includes constructs ubiquitous in object-oriented notations, and it features automated analysis capabilities by resorting to the Alloy Analyzer [20]. Given the *static* nature of Alloy, DynAlloy was devised as a procedural extension of Alloy. In order to model state change in Alloy, the *DynAlloyToAlloyTranslator* may introduce several **Alloy relations** to represent different values (or *incarnations*) of the same DynAlloy variable in an SSA-like form [1].

In order to translate the Alloy specification into a SAT-problem, the Alloy Analyzer translates every Alloy relation into a set of **propositional**

variables. Each propositional variable is intended to model whether 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 verification, 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%) propositional 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 [31] 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.

Since tighter upper bounds contain fewer propositional variables, they contribute to the SAT-based analysis. 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*. Replacing these variables does not alter the validity of the SAT problem since pairs that are not in U_f cannot belong to f, and it allows us to reduce the number of propositional variables. Given the fact that the worst-case time complexity of SAT is exponential on the number of propositional variables in the SAT-formula, it is often the case that reducing the number of propositional variables leads to smaller analysis times.

Given a class invariant and the desired scope of verification, TACO is able to automatically synthesize a tighter upper bound for the initial state. In the example, only acyclic data structures are allowed due to the invariant annotation. This forbids all tuples of the form $\langle N_i, N_i\rangle$ to belong to relation $next_0$ in any Alloy model. For each tuple in the scope of verification, TACO

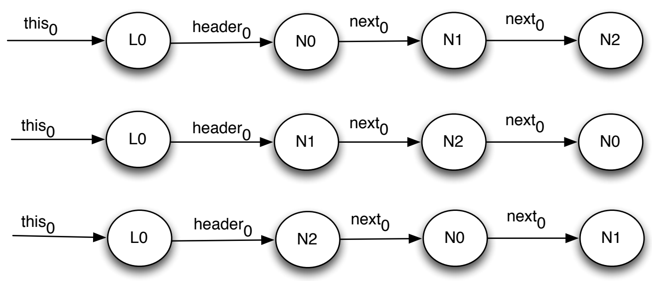


Figure 4: Three isomorphic Alloy instances for a singly linked list

creates an Alloy formula and tests its validity. If the Alloy formula is valid, then the tuple is infeasible under the current class invariant. Therefore, this tuple may be safely removed from the upper bound.

More tuples may be infeasible due to symmetry restrictions. A *symmetry* is a permutation of the signature atoms. One way to avoid considering permutations of an Alloy model is by introducing *symmetry breaking predicates.* As this may contribute to significant reductions in SAT-solving time [11], KodKod (Alloy's backend) includes general-purpose symmetry breaking [31]. TACO adds more symmetry restrictions ruling out all permutations of the Java memory heap [17]. This is possible due to the fact that we are dealing with an Alloy representation of the initial Java memory heap. Following the example, given the isomorphic Alloy instances shown in Fig. 4, TACO symmetry breaking predicates will prevent all valuations except the first Alloy instance. Therefore, tuples such as $\langle N_1, N_0 \rangle$ and $\langle N_2, N_1 \rangle$ will not belong to any valid Alloy valuation of relation $next_0$. In turn, the feasibility analysis for these tuples will remove them from the upper bound. In the presented example, TACO discovers a tighter upper bound that removes over 70% of the propositional variables representing the initial state.

As shown in Fig. 2, tighter upper bounds discovered by TACO are stored in a repository. Since bounds are often reused during the analysis of different methods in a class, the cost of computing them is amortized. Moreover, the feasibility of a tuple does not depend on the feasibility of other tuples enabling the parallel computation of bounds and, thus, obtaining significant performance gains.

2.1 Problem Statement

The technique introduced in [17] limits itself to provide bounds only to those propositional variables that 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 behavior at various points in a program. For instance, we could conclude that (under the scope of verification 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 conservatively over approximate the set of possible values every Java variable and field may store within the provided scope of verification. Using this analysis we can propagate the upper bounds from the initial state to the intermediate states. The resulting tight upper bounds for the intermediate Alloy relations 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 §6, this technique leads to a potential improvement in the performance of the SAT-based verification.

3 DynAlloy: An intermediate language for program analysis

In this section we present the syntax and semantics of DynAlloy, the intermediate representation of TACO. DynAlloy is an extension of Alloy based on first-order dynamic logic [18]. We chose DynAlloy as a target for performing 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.

Before describing DynAlloy we will briefly comment on the Alloy language, which is the core language used in DynAlloy expressions.

3.1 Alloy

A brief overview of Alloy's syntax and semantics taken from [20] is given in Fig. 5. Alloy was designed with the goal of making specifications automatically analyzable. This has made the Alloy Analyzer an appealing tool for an active and growing community.

We refer the reader to [21] for a complete description of the Alloy language. Alloy's semantics is defined in terms of sets and relational operations. Each relation is a set of tuples of an arbitrary (but fixed) arity. The language provides first order logic quantification (*all* and *some* formulas) as well as transitive closure (the $+expr$ operator). The navigation operator ($expr.expr$) resembles object oriented notations. The semantics of Alloy is given in terms of two functions: X, which maps Alloy expressions to relations, and M, which maps formulas to their truth value.

problem ::= decl* formula
decl ::= *var* : *typexpr*
typexpr ::=
type
| *type* $\rightarrow$ *type*
| *type* $\Rightarrow$ *typexpr*

formula ::=
expr *in* expr (subset)
|!formula (neg)
| formula && formula (conj)
| formula || formula (disj)
| *all v* : *type* | formula (univ)
| *some v* : *type* | formula (exist)

expr ::=
expr + expr (union)
| expr & expr (intersection)
| expr − expr (difference)
|∼ expr (transpose)
| expr.expr (navigation)
| +expr (transitive closure)
| {*v* : *t* | formula} (set former)
| *Var*

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

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

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

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

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

$R \rightarrow S$ denotes the Cartesian product between relations R and S. $R{+}{+}S$ denotes the relational overriding defined as follows[1]:

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

$[\cdot]_0$ is the unary operator that returns the only element present in a singleton set: $[\{a\}]_0 = a$.

