Abstract Domains and Solvers for Sets Reasoning

Arlen Cox, Bor-Yuh Evan Chang[1], Huisong Li[2], Xavier Rival[2]

University of Colorado Boulder[1] Inria/CNRS/ENS Paris/PSL*[2]

Abstract. When constructing complex program analyses, it is often useful to reason about not just individual values, but collections of values. Symbolic set abstractions provide building blocks that can be used to partition elements, relate partitions to other partitions, and determine the provenance of multiple values, all without knowing any concrete values. To address the simultaneous challenges of scalability and precision, we formalize and implement an interface for symbolic set abstractions and construct multiple abstract domains relying on both specialized data structures and off-the-shelf theorem provers. We develop techniques for lifting existing domains to improve performance and precision. We evaluate these domains on real-world data structure analysis problems.

1 Introduction

The verification of program properties that involve data structures is a challenging problem [2, 9, 10, 12, 13, 16, 19]. One key reason for this is that if a data structure is unbounded, there is a potentially unbounded number of constraints on its elements. Since these constraints often affect important properties such as memory safety [16], functional correctness [19], or basic program behavior [9], it is vital to develop techniques for efficiently reasoning about relationships between unbounded numbers of elements.

This paper focuses on the use of set constraints to reason about unbounded collections of elements. Set constraints can be used to dynamically partition data structures, correlate collections of elements with one another, or determine analysis case splits. They are useful for representing data and pointer relationships in structures such as maps, graphs, lists, sets, and arrays. They can be combined with other techniques such as separation logic [9, 16] and numerical analyses [8] to enhance those analyses.

For example, consider the program in Figure 1 that copies one map on top of another. Within the loop, there is a complex relationship between the sets of keys of `src` and `dst`. At the specified point, the keys of `src` can be partitioned into three parts. The keys already visited X_v by the loop, the element currently being visited $\{\mathtt{x}\}$ by the loop , and the keys not visited X_n by the loop. The keys of `dst` can be partitioned into those $\mathit{keys}(\mathtt{dst})_0$ originally in `dst` that have not been overwritten, and those X_v that have been overwritten or added from `src`. This set reasoning allows precise symbolic tracking of the provenance of map partitions.

This paper focuses on abstractions for states described by the logic for *symbolic sets*. The logic consists of a Boolean algebra over the set variables with singleton

```
def extend(dst, src):
  for x in src:
      (|∃X_v,X_n. keys(src) = X_v ⊎ {x} ⊎ X_n ∧ keys(dst) = (keys(dst)_0 \ X_v) ⊎ X_v|)
    dst[x] = src[x]
```

Fig. 1. Set constraints can relate portions of data structures

sets. We find that this subset is sufficiently large to be useful and we believe that it serves as a good starting point for extensions to the logic, such as reasoning about explicit set contents or more precise cardinality.

However, despite the fact that we are not reasoning about the values contained in sets or complex cardinalities, Boolean algebras, by themselves, are challenging for invariant generation. Naive approaches such as saturation and pattern matching rarely work without complex heuristics [10, 19]. It is unavoidable that the worst-case time for precise invariant generation will be exponential because of the Boolean algebra. However, it is desirable that invariant generation should be efficient in the common cases, and unlike systems that involve complex heuristics, lose precision only in understandable and predictable ways.

In this paper we aim to design scalable, precise, and predictable abstractions for symbolic sets by combining new abstract domains with performance/precision-enhancing functors that lift existing set abstractions to new set abstractions. Specifically, we make the following contributions:

- We define a common interface for symbolic set abstractions that is designed to meet the needs of static analyzers (Section 3).
- Using specialized data structures, we construct a battery of symbolic set abstract domains and performance-/precision-enhancing functors designed to target real-world data structure verification problems (Section 4).
- We adapt an off-the-shelf satisfiability-modulo-theories solver to the set abstraction interface (Section 5).
- We compare abstractions for symbolic sets, finding that, while specialized abstractions are preferable, binary decision diagrams lifted with dynamic packing is a good compromise in scalability, performance, and predictability (Section 6).

2 Overview

In this section, we present two static analyses that make use of set reasoning in order to compute high-level semantic properties of programs. These analyses rely on abstract interpretation [6] and on an abstraction of program states that describes data structures and their contents. An abstract domain defines a set of predicates that an analysis may use, as well as operators to over-approximate the effect of program behaviors on these predicates, and their implementation.

Inference of properties of open objects. Dynamic programming languages such as JavaScript feature *open objects* that support dynamic addition and deletion

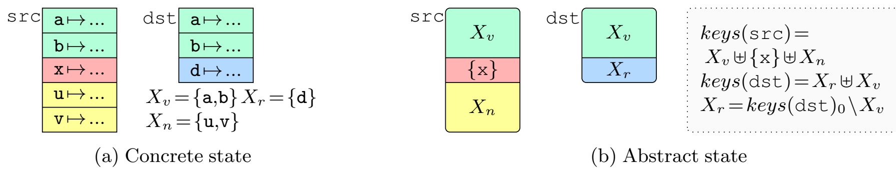
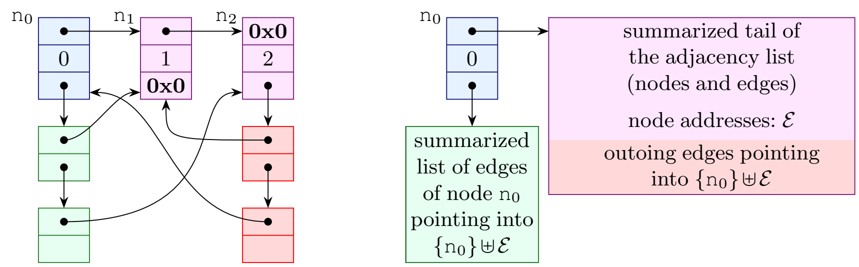


Fig. 2. Open objects and their abstraction

of attributes and iteration over them. The analysis presented in [9] verifies open-object-/map-manipulating programs such as the one in Figure 1, by inferring relations between the sets of attributes of distinct objects. Since objects may have an unbounded number of attributes, the analysis must abstract the attributes and their contents. Figure 2 represents a simplified state at the indicated point in Figure 1 after two iterations (thus two fields were copied). We focus on the set of attributes of each object and ignore their contents (which could be described using similar techniques). To precisely abstract the relations between the attributes of both objects (e.g. copied attributes are common to both objects), we partition the attributes into a series of attribute sets and express relations among these sets. The purpose of the set abstract domain is to represent such set relations. Figure 2b depicts such an abstract state, where X_n,X_r,X_v stand for sets of attributes, which are made explicit in Figure 2a, the concrete state.

Moreover, to infer these invariants, the analysis needs to reason about both object structures and attribute sets. Initially it assumes no set relations, and the fields of each object should be associated to an arbitrary set of attributes. When the analysis enters the body of the loop, it needs to *single out* attribute `x`, i.e. to replace set X_v by $X_v \uplus \{\mathtt{x}\}$,which produces the equalities of Figure 2. When it exits the loop, the analysis should *generalize* both the object and set constraints abstractions, which requires *eliminating* the singleton $\{\mathtt{x}\}$ from the equations (it is visible only in the loop body) and synthesizing a new, more general collection of constraints. To allow these steps, the set abstraction should provide basic operations over set predicates, including (1) the addition of a set constraint, (2) the proving of a set constraint, (3) the removal of a set variable, and (4) the generalization of two set abstract states.

Shape analysis in presence of unstructured sharing. The shape analysis for data-structures with unbounded sharing presented in [16] relies on separation logic [20] to describe memory states and on inductive definitions to summarize unbounded structures such as lists. Unstructured sharing is very challenging as it cannot be described using conventional inductive definitions. Figure 3a displays the representation of a three nodes graph using an adjacency list data-structure. To summarize such a structure using inductive predicates in separation logic, [16] augments the list inductive predicates with set information, which express where edges may point to. Figure 3b shows this representation in a form where the first node is kept materialized. It asserts that the edges of that node and other nodes point to the address of a valid node, namely an element of $\{\mathsf{n}_0\} \uplus \mathcal{E}$. The analysis of [16] introduces a summary predicate $\mathbf{graph}(\mathsf{n}_0,\mathcal{N})$ where n_0 is the address of the first node and $\mathcal{N}$ the set of all node ad-



(a) A concrete memory state

(b) Abstract memory state

Fig. 3. Summarization of an adjacency list-based graph representation

dresses. This predicate is defined by induction over the "backbone" of the structure, and fully takes into account the property that all edges point to a valid node address in $\mathcal{N}$. Henceforth, abstract states comprise both a *memory* part (which consists of a formula in separation logic with inductive predicates) and a *set abstraction*.

To compute such summaries, the analysis needs to perform similar operations as the analysis for open objects, in order to add set constraints to the set abstract state, prove set constraints, remove set variables, and generalize abstract states.

3 Logic and Set Abstraction

We now define the elements and operators of a set abstract domain that meets the needs of all the analyses shown in Section 2.

Concrete states. In this paper, we use symbols W, X, Y, and Z as set variables and let $\mathbb{X}_{\mathbf{s}}$ represent the set of all such variables. We are interested in purely symbolic set relations, and do not make any assumption on the type of the set elements (in practice these are pointers or scalars). We let $\mathbb{V}$ denote the set of all these elements. A concrete state is a function $\sigma : \mathbb{X}_{\mathbf{s}} \to \mathcal{P}(\mathbb{V})$. We write $\mathbb{S}$ for the set of such elements.

Symbolic sets. Before we set up the signature of abstract domains, we fix a language of set predicates, that will be used as a basis for abstract elements, and for the communication with the set abstract domain.

Definition 1 (Symbolic Sets). Symbolic sets *are defined by the grammar:*

$$L(\in \mathbb{C}) ::= L \wedge L \mid E \subseteq E \mid |X| = 1 \mid \top \mid \bot \qquad E ::= \emptyset \mid X \mid E^{\mathrm{c}} \mid E \cup E \mid E \uplus E$$

The meaning of these constraints is straightforward, but we give a formal definition in Figure 4 for clarity. A model of a set expression E is a concrete state σ and a set of concrete values c. A model of a logical expression L is a concrete state σ. The concretization is $\gamma(L) = \{ \sigma \mid \sigma \models L \}$ and we use $(\!|L|\!)$ for abstract states with the same concretization. We shall also use the following derived logical forms for simplicity:

$$E_1 \cap E_2 \stackrel{\mathrm{def}}{=} ({E_1}^{\mathrm{c}} \cup {E_2}^{\mathrm{c}})^{\mathrm{c}} \qquad E_1 = E_2 \stackrel{\mathrm{def}}{=} E_1 \subseteq E_2 \wedge E_2 \subseteq E_1 \qquad E_1 \setminus E_2 \stackrel{\mathrm{def}}{=} E_1 \cap {E_2}^{\mathrm{c}}$$