Figure 5: Grammar and semantics of Alloy

3.2 DynAlloy

The aim of the DynAlloy specification language is to provide a formal characterization of imperative sequential programs while keeping the relational semantics of Alloy. Fig. 6 shows a relevant fragment of DynAlloy's grammar. DynAlloy extends Alloy by allowing the user to write partial correctness assertions of the form {formula}*program*{formula}. This fragment corresponds to the DynAlloy programs output by the *JavaToDynAlloy* translator, part of TACO. As shown in Fig. 3b, typical structured programming constructs can be described using these basic logical constructs. For example, `if` B `then` P `else` Q `fi` can be written as the DynAlloy program $B?; P + (\neg B)?; Q$. Similarly, `while` B `do` P `od` can be expressed as $(B?; P)^*; (\neg B)?$.

More complex programming language features such as dynamic memory allocation can also by modeled using grammar shown in Figure 6. For instance, allocating a fresh object of type T can be written as the following DynAlloy test program:

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

Figure 6: Relevant DynAlloy fragment

```
( x in T && x !in (l1+...+ln).*(f1+...+fm) )?
```

where $l_1, \ldots, l_n$ is the set of all alive variables (including static fields) and $f_1, \ldots, f_m$ is the set of all fields of any type. Observe that the set `(l1+...+ln).*(f1+...+fm)` denotes a transitive closure with all objects that might be accessed by the program. If the target programming language semantics includes object initialization (like Java), the field initialization is performed afterwards using store actions.

3.3 DynAlloy relational semantics

Since DynAlloy's 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.

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

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

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

4 Propagating Values in DynAlloy Programs

In this section, we present a dataflow analysis technique for DynAlloy programs that computes an over approximation of all the possible variable assignments for every program location.

4.1 Collecting Semantics

Now we present an alternative definition based on the collecting semantics [25] which is useful for proving the correctness of our proposed dataflow technique.

We start by defining two helper functions we will use for updating states.

Definition 1 *Given a concrete state $c \in E$, variables $x, y \in JVar \uplus JField$ and a value $v \in Atom \cup \mathcal{P}(Atom \times Atom)$, we define the update of a state c as follows:*

$$c[x \mapsto v](y) = \begin{cases} v & \textit{when } y = x, \\ c(y) & \textit{otherwise.} \end{cases}$$

The transfer function is used to explain how a concrete state changes when a DynAlloy statement is executed. We denote by C the power set of concrete states (namely, $C = \mathcal{P}(E)$).

Definition 2 *The transfer function $\mathcal{F} : DynAlloyProgram \times C \to C$ is the following:*

$$\begin{aligned}
\mathcal{F}(skip, cs) &= cs, \\
\mathcal{F}(\phi?, cs) &= \{c \mid c \in cs \wedge M[\phi]_c\}, \\
\mathcal{F}(v := expr, cs) &= \{c[v \mapsto [X[expr]_c]_0] \mid c \in cs\}, \\
\mathcal{F}(v.f := expr, cs) &= \{c[f \mapsto c(f){+}{+}(\{c(v)\} \to X[expr]_c)] \mid c \in cs\}.
\end{aligned}$$

Observe that for the case of the variable assignment we use $[X[expr]_c]_0]$. This is because we know that in a state a variable has exactly one possible value.

Given a DynAlloy program P it is possible to obtain its control flow graph (CFG). We assume the CFG has a unique entry (respectively, exit) point without incoming (respectively, outgoing) edges.

A collecting semantics defines how information flows through the CFG. In the collecting semantics, every time a new value traverses a node it is recorded. Therefore, each node keeps track of all values passing through it.

Definition 3 *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:

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

$$out(n) = \mathcal{F}(n, in(n)).$$

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

4.2 Abstract Semantics

We represent an *abstract state* as a mapping of DynAlloy variables to their corresponding abstract values.

Definition 4 *The abstract value of a DynAlloy variable representing a Java variable is a set of atoms. Therefore, we define A, the abstract domain, as*

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

An abstract value represents the set of all concrete values a DynAlloy variable *may* take (i.e., an over approximation) at a given program location. In order to operate with this abstraction we need A to be endowed with a lattice structure.

Definition 5 *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* $\forall x \in JVar \uplus JField.(a''(x) = a(x) \cup a'(x))$.

Notice that $\langle A, \sqsubseteq \rangle$, as defined in Def. 5, is indeed a lattice.

We define abstract state update similarly to concrete state update.

Definition 6 *Given an abstract state $a \in A$, variables $x, y \in JVar \uplus JField$ and a value $v \in \mathcal{P}(Atom) \cup \mathcal{P}(Atom \times Atom)$, we define the update of a state a as follows:*

$$a[x \mapsto v](y) = \begin{cases} v & \text{when } y = x, \\ a(y) & \text{otherwise.} \end{cases}$$

Definition 7 *The abstraction function $\alpha : C \to A$ formalizes the notion of approximation of concrete states by an abstract state. Given a set of concrete states $cs \in C$,*

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

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 the dual of α. Namely,

$$\gamma(a) = \bigcup \{cs \mid \alpha(cs) \sqsubseteq a\}. \tag{1}$$

Proposition 1 $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *is a Galois connection.*

Proof: We resort to [10, Prop. 7] which guarantees this property when α is a complete join morphism and γ is defined using α exactly as we did in (1).

The proof that α is a complete join morphism is straightforward by following the definitions of α, $\sqcup$ and using that $\cup$ is a complete join-morphism.

$$\alpha(\bigcup_{cs \in C'} cs)(v) = \{c(v) \mid c \in \bigcup_{cs \in C'} cs\} = \bigcup_{cs \in C'} \{c(v) \mid c \in cs\}.$$

On the other hand,

$$\begin{aligned} \bigsqcup_{cs \in C'} \alpha(cs)(v) &= \bigsqcup_{cs \in C'} \{c(v) \mid c \in cs\} \\ &= \bigcup_{cs \in C'} \{c(v) \mid c \in cs\}. \end{aligned}$$

■

Let $\hat{X}$ be the *abstract evaluation function*. It agrees with X, with the exception that for a variable v, $X[v]_c$ returns the singleton $\{c(v)\}$ while $\hat{X}[v]_a$ returns the set $a(v)$.