$$\sigma,c \models \emptyset \text{ iff } c=\emptyset \qquad \sigma,c \models X \text{ iff } c=\sigma(X) \qquad \sigma,c \models E^{c} \text{ iff } \sigma,c' \models E \text{ and } \forall v \in \mathbb{V}.\, v \in c \Leftrightarrow v \notin c'$$
$$\sigma,c \models E_1 \cup E_2 \text{ iff } \sigma,c_1 \models E_1 \text{ and } \sigma,c_2 \models E_2 \text{ and } \forall v \in \mathbb{V}.\, v \in c \Leftrightarrow v \in c_1 \vee v \in c_2$$
$$\sigma,c \models E_1 \uplus E_2 \text{ iff } \sigma,c_1 \models E_1 \text{ and } \sigma,c_2 \models E_2 \text{ and } \forall v \in \mathbb{V}.\, v \in c \Leftrightarrow v \in c_1 \vee v \in c_2 \text{ and } c_1 \cap c_2 = \emptyset$$
$$\sigma \models L_1 \wedge L_2 \text{ iff } \sigma \models L_1 \text{ and } \sigma \models L_2 \qquad \sigma \models |E| = 1 \text{ iff } \sigma,c \models E \text{ and } \exists v \in \mathbb{V}.\, c = \{v\}$$
$$\sigma \models E_1 \subseteq E_2 \text{ iff } \sigma,c_1 \models E_1 \text{ and } \sigma,c_2 \models E_2 \text{ and } \forall v \in \mathbb{V}.\, v \in c_1 \rightarrow v \in c_2 \qquad \sigma \models \top \qquad \sigma \not\models \bot$$

Fig. 4. Symbolic set constraint language

Set abstraction. A *set abstract domain* is defined by a set of *abstract elements* $\mathbb{D}^\sharp$ which describe the family of logical properties it can express and a concretization function $\gamma : \mathbb{D}^\sharp \rightarrow \mathcal{P}(\mathbb{S})$ that maps each element of $\mathbb{D}^\sharp$ into the set of concrete states that satisfy it. Abstract elements are characterized by (1) the symbolic sets they describe and (2) their machine representation. The latter is usually very different from the formulas, and will be discussed in Section 4.

Example 1 ((Non-)Emptiness set domain). A very basic example of such a domain is the *(non-)emptiness* domain that comprises the following elements:
- $\bot$, which denotes the unsatisfiable abstract constraint (i.e., $\gamma(\bot) = \emptyset$));
- the functions from $\mathbb{X}_\mathbf{s}$ into $\{[=\emptyset], [\neq\emptyset], \top\}$, which map each set variable into its emptiness value.

For instance, $\{X \mapsto \top; Y \mapsto [=\emptyset]\}$ stands for $(\!|Y \subseteq \emptyset|\!)$ and concretizes into $\gamma(Y \subseteq \emptyset)$.

Operations over Set Abstractions. We now formalize the main operations and logical elements needed so that we can use a set abstract $\mathbb{D}^\sharp$ domain for either of the static analyses shown in Section 2.

– *Basic logical elements.* Static analyses typically start with an unconstrained state. This is indicated by a $\top_{\mathbb{D}^\sharp} \in \mathbb{D}^\sharp$ element with full concretization, i.e., $\gamma(\top_{\mathbb{D}^\sharp}) = \mathbb{S}$. Similarly, the abstract element $\bot_{\mathbb{D}^\sharp} \in \mathbb{D}^\sharp$ should describe the unsatisfiable abstract constraint (i.e., $\gamma(\bot_{\mathbb{D}^\sharp}) = \emptyset$). In Example 1, $\bot_{\mathbb{D}^\sharp}$ is $\bot$ and $\top_{\mathbb{D}^\sharp}$ is $\lambda(x \in \mathbb{X}_\mathbf{s}) \cdot \top$. Moreover, a static analysis often has to determine if an abstract state describes unsatisfiable constraints. Thus, $\mathbb{D}^\sharp$ should provide an operator $\mathbf{isbot}_{\mathbb{D}^\sharp} : \mathbb{D}^\sharp \rightarrow \{\mathbf{true}, \mathbf{false}\}$ such that $\mathbf{isbot}_{\mathbb{D}^\sharp}(\sigma^\sharp) = \mathbf{true} \Longrightarrow \gamma(\sigma^\sharp) = \emptyset$.

– *Forgetting a set variable.* Static analysis tools drop set variables that become redundant. In the open object example of Section 2, this occurs when the singleton symbol is eliminated at the end of the loop. To do this, we require the set abstract domain $\mathbb{D}^\sharp$ to provide an operator $\mathbf{forget}_{\mathbb{D}^\sharp} : \mathbb{D}^\sharp \times \mathbb{X}_\mathbf{s} \rightarrow \mathbb{D}^\sharp$ that discards a symbol from the abstract state.

– *Assuming set constraints.* As noted in Section 2, an important set reasoning step *restricts an abstract state with set constraints*, thus set domain $\mathbb{D}^\sharp$ should provide an operator $\mathbf{assume}_{\mathbb{D}^\sharp} : \mathbb{D}^\sharp \times \mathbb{C} \rightarrow \mathbb{D}^\sharp$, which conservatively represents a constraint into an abstract state, i.e. ensures that, for all $\sigma^\sharp, L$, $\gamma(\sigma^\sharp) \cap \gamma(L) \subseteq \gamma(\mathbf{assume}_{\mathbb{D}^\sharp}(\sigma^\sharp, L))$. Note that this operator also makes use of the symbolic set language of Definition 1 in order to describe constraints communicated to the domain.

– *Verifying set constraints.* Similarly, set reasoning should allow *verifying set constraints*, thus the set domain $\mathbb{D}^\sharp$ should provide an operator $\mathbf{prove}_{\mathbb{D}^\sharp} : \mathbb{D}^\sharp \times \mathbb{C} \rightarrow \{\mathbf{true},\mathbf{false}\}$, which conservatively attempts to verify that a symbolic set constraint holds under some abstract states, i.e. ensures that, for all $\sigma^\sharp, L$, $\mathbf{prove}_{\mathbb{D}^\sharp}(\sigma^\sharp, L) = \mathbf{true}$ implies that $\gamma(\sigma^\sharp) \subseteq \gamma(L)$.

– *Generalizing set abstractions.* The analysis of loops is commonly based on the computation of abstract post-fixpoints [6], thus $\mathbb{D}^\sharp$ should provide sound over-approximation of the union of sets concrete states. In the logical point of view, this amounts to computing a common weakening for two abstract constraints. This is performed by an operator $\mathbf{join}_{\mathbb{D}^\sharp} : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$ such that, for all $\sigma_0^\sharp, \sigma_1^\sharp$, $\gamma(\sigma_0^\sharp) \cup \gamma(\sigma_1^\sharp) \subseteq \gamma(\mathbf{join}_{\mathbb{D}^\sharp}(\sigma_0^\sharp, \sigma_1^\sharp))$. Widening operator $\mathbf{widen}_{\mathbb{D}^\sharp}$ should satisfy the same property and ensure termination of any sequence of abstract iterates.

– *Deciding entailment over set abstractions.* Finally, the operator $\mathbf{is\_le}_{\mathbb{D}^\sharp} : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \rightarrow \{\mathbf{true}, \mathbf{false}\}$ conservatively decides implication among abstract set constraints (by ensuring that $\mathbf{is\_le}_{\mathbb{D}^\sharp}(\sigma_0^\sharp, \sigma_1^\sharp) = \mathbf{true} \Longrightarrow \gamma(\sigma_0^\sharp) \subseteq \gamma(\sigma_1^\sharp)$), and allows verifying the convergence of abstract iterates.

4 Constructed Set Abstractions

An abstract domain is defined by a class of set constraints, their machine representation, and the abstract operations following the signatures given in Section 3. In this section, we introduce three basic set abstract domains (respectively based on linear constraints, QUIC graphs, and BDDs) and two set abstract domain functors, that lift a set domain into another, more expressive or efficient one.

4.1 Linear Set Constraints

Abstract elements and their concretization. Our first set abstract domain relies on *linear* set equality constraints, of the form $(\!| X = \{y_0, \ldots, y_k\} \uplus Z_0 \uplus \ldots \uplus Z_l |\!)$. The advantage of such constraints is to provide a rather straightforward normalization of the representation of constraints. Note they also include emptiness constraints. Our implementation of abstract domain $\mathbb{D}_\ell^\sharp$ describes three kinds of constraints:

- acyclic *linear* constraints of the form $(\!| X = Y_0 \uplus \ldots \uplus Y_k \uplus Z_0 \uplus \ldots \uplus Z_l |\!)$, where $Y_0, \ldots, Y_k$ are singletons (containing $y_0, \ldots, y_k$ respectively). In the implementation, each variable may appear at most *once* as the left-hand side of such a constraint, to enable normalization;
- inclusion constraints of the form $(\!| Y \subseteq X |\!)$;
- equality constraints of the form $(\!| Y = X |\!)$.

Thus, an element of $\mathbb{D}_\ell^\sharp$ is either $\perp$ or a conjunction of such constraints. The associated concretization $\gamma_\ell : \mathbb{D}_\ell^\sharp \rightarrow \mathcal{P}(\mathbb{S})$ is of the same form as that of the symbolic sets language of Definition 1 (thus, we do not formalize it in full details). The machine representation utilizes persistent dictionaries, that stand for functions over a finite domain. This reduces basic queries for facts (such as, “does abstract state $\sigma^\sharp$ entail that $X \subseteq Y \uplus Z$?”) to dictionary searches.

Abstract operators. The core algorithm of $\mathbb{D}_\ell^\sharp$ normalizes abstract values by expanding nested linear constraints. For instance, $(\!|X_0 = X_1 \uplus X_2 \wedge X_1 = X_3 \uplus X_4|\!)$ is rewritten into $(\!|X_0 = X_2 \uplus X_3 \uplus X_4 \wedge X_1 = X_3 \uplus X_4|\!)$ at the machine representation level. This process terminates as constraints represented in $\mathbb{D}_\ell^\sharp$ do not contain cycles. It is performed incrementally by all abstract operations.

Abstract operations $\mathbf{isbot}_{\mathbb{D}^\sharp}$, $\mathbf{assume}_{\mathbb{D}^\sharp}$, $\mathbf{prove}_{\mathbb{D}^\sharp}$ are all made very fast by this normalization. Operation $\mathbf{forget}_{\mathbb{D}^\sharp}$ simply drops all constraints that involve a given set variable. Finally, $\mathbf{join}_{\mathbb{D}^\sharp}$ and $\mathbf{widen}_{\mathbb{D}^\sharp}$ need to *generalize* constraints.

Example 2. Let us assume that $\sigma_0^\sharp$ (resp., $\sigma_1^\sharp$) stands for the set of constraints $(\!|X_0 = X_1 \uplus X_2 \wedge X_3 = \emptyset|\!)$ (resp., $(\!|X_0 = X_1 \uplus X_2 \uplus X_3|\!)$). Then $\mathbf{join}_{\mathbb{D}^\sharp}(\sigma_0^\sharp, \sigma_1^\sharp)$ returns an element that represents the constraint $(\!|X_0 = X_1 \uplus X_2 \uplus X_3|\!)$.