Definition 8 *The* abstract transfer function $\hat{\mathcal{F}} : \mathit{DynAlloyProgram} \times A \to A$ *is defined by the following rules:*

$$\begin{aligned}
&\hat{\mathcal{F}}(\mathit{skip}, a) &&= a,\\
&\hat{\mathcal{F}}(\phi?, a) &&= a,\\
&\hat{\mathcal{F}}(v := \mathit{expr}, a) &&= a[v \mapsto \hat{X}[\mathit{expr}]_a],\\
&\hat{\mathcal{F}}(v.f := \mathit{expr}, a) &&= \textbf{\textit{let}}\ \mathit{from} = a(v), \mathit{to} = \hat{X}[\mathit{expr}]_a\ \textbf{\textit{in}}\\
&\qquad \textbf{\textit{if}}\ |\mathit{from}| = 1,\\
&\qquad\qquad \textbf{\textit{then}}\ a[f \mapsto a(f) {+}{+} (\mathit{from} \to \mathit{to})]\ \textit{(strong update)}\\
&\qquad\qquad \textbf{\textit{else}}\ a[f \mapsto a(f) \cup (\mathit{from} \to \mathit{to})]\ \textit{(weak update).}
\end{aligned}$$

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.

Definition 9 *We define the analysis dataflow equations using $\hat{\mathcal{F}}$ as follows. For each node n in the CFG of P,*

$$\hat{in}(n) = \begin{cases} \alpha(\{c_0 | M[\phi]_{c_0}\}) & n \text{ is the entry of } CFG(P), \\ \bigsqcup_{p \text{ in } pred(n)} \hat{out}(p) & \text{otherwise} . \end{cases}$$

$$\hat{out}(n) = \hat{\mathcal{F}}(n, in(n)).$$

4.3 Correctness

Here we show that the abstraction is a sound over approximation of the collecting semantics. Before the proof we introduce a lemma that will be handy for proving the correctness of our analysis.

Lemma 1 *Let $cs \in C$, $a \in A$ and let* expr *in domain X. Then,*

$$\alpha(cs) \sqsubseteq a \implies \forall c \in cs \,.\, X[expr]_c \subseteq \hat{X}[expr]_a.$$

Proof: We need to prove the property for the three possible forms of *expr*: *null*, v and $v.f$.

expr $= null$: $\forall c \in cs \,.\, X[null]_c = \{\langle null \rangle\} = \hat{X}[null]_a$.
Then, $\forall c \in cs \,.\, X[null]_c \subseteq \hat{X}[null]_a$.
expr $= v$: Since $\alpha(cs) \sqsubseteq a$, $\alpha(cs)(v) \subseteq a(v)$. Then, by Def. 7, $\{c(v) \mid c \in cs\} \subseteq a(v)$. Thus, $\forall c \in cs \,.\, \{c(v)\} \subseteq a(v)$.
Since $X[v]_c = \{c(v)\}$ and $\hat{X}[v]_a = a(v)$, $\forall c \in cs \,.\, X[v]_c \subseteq \hat{X}[v]_a$.
expr $= v.f$: Since $\alpha(cs) \sqsubseteq a$, $\alpha(cs)(v) \subseteq a(v)$ and $\alpha(cs)(f) \subseteq a(f)$. By Def. 7, $\{c(v) \mid c \in cs\} \subseteq a(v)$ and $\bigcup_{c \in cs} c(f) \subseteq a(f)$. Then, $\forall c \in cs \,.\, \{c(v)\} \subseteq a(v)$ and $c(f) \subseteq a(f)$.
By definition of operator $;$, $\{c(v)\}; c(f) \subseteq a(v); a(f)$. Since $X[v.f]_c = \{c(v)\}; c(f)$ and $\hat{X}[v.f]_a = a(v); a(f)$, $\forall c \in cs \,.\, X[v.f]_c \subseteq \hat{X}[v.f]_a$.

■

We also introduce a lemma concerning the relational override operator ++. We will apply this lemma along the proof of correctness of the strong update operator.

Lemma 2 *Given $R_1 \subseteq R_2$, and $S_1 \subseteq S_2$. If $dom(S_1) = dom(S_2)$, then,*

$$R_1{+}{+}S_1 \subseteq R_2{+}{+}S_2.$$

The proof is straightforward following the definition of the ++ operator introduced in Fig. 5 .

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

$$\alpha(cs) \sqsubseteq a \implies \alpha(\mathcal{F}(n, cs)) \sqsubseteq \hat{\mathcal{F}}(n, a).$$

Proof: We prove it for the different types of statements (nodes in the CFG):

$n =$ *skip*: $\alpha(\mathcal{F}(skip, cs)) = \alpha(cs)$. By hypothesis, $\alpha(cs) \sqsubseteq a = \hat{\mathcal{F}}(skip, a)$.
$n = \phi?$: $\mathcal{F}(\phi?, cs) = \{c \mid c \in cs \wedge M[\phi]_c\} \subseteq cs$. Then, by monotonicity of α, $\alpha(\mathcal{F}(\phi?, cs)) \sqsubseteq \alpha(cs)$. By hypothesis, $\alpha(cs) \sqsubseteq a = \hat{\mathcal{F}}(\phi?, a)$.
$n =$ $v := expr$: By hypothesis, $\alpha(cs) \sqsubseteq a$. Then, by Def. 5,

$$\forall x \in JVar \uplus JField \,\centerdot\, \alpha(cs)(x) \subseteq a(x). \qquad (2)$$

First, let $x \in JField$. Then, by Def. 2, $\alpha(\mathcal{F}(v := expr, cs))(x) = \alpha(\{c[v \mapsto [X[expr]_c]]_0 \mid c \in cs\})(x)$ which, by Def. 7, equals

$$\bigcup_{c \in cs} c[v \mapsto [X[expr]_c]]_0(x).$$

Observe that $x \in JField$ and $v \in JVar$. Since $JField \cap JVar = \emptyset$, it follows that $x \neq v$. Therefore,

$$\bigcup_{c \in cs} c[v \mapsto [X[expr]_c]]_0(x) = \bigcup_{c \in cs} c(x).$$

Since $x \neq v$, by hypothesis,

$$\bigcup_{c \in cs} c(x) \subseteq a(x) = a[v \mapsto \hat{X}[expr]_a](x). \qquad (3)$$

By Def. 8, $\hat{\mathcal{F}}(v := expr, a) = a[v \mapsto \hat{X}[expr]_a]$. Then,

$$\alpha(\mathcal{F}(v := expr, cs))(x) \subseteq \hat{\mathcal{F}}(v := expr, a)(x).$$

Let now $x \in JVar$. Then, by Def. 2,

$$\alpha(\mathcal{F}(v := expr, cs))(x) = \alpha(\{c[v \mapsto [X[expr]_c]]_0 \mid c \in cs\})(x)$$

which, by Def. 7, equals

$$\{c[v \mapsto [X[expr]_c]]_0(x) \mid c \in cs\}. \qquad (4)$$

We now consider the cases when $v = x$ and $v \neq x$ in order to prove that

$$\{c[v \mapsto [X[expr]_c]]_0(x) \mid c \in cs\} \subseteq a[v \mapsto \hat{X}[expr]_a](x).$$

Consider (4). Splitting into the cases $v = x$ and $v \neq x$,

$$(4) = \{[X[expr]_c]_0 \mid c \in cs \wedge v = x\} \cup \{c(x) \mid c \in cs \wedge v \neq x\}.$$

First, note that by Lemma 1 we know $\forall c \in cs \,\centerdot\, X[expr]_c \subseteq \hat{X}[expr]_a$. In particular, since we applied it to a subset, when $v = x$

$$\{[X[expr]_c]]_0) \mid c \in cs \wedge v = x\} \subseteq \hat{X}[expr]_a = a[v \mapsto \hat{X}[expr]_a](x) \qquad (5)$$

By fact (2), when $v \neq x$, we prove

$$\{c(w) \mid c \in cs \wedge v \neq x\} \subseteq a(x) = a[v \mapsto \hat{X}[expr]_a](x). \qquad (6)$$

Therefore, by facts (3), (5) and (6),

$$\alpha(\mathcal{F}(v := expr, cs))(x) \subseteq a[v \mapsto \hat{X}[expr]_a](x) = \hat{\mathcal{F}}(v := expr, a)(x).$$

Then, by Def. 5,

$$\forall x \in JVar \uplus JField \,\centerdot\, \alpha(\mathcal{F}(v := expr, cs))(x) \sqsubseteq \hat{\mathcal{F}}(v := expr, a)(x).$$

$n = v.f := expr$: We will use an approach similar to the one we used for proving the previous case, but considering instead that the definition of $\hat{\mathcal{F}}(v.f := expr, a)$ distinguishes the case for strong and weak updates. Therefore, we need to prove that the inclusion holds in these two cases.

First case: $|a(v)| = 1$ (strong update).
By Def. 8,

$$\hat{\mathcal{F}}(v.f := expr, a) = a[f \mapsto a(f){+}{+}(a(v) \to \hat{X}[expr]_a)].$$

Let us consider first the case in which $x \in JVar$. Then,

$$\begin{aligned} &\alpha(\mathcal{F}(v.f := expr, cs))(x) \\ &\quad = \alpha(\{c[f \mapsto c(f){+}{+}(\{c(v)\} \to X[expr]_c) \mid c \in cs\})(x) \quad \text{(By Def. 2)} \\ &\quad = \{c[f \mapsto c(f){+}{+}(\{c(v)\} \to X[expr]_c)](x) | c \in cs\}. \qquad \text{(By Def. 7)} \end{aligned}$$

Observe that $x \in JVar$ and $f \in JField$, since $JField \cap JVar = \emptyset$, it follows that $x \neq f$. Therefore,

$$\{c[f \mapsto c(f){+}{+}(\{c(v)\} \to X[expr]_c)](x) | c \in cs\} = \{c(x) | c \in cs\}.$$

Since $x \neq f$, by hypothesis,

$$\{c(x) | c \in cs\} \subseteq a(x) = a[f \mapsto a(f){+}{+}(a(v) \to \hat{X}[expr]_a)](x). \tag{7}$$

We now consider the case in which $x \in JField$. Then,

$$\begin{aligned} &\alpha(\mathcal{F}(v.f := expr, cs))(x) = \\ &\quad \alpha(\{c[f \mapsto c(f){+}{+}(\{c(v)\} \to X[expr]_c)] | c \in cs\})(x) = \\ &\quad \bigcup_{c \in cs} c[f \mapsto c(f){+}{+}(\{c(v)\} \to X[expr]_c)](x). \end{aligned} \tag{8}$$

Consider (8). Splitting into the cases $f = x$ and $f \neq x$:

$$(8) = \bigcup_{f = x \wedge c \in cs} c(f){+}{+}(\{c(v)\} \to X[expr]_c) \cup \bigcup_{f \neq x \wedge c \in cs} c(x).$$

Let us analyze first the case when $f = x$. Since we know by Def. 5 that $\forall x \in JVar \uplus JField \centerdot \alpha(cs)(x) \subseteq a(x)$, it holds that $\forall c \in cs$:

1. $c(f) \subseteq a(f)$ (since $f \in JField$, $c(f), a(f) \in \mathcal{P}(Atom \times Atom)$).
2. Since $|a(v)| = 1$ and $\{c(v)\} \subseteq a(v)$, it follows $\{c(v)\} = a(v)$.
3. By Lemma 1, $X[expr]_c \subseteq \hat{X}[expr]_a$.

It then follows that $\{c(v)\} \to X[expr]_c \subseteq a(v) \to \hat{X}[expr]_a$. The conditions are then given for the application of Lemma 2, namely,

- $c(f) \subseteq a(f)$
- $\{c(v)\} \to X[expr]_c \subseteq a(v) \to \hat{X}[expr]_a$
- $dom(\{c(v)\} \to X[expr]_c) = dom(a(v) \to \hat{X}[expr]_a)$

Therefore, by Lemma 2,

$$c(f){+}{+}(\{c(v)\} \to X[expr]_c) \subseteq a(f){+}{+}(a(v) \to \hat{X}[expr]_a).$$

Furthermore,

$$\begin{aligned}\bigcup_{f=x \wedge c\in cs} & c(f){+}{+}(\{c(v)\} \to X[expr]_c) \\ & \subseteq a(f){+}{+}(a(v) \to \hat{X}[expr]_a) \\ & = a[f \mapsto a(f){+}{+}(a(v) \to \hat{X}[expr]_a)](f) \\ & = \hat{\mathcal{F}}(v.f := expr, a)(f). \qquad (9)\end{aligned}$$