MemCAD [16] relies on $\mathbb{D}_\ell^\sharp$ to represent set constraints since it mainly needs to express constraints over set partitions. On the other hand, $\mathbb{D}_\ell^\sharp$ is not adapted to the precise description of non disjoint unions.

4.2 QUIC graphs

A QUIC graph [10] is a directed hypergraph data structure used to represent relational set constraints. Each edge in the hypergraph corresponds to a subset constraint and each hypergraph is a conjunction of subset constraints where each constraint is of the form $(\!|X_1 \cap ... \cap X_n \subseteq Y_1 \cup ... \cup Y_m|\!)$. Each variable can also be constrained to be a singleton, with constraints such as $(\!||X| = 1|\!)$. The concretization $\gamma_q : \mathbb{D}_q^\sharp \to \mathcal{P}(\mathbb{S})$ is of the same form as that of the symbolic sets language of Definition 1.

QUIC graphs are designed for efficiently performing two operations: (1) $\mathbf{forget}_{\mathbb{D}^\sharp}$, which matches edges containing the symbol to be forgotten with each other to produce new edges without that symbol; and (2) content reasoning, which is not a design goal for symbolic sets. The $\mathbf{join}_{\mathbb{D}^\sharp}$ and $\mathbf{widen}_{\mathbb{D}^\sharp}$ operations are primarily based on saturation heuristics. They keep common conjunctions from both arguments. To aid this process, they use a form of saturation that produces new conjuncts based on pattern matches. A sufficiently large set of patterns must be provided to attain precision, but additional patterns increase the cost of joins.

Example 3 (QUIC graph join). Consider the following join operation:

$$\sigma_0^\sharp = (\!|W \subseteq X \wedge X \subseteq Z|\!) \qquad \sigma_1^\sharp = (\!|W \subseteq Y \wedge Y \subseteq Z|\!) \qquad \mathbf{join}_{\mathbb{D}^\sharp}(\sigma_0^\sharp, \sigma_1^\sharp)$$

There is an obvious result: $(\!|W \subseteq Z|\!)$. Whether or not QUIC graphs derive this result or $(\!|\top|\!)$ is determined by the pattern matches that are installed. If the pattern that takes $(\!|X \subseteq Y \wedge Y \subseteq Z|\!)$ and generates $(\!|X \subseteq Z|\!)$ is used, the pattern will be applied to both sides and then common conjuncts kept, getting the desired result. Without that pattern or a similar substitute, QUIC graphs derive $(\!|\top|\!)$.

4.3 BDD-based Set Constraints

Binary decision diagrams (BDDs) [21] are a canonical representation of Boolean algebraic functions. There are three basic syntactic elements of a BDD. The TRUE and FALSE elements represent the obvious constants, but $\mathrm{ITE}(X, B_t, B_e)$ is an if-then-else structure. If the variable X is **true**, the result of evaluating B_t is returned, otherwise the result of evaluating B_e is returned.

$$B ::= \textsc{True} \mid \textsc{False} \mid \mathrm{ITE}(X, B_t, B_e)$$

What makes BDDs canonical is that we only consider reduced, ordered BDDs, where it is assumed that there is a total order $\prec$ on the variables. An $\mathrm{ITE}(X, B_t, B_e)$ can only be constructed if $X \prec X'$ for all variables X' in B_t or B_e. Additionally, structural sharing is mandated, so the reuse of the same syntax is referentially identical to any other use of that syntax.

The encoding of constraints maps operators from their constraint form (as in Definition 1) to their Boolean algebraic form: $\cup \mapsto \vee$, $\cap \mapsto \wedge$, ${}^{c} \mapsto \neg$, $\subseteq \mapsto \rightarrow$, $= \mapsto \leftrightarrow$. All but singleton set constraints are directly and exactly represented by the BDD. Singleton constraints are not currently used by the BDD-based abstraction.

Domain operations are straightforward: $\mathbf{join}_{\mathbb{D}^\sharp}$ and $\mathbf{widen}_{\mathbb{D}^\sharp}$ are implemented with the $\vee$ operation, which is precise and does not need any rules or heuristics; $\mathbf{forget}_{\mathbb{D}^\sharp}$ takes advantage of reasonably efficient quantifier elimination provided by BDDs and uses existential quantifier elimination to drop variables. Queries such as $\mathbf{is\_le}_{\mathbb{D}^\sharp}$ are easily implemented using validity checking functionality provided by BDDs. Critically, because BDDs are a canonical form, many operations such as $\mathbf{forget}_{\mathbb{D}^\sharp}$ and $\mathbf{assume}_{\mathbb{D}^\sharp}$ become much more efficient, whereas the operation $\mathbf{isbot}_{\mathbb{D}^\sharp}$ becomes an $O(1)$ check.

Example 4 (BDD-based join). Consider the same inputs as Example 3. Encoding them to BDDs (and using some Boolean-algebraic notation as shorthand) yields the following results:

$$\begin{aligned}
\sigma_0^\sharp &= (\!|W \subseteq Y \wedge Y \subseteq X|\!) = \mathrm{ITE}(W, X \wedge Y, \mathrm{ITE}(X, \textsc{True}, \neg Y)) \\
\sigma_1^\sharp &= (\!|W \subseteq Z \wedge Z \subseteq X|\!) = \mathrm{ITE}(W, X \wedge Z, \mathrm{ITE}(X, \textsc{True}, \neg Z)) \\
\mathbf{join}_{\mathbb{D}^\sharp}(\sigma_0^\sharp, \sigma_1^\sharp) &= \mathrm{ITE}(W, X \wedge \mathrm{ITE}(Y, \textsc{True}, Z), \\
&\qquad \mathrm{ITE}(X, \textsc{True}, \mathrm{ITE}(Y, \neg Z, \textsc{True})))
\end{aligned}$$

The result of this join is equivalent to the set constraints $(\!|W \subseteq X|\!)$, $(\!|W \subseteq Y \cup Z|\!)$, and $(\!|Y \cap Z \subseteq X|\!)$, which includes not only the obvious result of $(\!|W \subseteq X|\!)$, but also other, possibly useful results. It is a precise join.