Finally, we analyze the case when $f \neq x$. By hypothesis,

$$\bigcup_{c\in cs \wedge f\neq x} c(x) \subseteq a(x).$$

If $f \neq x$, then $a[f \mapsto a(f){+}{+}(a(v) \to \hat{X}[expr]_a)](x) = a(x)$. Then,

$$\begin{aligned}\bigcup_{c\in cs \wedge f\neq x} c(x) & \subseteq a[f \mapsto a(f){+}{+}(a(v) \to \hat{X}[expr]_a)](x) \\ & = \hat{\mathcal{F}}(v.f := expr, a)(x). \qquad (10)\end{aligned}$$

By (7), (9) and (10), we conclude

$$\forall x \in JVar \uplus JField\,\centerdot\, \alpha(\mathcal{F}(v.f := expr, cs))(x) \subseteq \hat{\mathcal{F}}(v.f := expr, a)(x).$$

Therefore, by Def. 7 we conclude, for the strong update case,

$$\alpha(\mathcal{F}(v.f := expr, cs)) \subseteq \hat{\mathcal{F}}(v.f := expr, a).$$

Second case: $|a(v)| > 1$ (weak update).
By Def. 8, $\hat{\mathcal{F}}(v.f := expr, a) = a[f \mapsto a(f) \cup (a(v) \to \hat{X}[expr]_a)]$. We will again consider the cases in which $x \in JVar$ or $x \in JField$.
If $x \in JVar$, an analogous reasoning to the one we applied for the strong update case allows us to conclude that

$$\alpha(\mathcal{F}(v.f := expr, cs))(x) \subseteq a(x) = a[f \mapsto a(f){+}{+}(a(v) \to \hat{X}[expr]_a)](x).$$

Since $x \neq f$, $a(x) = a[f \mapsto a(f) \cup (a(v) \to \hat{X}[expr]_a)](x)$. Therefore,

$$\alpha(\mathcal{F}(v.f := expr, cs))(x) \subseteq a[f \mapsto a(f) \cup (a(v) \to \hat{X}[expr]_a)](x). \qquad (11)$$

If $x \in JField$, then

$$\begin{aligned} & \alpha(\mathcal{F}(v.f := expr, cs))(x) \\ \text{(by Def. 2)} \quad & = \alpha(\{c[f \mapsto c(f){+}{+}(\{c(v)\} \to X[expr]_c)] | c \in cs\})(x) \\ \text{(by Def. 7)} \quad & = \bigcup_{c\in cs} c[f \mapsto c(f){+}{+}(\{c(v)\} \to X[expr]_c)](x). \qquad (12)\end{aligned}$$

As with the strong update case, we consider (12) splitting into the cases $f = x$ and $f \neq x$:

$$(12) = \bigcup_{f=w \wedge c \in cs} c(f){+}{+}(\{c(v)\} \rightarrow X[expr]_c) \cup \bigcup_{c \in cs \wedge f \neq w} c(x)$$

The case for $f \neq x$ is analogous to the strong update proof. Let us focus on the case when $f = x$. Since we know by Def. 5 that $\forall x \in JVar \uplus JField \,\bullet\, \alpha(cs)(x) \subseteq a(x)$, it holds that $\forall c \in cs$:

1. $c(f) \subseteq a(f)$.
2. $\{c(v)\} \rightarrow X[expr]_c \subseteq a(v) \rightarrow \hat{X}[expr]_a$. This is the case because
   - $\{c(v)\} \subseteq a(v)$ and
   - $X[expr]_c \subseteq \hat{X}[expr]_a$ by Lemma 1.

Thus, we derive by monotonicity of $\cup$,

$$c(f) \cup (\{c(v)\} \rightarrow X[expr]_c) \subseteq a(f) \cup (a(v) \rightarrow \hat{X}[expr]_a).$$

Therefore, since $R{+}{+}S \subseteq R \cup S$,

$$\begin{aligned} c(f){+}{+}(\{c(v)\} \rightarrow X[expr]_c) &\subseteq c(f) \cup (\{c(v)\} \rightarrow X[expr]_c) \\ &\subseteq a(f) \cup (a(v) \rightarrow \hat{X}[expr]_a). \end{aligned}$$

Finally,

$$\begin{aligned} &\bigcup_{f=w \wedge c \in cs} c(f){+}{+}(\{c(v)\} \rightarrow X[expr]_c) \\ &\qquad \subseteq a(f) \cup (a(v) \rightarrow \hat{X}[expr]_a) \\ &\qquad = a[f \mapsto a(f) \cup (a(v) \rightarrow \hat{X}[expr]_a)](f) \\ &\qquad = \hat{\mathcal{F}}(v.f := expr, a)(f). \end{aligned} \tag{13}$$

By (11), (13) and the case when $f \neq x$, we can conclude

$$\forall x \in JVar \uplus JField \,\bullet\, \alpha(\mathcal{F}(v.f := expr, cs))(x) \subseteq \hat{\mathcal{F}}(v.f := expr, a)(x).$$

Therefore, by Def. 7, we can conclude for the weak update case

$$\alpha(\mathcal{F}(v.f := expr, cs)) \subseteq \hat{\mathcal{F}}(v.f := expr, a).$$

Therefore,

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

■

The following corollary shows that the abstraction computed using the dataflow analysis is a safe approximation of the collecting semantics.

Corollary 1 *The dataflow equations for the value propagation analysis presented in Def. 9 are a safe abstraction of the equations for the collecting semantics defined in Def. 3. That is, for each node $n \in CFG(P)$, $\alpha(LFP(in(n))) \subseteq LFP(\hat{in}(n))$ and $\alpha(LFP(out(n))) \subseteq LFP(\hat{out}(n))$.*

Following the approach presented in [24] we can define[2]:

$$f(in, out)(n) = (\bigcup_{p\ in\ pred(n)} out(p), \mathcal{F}(in(n))),$$

$$\hat{f}(\hat{in}, \hat{out})(n) = (\bigsqcup_{p\ in\ pred(n)} \hat{out}(p), \hat{\mathcal{F}}(\hat{in}(n))).$$

Note that both f and $\hat{f}$ are monotone functions build from $\mathcal{F}$ and $\hat{\mathcal{F}}$. Then, from Theorem 1 and [9, 24] we know that $\alpha(LFP(f)) \subseteq LFP(\hat{f})$

■

Termination: Due to the monotonicity of dataflow equations and the fact that a finite *Atom* set leads to a finite and complete lattice (both the concrete and abstract domains are finite), using the Kleene's fixpoint theorem it is possible to compute a least fixpoint in a finite number of steps.

5 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 *tight* 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 modeling 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 invocations 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 [15]. Once the single static assignment (SSA) transformation is applied, DynAlloy variables and Alloy relations match. Therefore, *performing our value-propagation analysis on this final DynAlloy representation, yields an over approximation of all possible values each Alloy relation may have.*

[2] For the sake of simplicity and presentation we avoid showing the case for *entry* in the definitions of f and $\hat{f}$.

By default, Alloy associates a conservative upper bound (i.e., a set of Atoms for unary relations, a Cartesian product for binary relations, and so on), denoted U_x, for each relation x that was not explicitly bounded. This upper bound represents all possible values a variable or field may have according to its type and the scope of the analysis.

If an upper bound is found in its repository, TACO instruments the Alloy representation by including the stored upper bound, refining the initial state. We denote by U_x^{TACO} the final set of upper bounds obtained by this process. Note that $U_x^{TACO} \subseteq U_x$.

5.1 Initializing variables for Dataflow analysis

It is worth mentioning that, as the upper bounds stored in TACO's repository can be seen as an over approximation of the values of the initial Java memory heap, we can use them as a refined entry abstraction for our dataflow analysis.

We initialize each variable p representing the initial heap for the method under analysis with a TACO upper bound, if one is available, or use the default bound otherwise (i.e., $U_p^{TACO} \cap U_p$). For any other DynAlloy variable x, representing local variables, no tuple is associated (i.e., $a(x) = \emptyset$).

5.2 Using the dataflow analysis output

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 represents the possible values it may take. Thus, this 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.

Recall that Corollary 1 states that the propagation analysis is a safe approximation of the collecting semantics of the original program. We fed the analysis with the TACO upper bounds U_x^{TACO} which also represents a safe over approximation of the initial state. Therefore, applying the analysis with these bounds we obtain a safe over approximation of how these initial values are propagated through the control flow graph.

Since we applied the analysis over the SSA version of the DynAlloy program, we know that each variable is assigned only once. Thus, it is enough to consider the output of the analysis at the exit node in the CFG since it will contain the values for all the variables. For cases where x's abstract value maps to an empty set, it means that no value was actually propagated into that variable. However, a special measure has to be taken in order to enforce Alloy's relational constraints since at model creation it needs to assign a value to each variable . Therefore, for those cases we assign that variable a default value according to its type.

Definition 10 *Let $a_{exit} = \hat{in}(exit)$ be the computed abstract value for the exit node of the CFG.*

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

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.

Let U_x^{TACO} be the upper bound supplied by TACO to KodKod when no dataflow analysis is performed. Let U_x^{DF} be the bounds computed by the dataflow analysis. The following theorem ensures that using the latter bounds is safe (i.e., it does not miss failures).

Theorem 2 *Let θ be the Alloy formula output by the* DynAlloyToAlloy *translator. Given an Alloy instance I such that*

$$M[\theta \wedge \bigwedge_x (x \subseteq U_x^{TACO})]_I = \textbf{\textit{true}},$$

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

Proof: Let I be an Alloy instance satisfying the hypothesis. Recall that M takes a formula (in this case θ), a valuation (in this case I) and computes its truth value. Formula θ codifies the verification condition of the program and represents the possible variable assignments at every program point. The Alloy instance I is a particular valuation of those variables constrained by U_x^{TACO} that makes M to evaluate to true, and represents one of the possible program traces.
We identify two cases:

1. Variable $x \in JVar \uplus JField$ is a parameter for the method under analysis, the receiver object *this*, or a local variable that is declared (and perhaps also assigned) in the trace determined by instance I.
2. Variable $x \in JVar$ is a local variable that is not declared (and therefore not assigned) in the trace determined by instance I.

For example, for the method `m` in Fig. 7, if we consider trace 1,2,4,5,6, variables *this*, p_1, p_2 and x_1 are comprised in the first case. Variable y_1, on the other hand, is characterized by the second case.

Recall that the variables in the Alloy representation match the variables in the SSA version of the DynAlloy program. As mentioned before, due to Corollary 1 and since we fed this analysis with a safe over approximation of the initial state (i.e., U_x^{TACO}) we know that U_x^{DF} is a safe over approximation of all possible variable assignments for all variables. Therefore, in the first case we may safely define $I'(x)$ as $I(x)$. Notice that since the first case assigns values to all the variables that determine the execution trace, and the values agree with the ones assigned by instance I, the theorem holds.

```
m(A p1, B p2) {
1:  A x1  = p1;
2:  if(x.f>0)
3:     B y1 = p2;
4:  else
5:     assert false;
7:   return
}
```

Figure 7: The variable $y1$ is not assigned in the path 1,2,4,5,6.

Still, the definition for instance I is not complete, since it must assign values to all variables. The remaining variables, i.e., those characterized in the second case, do not have any impact on the executed trace, and their value can be assigned in any way that preserves their typing. We then define

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

It then clearly follows that

$$M[\theta \wedge \bigwedge_x (x \subseteq U_x^{TACO})]_I = \mathbf{true} \implies M[\theta \wedge \bigwedge_x (x \subseteq U_x^{DF)})]_{I'} = \mathbf{true}.$$

■

5.3 Loop Optimization

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

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

Although semantically correct, this representation of the while construct permits many permutations. For instance: the program trace $B?;P;\mathit{skip}$ is equivalent to $\mathit{skip};B?;P$. 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_k is recursively defined by the equations

$$\begin{aligned} T_0(B,P) &= \mathit{skip}, \\ T_n(B,P) &= (((B?;P);T_{n-1}(B,P)) + \mathit{skip}). \end{aligned}$$

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	0.14	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.findMax.s20	25.82	25.28	0.14	1.02	1412	5.95%
AvlTree.find.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.remove.s12	7251.39	14322.00	0.44	0.51	2396	13.28%
BSTree.insert.s09	18344.38	1214.49	1.79	15.10	13369	5.42%

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

6 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?

In order to answer these questions we implemented TacoFlow. TacoFlow[3] is an extension of TACO that implements the approach described in §5. 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 benchmark presented in [17] and compare the analysis times of TACO and TacoFlow. 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. [32]; **TreeSet**: The implementation of

[3] TacoFlow and the benchmark are available at `http://www.dc.uba.ar/tacoflow`.

class `TreeSet` from package `java.util`, based on red-black trees; **AVL-Tree**: 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 [32]; 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 [17].