We implement the BDD abstraction on top of the CU decision diagrams package [22], which is high performance and offers the ability to extract prime implicants (as in [5]). The prime implicants of the negation of the Boolean function are easily converted to conjuncts of the form used by QUIC graphs.

4.4 The Equalities Domain Functor: Compact Equality Constraints

When analyzing real programs, in addition to complex set constraints, there are often many very simple equality constraints of the form $(\!|X = Y|\!)$. These can be a problem in several ways. For example, equalities are normalized and handled precisely in BDDs, but they can grow the size of the representation significantly. This results in significantly increased memory usage and decreased efficiency since many BDD operations rebuild the BDD. In QUIC graphs, equalities grow the size of the graph, and place significantly more load on the pattern matching system, potentially causing an explosion in the number of constraints. This is because QUIC graphs can represent each variant of an expression rewritten using all available equalities. In linear set abstractions, there are similar potential problems.

As a result, abstractions like QUIC graphs and the linear set abstraction have special handling for equality. This improves performance and precision at the cost of complexity. Instead, much of this complexity can be moved outside the abstraction and handled by lifting the abstraction to one that keeps track of equalities separately from other kinds of constraints.

The equality functor serves as an intermediary between the domain interface and the abstract domain that is being lifted. It intercepts equality constraints and handles them externally, preventing them from being seen by the underlying abstract domain. This saves the domain from the cost and complexity of handling the equalities.

The equality functor defines a set of equivalence classes Q. The set of equivalence classes is a map $\mathbb{X}_\mathbf{s} \to \mathbb{X}_\mathbf{s}$ that maps each variable to the chosen representative for the equivalence class. The functor then lifts an abstract state $\mathbb{D}^\sharp$ into a tuple $(\mathbb{D}^\sharp, Q)$. In the lifting, $\mathbb{D}^\sharp$ is restricted to only have symbols that are representatives for the equivalence class. Therefore, when an equality is added that merges two equivalence classes, the resulting representative replaces the two previous representatives in $\mathbb{D}^\sharp$.

The concretization ensures that all symbols in the same equivalence class map to the same concrete set:

$$\gamma((Q,\mathbb{D}^\sharp)) = \{ \sigma \mid \sigma \in \gamma(\mathbb{D}^\sharp) \wedge \forall X,Y \in \mathbb{X}_\mathbf{s}^2 . Q(X) = Q(Y) \to \sigma(X) = \sigma(Y) \}$$

Domain operations $\mathbf{join}_{\mathbb{D}^\sharp}$, $\mathbf{widen}_{\mathbb{D}^\sharp}$, and $\mathbf{is\_le}_{\mathbb{D}^\sharp}$ unify their corresponding Qs, pushing any non-common equalities into the underlying domain. This ensures that the underlying domain determines the precision, but it is not required to handle most of the load of the equalities. The $\mathbf{assume}_{\mathbb{D}^\sharp}$ operation rewrites the constraint, extracting the equalities and rewriting remaining variables to their representatives before passing the constraint to the underlying domain.

Example 5 (Equality functor join). Consider the following two abstract states, where the underlying domain is just shown as symbolic set constraints:

$$\sigma_0^\sharp = ([W \mapsto W, X \mapsto W, Y \mapsto W], (\!| W \subseteq Z |\!)) \quad \sigma_1^\sharp = ([X \mapsto X, Y \mapsto X], (\!| W \subseteq X \wedge X \subseteq Z |\!))$$

In the join, the equivalence classes are unified, producing the resulting Q: $[X \mapsto X, Y \mapsto X]$. The equality $(\!| W = X |\!)$ from $\sigma_0^\sharp$ is not represented in the unification, so it is added back to the underlying domain in $\sigma_0^\sharp$. The result is therefore

$$([X \mapsto X, Y \mapsto X], \mathbf{join}_{\mathbb{D}^\sharp}((\!| W = X \wedge W \subseteq Z |\!), (\!| W \subseteq X \wedge X \subseteq Z |\!)))$$

4.5 The Packing Domain Functor: Sparse Constraints

Most relational domains have a complexity that is related to the number of variables constrained by the abstract state. For example, BDDs, in the worst case, are exponential in the number of variables. However, in many programs, there are relatively small clusters of variables that are related. Therefore it is possible to increase the efficiency of an analysis by representing each cluster of variables by a separate abstract state [1].

If each of m clusters of n variables is represented by a separate abstract state, rather than operations having a complexity of, for example, $O(2^{m \cdot n})$, they can have complexity $O(m \cdot 2^n)$. To do this, all variables are initially assumed to be in their own cluster. Clusters are merged whenever variables from each cluster occur in the same constraint. In this way the clusters are dynamically determined, which is required when an abstract domain is used as a library and thus a pre-analysis cannot be performed.