Since we are interested in measuring the worst case scenario for bounded verification (i.e., search space exhaustion), we focused mostly on the analysis of correct implementations. We include the case of BHeap.extMin to show that the analysis does not miss bugs.

Both TACO and TacoFlow were fed with an initial set of upper bounds. These upper bounds were computed using a cluster of computers as reported in our previous work [17]. TacoFlow uses these bounds 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. The SAT-Solver `minisat` [14] version 2.20 was used for all the analysis tasks.

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 TacoFlow, the cost of the dataflow analysis in TacoFlow and its speed-up (the ratio TACO/TacoFlow). The last two columns show the number of propositional variables of the SAT-formula produced by TACO and the percentage of reduction introduced by TacoFlow.

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.

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 TacoFlow. TacoFlow 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 di-

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.findMax	0.97
AvlTree.find	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.remove	0.60
BSTree.insert	18.26

Table 2: TACO$^+$ speed-up

Method	TACO$^+$vs TacoFlow
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.findMax	1.06
AvlTree.find	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.remove	0.84
BSTree.insert	1.33

Table 3: TacoFlow speed-up

rectly related to the gain proportion, a further investigation of this matter is necessary.

TacoFlow 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 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. 8). Even tough this result was not initially expected, it is actually a consequence of the introduction of the dataflow analysis in TacoFlow which needs a better encoding of loop unrolls to mitigate precision loss.

We now focus on Table 3. For every method we took the maximum scope that TACO$^+$ could analyze and ran TacoFlow 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.

Nevertheless, tighter upper bounds are not only used by KodKod to remove propositional variables. KodKod also uses the provided lower and upper bounds to compute general purpose symmetry breaking predicates. As we have already pointed out, these symmetry restrictions reduce many (but often not all) isomorphic valuations. This led to a third research question:

RQ3: Is the introduction of our tighter upper bounds degrading the symmetry breaking predicate that KodKod produces?

Up to this point, we fed upper bounds directly to the KodKod engine. KodKod creates its symmetry breaking predicate using the provided lower and upper bounds. In this setting, tighter upper bounds could lead not only to removal of propositional variables. They may also lead to distinguishing classes that could be seen as isomorphic with coarser bounds. We could avoid this effect if TACO directly injects the tighter upper bounds at the SAT level without affecting KodKod. In other words, we would like to modify the SAT problem instead of the KodKod input.

In order to answer the research question, we modified our tool to leave KodKod's Alloy input model unchanged and, using the calculated bounds, produce a new SAT-formula to be fed directly to the SAT-solver. For this, we built a function that maps the tuples of every Alloy relation to the propositional variable that models whether that tuple is contained in the relation or not. For every tuple which is not contained in the upper bounds of the relation, we add a clause to the resulting SAT-formula stating that the value of the corresponding propositional variable, obtained through the mapping, is false. The resulting SAT-formula is sent directly to the SAT-solver. In order to effectively propagate the constant values, we simplify the SAT-formula by applying the `precosat` solver [4] in simplification only mode. We avoid unfairness in our comparison by applying the same procedure on the CNF formula produced by KodKod, and then measuring the SAT-solving time for both SAT-problems.

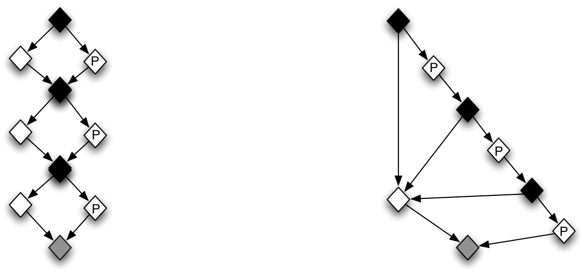
No footnote, referencia. Asi no le estas reconociendo el laburo a la genrte de precast con una cita

In Table 4 we show a comparison between the two approaches. TACO-KK stands for the version of TacoFlow that feeds upper bounds at the KodKod level, and TACO-SAT stands for the new version of TacoFlow that removes propositional variables by directly handling the SAT problem. In both columns we aggregate the computation cost of simplifying the SAT problem.

Although the speed-up on average was slightly positive, less than half of the 21 cases showed performance gains. There seems to be no correlation between the performance loss and the effect of the symmetry breaking predicate built by KodKod.

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 [17].

[4] `http://fmv.jku.at/precosat/`



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

Figure 8: Loop unroll encodings in TACO and TacoFlow.

Method	Analysis Times (secs.) TACO-KK	TACO-SAT	Speed-up
SList.contains.s20	2.34	1.49	1.57
SList.insert.s20	6.83	5.46	1.25
SList.remove.s20	5.98	7.53	0.79
AList.contains.s20	4.04	3.27	1.24
AList.insert.s20	4.9	5.33	0.92
AList.remove.s20	4.82	5.34	0.90
CList.contains.s20	42.94	50.24	0.85
CList.insert.s20	23.9	21.69	1.10
CList.remove.s20	223.58	267.85	0.83
AvlTree.findMax.s20	10.74	9.53	1.13
AvlTree.find.s20	41.05	45.12	0.91
AvlTree.insert.s17	4683	4649	1.01
BinHeap.findMin.s20	3.2	2.56	1.25
BinHeap.decK.s18	28.97	32.59	0.89
BinHeap.insert.s17	141.67	101.82	1.39
BinHeap.extMin.s20	36	144,68	0.25
TreeSet.find.s20	559.04	432.59	1.29
TreeSet.insert.s13	373.56	426.26	0.88
BSTree.contains.s13	4300.1	2312.5	1.86
BSTree.remove.s12	8076.1	10646	0.76
BSTree.insert.s09	244.42	286	0.85
Average			1.04

Table 4: Analysis times (in seconds) for TACO-KK and TACO-SAT

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 [5,12,22,28,32]. In addition, the algorithms found in the case studies are commonplace. They recurrently appear in many applications [29] 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 [12].

Finally, TacoFlow 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.

7 Related work

There is plenty of work aiming at improving SAT-based program verification and it is still a very active research topic. The most remarkable examples are the approaches implemented in CBMC [7], SAT-ABS [8], F-Soft [19], Saturn [33], Avinux [26] for the analysis of C code, and TACO [17], Miniatur [13] and JForge [12] for the analysis of Java code. For a comprehensive discussion of these tools please refer to [16].