An abstract state in the packing functor consists of one of three values: $\top$, $\bot$, or a map $M : \#_M \to \mathbb{D}^\sharp$ that maps cluster ids in $\#_M$ to abstract states from the domain being lifted. The $\top$ and $\bot$ values concretize as they do in Figure 4. The map concretizes as follows:

$$\gamma(M) = \{\sigma \mid \forall \sigma^\sharp \in \mathrm{Range}(M).\, \sigma \in \gamma(\sigma^\sharp)\}$$

Example 6 (Constraining a packed abstract state). Consider the following abstract state represented by the logic from Definition 1, lifted into two packs with ids 0 and 1: $\sigma_0^\sharp = [0 \mapsto (\!|X_0 \subseteq X_1|\!); 1 \mapsto (\!|Y_0 \subseteq Y_1|\!)]$. The operation $\mathbf{assume}_{\mathbb{D}^\sharp}(\sigma_0^\sharp, Y_1 \subseteq Y_2)$ operates only on pack id 1. It does not have to involve any computation on pack 0. The resulting pack 1 is: $1 \mapsto \mathbf{assume}_{\mathbb{D}^\sharp}((\!|Y_0 \subseteq Y_1|\!), Y_1 \subseteq Y_2)$.

5 Solver-based Abstractions

Because one of the core components of set abstraction is the Boolean algebra, it is possible to construct abstract domains from off-the-shelf satisfiability solvers. The construction relies upon the fact that the standard Boolean algebra is a finite height lattice ordered by implication. This means that no specific invariant generation procedure is required.

The syntax of the abstraction is the standard Boolean algebra with existential quantification. There are two reasons this is a good logic to use. First, it is a fairly well-supported logic for which there are efficient solvers. Second, it remains finite height and thus needs no specialized invariant generation procedure as would be required with a set logic such as BAPA [14]. Since cardinality is not a key requirement for symbolic sets, the analysis can often be sufficient without it.

Domain operations are translated into Boolean algebra formulas: $\mathbf{join}_{\mathbb{D}^\sharp}(\sigma_1^\sharp, \sigma_2^\sharp)$ translates into $\sigma_1^\sharp \vee \sigma_2^\sharp$; $\mathbf{assume}_{\mathbb{D}^\sharp}(\sigma^\sharp, L)$ translates into $\sigma^\sharp \wedge \mathrm{conv}(L)$, assuming that $\mathrm{conv}(L)$ converts the constraint L into its Boolean algebra equivalent as in Section 4.3; $\mathbf{forget}_{\mathbb{D}^\sharp}(\sigma^\sharp, X)$ translates into $\exists X.\, \sigma^\sharp$. These are accumulated across the whole analysis and thus may grow arbitrarily deep. It is possible that on-the-fly simplification could be used, but we elect to use whatever internal functionality is provided by the solver (in this case Z3 [11]).

Query operations are translated into solver queries. The implication test $\textbf{is\_le}_{\mathbb{D}^\sharp}(\sigma_1^\sharp,\sigma_2^\sharp)$ translates to $\textsc{Valid}(\sigma_1^\sharp \rightarrow \sigma_2^\sharp)$. This is implemented incrementally by conditionally adding constraints for each query and checking satisfiability under assumptions. The $\textbf{isbot}_{\mathbb{D}^\sharp}(\sigma^\sharp)$ query translates into $\textsc{Valid}(\neg\sigma^\sharp)$.

Example 7 (Solver-based abstraction operations). Domain operations accumulate constraints, so simplification is performed by the solver when a query happens. In the following sequence, there are no queries, so constraints only accumulate.

$$\begin{aligned}
\sigma_0^\sharp &= \top && = \textsc{True}\\
\sigma_1^\sharp &= \textbf{assume}_{\mathbb{D}^\sharp}(\sigma_0^\sharp, X \subseteq Y \wedge Y \subseteq Z) && = \textsc{True} \wedge X \rightarrow Y \wedge Y \rightarrow Z\\
\sigma_2^\sharp &= \textbf{forget}_{\mathbb{D}^\sharp}(\sigma_1^\sharp, Y) && = \exists Y.\, \textsc{True} \wedge X \rightarrow Y \wedge Y \rightarrow Z
\end{aligned}$$

If the query $\textbf{prove}_{\mathbb{D}^\sharp}(\sigma_2^\sharp, X \subseteq Z)$ is performed, the following check is made: $\textsc{Valid}((\exists Y.\, \textsc{True} \wedge X \rightarrow Y \wedge Y \rightarrow Z) \rightarrow (X \rightarrow Z))$. This holds trivially.

6 Evaluation

In this section, we evaluate the set abstractions. We aim to answer the following questions about set abstractions. Can set abstractions be sufficiently precise to be useful? Can precision be made available while providing scalability? What trade-offs are necessary to achieve scalability? To evaluate these questions we implemented all of the aforementioned abstractions as an OCaml library and then evaluated the abstractions using three different sets of problems: (1) traces of set domain operations as used in Memcad to perform shape analysis in the presence of unstructured sharing (from [16]), totaling 4521 domain operations; (2) traces of set domain operations as used in JSAna to verify functions in selected JavaScript libraries (from [7, 9]), totaling 23086 domain operations; and (3) the expressible subset of tests of the Python set data structure (as used for QUIC graphs [10]), totaling 207 lines of code. Results are shown in Table 1.