Here we will focus the discussion only on related work concerning the use of dataflow analysis to alleviate the task of the SAT-solver. We start by discussing some tools that perform dataflow analyses on C programs [4, 19]. Then, we comment about approaches that apply it for Java programs [12, 27, 30]. These papers use an approach similar to TACO in the sense they use Alloy as back-end but use dataflow analyses for different optimization purposes.

F-Soft [19] is a tool for verifying C source code. It can check C programs for runtime errors (e.g, pointer access violations, buffer overflows) or user provided contracts. Like TacoFlow it uses bounded model checking to prove properties. In other to reduce the work given to the model checker, F-SOFT tries to prove some properties beforehand by applying several dataflow analyses. These analysis include intervals, octagons, symbolic ranges and polyhedra. Our analyses targets Java programs which mades intensive use of heap data structures rather than numerical analysis. One aspect that is similar is that some of the F-Soft's analyses (e.g. range analysis) aim at reducing the propositional representation of values. In essence, we also try to reduce this representation by computing tight bounds on the possible values a variable (or field) can refer to.

Scoot [4] is a tool for analyzing data races in System C programs using the model checker SAT-ABS [8]. Scoot uses dataflow analysis for over-approximating the dependency relation during the analysis of concurrency. TacoFlow currently targets single-thread Java programs and uses dataflow in order to reduce the amount of propositional variables required to model all possible heaps within the given scope.

Taghdiri et al. [30] proposed an analysis to infer syntactic method summaries for heap-manipulating Java code. The aim is to use those specifications to enable modular verification. Abstract interpretation is used in order to compute those summaries. This approach is orthogonal to ours since since it is meant to work at the inter procedural level it can be combined with ours which is designed to reduce the cost at the intraprocedural level.

JForge [12] is a Java front-end of Forge, a program analysis framework that uses a bounded model checker (Alloy) to verify rich interface specifications written in JFSL. Forge uses a simple dataflow analysis to determine the potential target methods of an invocation and to find and eliminate logically infeasible branches. TACO also targets Java program but used a dataflow analysis to propagate a set of precomputed upper bounds and in that way reduce the number of propositional variables in the final encoding. In that, sense both tools may benefit from each other techniques as they are mutually independent.

In [27] the authors propose a technique for optimizing an incremental scope-based model checking using a divide-and-solve approach as a mean to improve the scalability of bounded model checking approaches. To do that, they rely on a dataflow analysis (variable-definitions) to split the SAT-problem into several simpler sub-problems.

8 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. We also analyzed the effect of removing variables at the Alloy and SAT levels. Our findings were inconclusive over this topic. We were unable to establish a direct link between the symmetry breaking predicate built by KodKod and the performance degradations we experienced by removing propositional variables.

Nevertheless, the whole approach still led to an important increase of performance for several case studies. We still need to perform more experiments in order to assess with confidence whereas the approach is capable of increasing the scope of analysis beyond the current state-of-the-art.

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.

One of the main difficulties in bounded verification is figuring out which are the right scopes to choose for each kind of objects. On the one hand, if the scopes are too small the analysis ability to find counter-examples might be drastically reduced. On the other hand, bigger scopes may exhaust computing resources. We believe inference techniques relying on dataflow analysis (e.g., points-to analysis) may be useful to assist developer in finding the right scope for each different kind of object.

Acknowlegments

We thank the anonymous reviewers for their insightful comments and suggestions that help us improving the paper. This work is partially supported by PICT-PAE 2278, PICT-1774, PICT-2351, UBACYT W0813, UBACYT

20020110200075, CONICET PIP955, PIP 11220110100596CO, LIA INFINIS, EA ANCOME and MEALS 295261.

References

1. B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM, 1988.
2. Jason Belt, Robby, and Xianghua Deng. Sireum/topi ldp: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 355–364. ACM, 2009.
3. A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, 2009.
4. N. Blanc, D. Kroening, and N. Sharygina. Scoot: A tool for the analysis of systemc models. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–470, 2008.
5. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133. ACM, 2002.
6. Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: advanced specification and verification with jml and esc/java2. In *Proceedings of the 4th international conference on Formal Methods for Components and Objects*, pages 342–363. Springer-Verlag, 2006.
7. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.
8. Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.
9. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.
10. Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2–3):103 – 179, 1992.
11. James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.
12. Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 130–145. Springer-Verlag, 2008.
13. Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a sat solver. In *Proceedings of the the 6th joint meeting of the European software*

engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC-FSE '07, pages 195–204, New York, NY, USA, 2007. ACM.

14. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
15. Marcelo Frias, Juan Pablo Galeotti, Carlos López Pombo, and Nazareno Aguirre. Dynalloy: upgrading alloy with actions. In *Proceedings of the 27th international conference on Software engineering*, pages 442–451. ACM, 2005.
16. Juan Pablo Galeotti. *Software Verification Using Alloy.* PhD thesis, University of Buenos Aires, 2010.
17. Juan Pablo Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 25–36. ACM, 2010.
18. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* The MIT Press, 2000.
19. Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *CAV'05*, pages 301–306, 2005.
20. Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.
21. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis (Revised Edition).* The MIT Press, 2012.
22. Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 14–25. ACM, 2000.
23. Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.
24. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis.* Springer-Verlag New York Inc, 1999.
25. Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.
26. H. Post, C. Sinz, and W. Küchlin. Towards automatic software model checking of thousands of linux modules – a case study with avinux. *Software Testing, Verification and Reliability*, 19(2):155–172, 2009.
27. Danhua Shao, Divya Gopinath, Sarfraz Khurshid, and Dewayne E. Perry. Optimizing incremental scope-bounded checking with data-flow analysis. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 408–417. IEEE Computer Society, 2010.
28. Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. Testing container classes: random or systematic? In *Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software*, FASE'11/ETAPS'11, pages 262–277. Springer-Verlag, 2011.
29. Junaid Haroon Siddiqui and Sarfraz Khurshid. An empirical study of structural constraint solving techniques. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, pages 88–106. Springer-Verlag, 2009.

30. Mana Taghdiri, Robert Seater, and Daniel Jackson. Lightweight extraction of syntactic specifications. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 276–286. ACM, 2006.
31. Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, pages 632–647. Springer-Verlag, 2007.
32. Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for java containers using state matching. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48. ACM, 2006.
33. Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29:16–es, 2007.
34. Kuat Yessenov. A light-weight specification language for bounded program verification. Master's thesis, MIT, 2009.