Because the definition of necessary precision depends on the use of a domain, we measure precision by comparing against a standard for precision. For Memcad, the linear set abstraction (**lin**) was designed to be as precise as is needed for the Memcad benchmarks. This means that any abstraction that achieves the same number of proofs without timeout is sufficiently precise. It is important to note that many of these proofs are not intended to succeed. They are used as queries internally with the analysis, so it is not possible to achieve 100%. From the results, we see that all of the BDD-based abstractions (**bdd**) achieve this. We can also see that the equality (eq) and packing (pack) functors, regardless of the order in which they are applied, do not change precision when applied to the BDD. However, when applied to the linear set abstraction, they sometimes change precision. This is because they affect internal representation and may affect the heuristics used within the abstract domain. QUIC graphs (**quic**) and SMT (**smt**) do not perform as well under any

Table 1. Number of proved properties ($\mathbf{prove}_{\mathbb{D}^\sharp}$), average aggregate run time for non-timed-out benchmarks (Time), and number of timed-out benchmarks (TO) for 24 Memcad benchmarks, 5 JSAna benchmarks, and 24 Python benchmarks.

Config	Memcad(24)		JSAna(5)		Python(24)	
	$\mathbf{prove}_{\mathbb{D}^\sharp}$	Time(TO)	$\mathbf{prove}_{\mathbb{D}^\sharp}$	Time(TO)	$\mathbf{prove}_{\mathbb{D}^\sharp}$	Time(TO)
lin	612/1366	0.036(0)	0/525	0.435(0)	4/42	0.004(0)
eq	608/1366	0.035(0)	0/525	0.235(0)	4/42	0.007(0)
pack	612/1366	0.049(0)	0/525	0.652(0)	4/42	0.006(0)
eq+pack	609/1366	0.045(0)	0/525	0.785(0)	4/42	0.011(0)
pack+eq	608/1366	0.067(0)	0/525	0.393(0)	4/42	0.011(0)
bdd	612/1366	0.021(0)	176/525	21.793(0)	34/42	0.105(0)
eq	612/1366	0.041(0)	176/525	1.206(0)	34/42	0.112(0)
pack	612/1366	0.052(0)	176/525	0.262(0)	34/42	0.109(0)
eq+pack	612/1366	0.055(0)	176/525	1.692(0)	34/42	0.116(0)
pack+eq	612/1366	0.086(0)	176/525	1.796(0)	34/42	0.119(0)
quic	596/1366	4.299(1)	155/525	54.616(0)	20/39	0.412(2)
eq	549/1366	2.289(0)	116/525	4.633(0)	18/39	0.416(2)
pack	605/1366	5.556(0)	155/525	48.517(0)	20/39	0.454(2)
eq+pack	549/1366	2.307(0)	121/525	8.201(0)	18/39	0.456(2)
pack+eq	55/58	0.080(10)	121/525	9.307(0)	17/38	0.362(3)
smt	177/315	44.995(4)	12/23	0.039(4)	34/41	0.296(1)
eq	416/927	35.798(1)	62/152	5.753(2)	31/41	0.294(1)
pack	177/315	16.329(4)	12/23	0.787(4)	34/41	5.553(1)
eq+pack	438/927	40.621(1)	27/73	9.355(3)	31/41	10.884(1)
pack+eq	231/458	12.027(3)	12/23	0.609(4)	31/41	10.838(1)

configuration. The reason is that QUIC graphs do not employ appropriate heuristics for all of the cases needed by Memcad and both have performance problems that cause them to time out before completing some benchmarks.

For the JSAna benchmarks, the BDD abstraction was designed to meet its precision needs and adding the equality or packing functor does not affect precision in any way. It only affects performance. However, the linear sets abstraction is not able to cope with the non-disjoint-union constraints that arise frequently in the JSAna benchmarks and thus loses all precision rapidly. By comparison, QUIC graphs perform well. They are unable to prove as many properties as is needed by JSAna, but they are still able to prove many properties. Once again, tuning the heuristics could improve this precision, but possibly at the cost of performance. SMT, once again, does not perform well because of efficiency problems. On the benchmarks where it completes, it is identical in precision to BDDs.

The Python benchmarks are slightly different because they are an analysis of programs rather than traces of domain operations. Each program contains a couple of properties to verify, so the target is 100%. Here we see that none of the abstractions are able to achieve 100%. The linear set abstraction cannot achieve this because it is unable to represent the non-disjoint-union constructs. The BDD and SMT abstractions cannot achieve 100% because they do not support full cardinality reasoning. Once again, QUIC graphs are insufficient because of the limited heuristics they employ as well as some performance problems.

The scalability of the abstractions can be seen in Table 1 in the total analysis time, which measures the time to run the full benchmark suite, on average. The times are only directly comparable if there are no time outs, which happens after 60 seconds per benchmark. We first see that the linear domain is reliably fast. Applying the equality and packing functors generally does not affect performance significantly. By comparison, BDDs are less reliable. While they perform well in the Memcad benchmarks, nearly matching the linear domain, we see significant variability in the JSAna benchmarks. In fact, without any of the functors as in [10], performance can be unacceptably slow at almost 22 seconds to analyze five functions. However, the addition of the packing functor, in particular, makes a significant difference. It lowers the cost of the analysis to a fraction of a second without losing any precision. However, the variability here indicates that, depending on the particular benchmark (or, in fact, the BDD implementation), the optimum combination of functors may vary. Regardless, selecting the packing functor seems to be a benefit without significant risk. The QUIC graphs performance is unreliable. Due to the expensive pattern matching machinery, it does not compare in terms of performance, though it is helped significantly by the equality functor, at the cost of precision. The SMT domain fails to perform, timing out on at least one test in each benchmark suite. This is because the SMT solver is failing to operate incrementally. In essence, it has the same workload as the BDD, but it discharges its proofs lazily. This laziness is not necessarily a problem if work can be reused from one proof to the next, but it appears that this is not the case right now. We suspect that the combination of doing validity proofs (instead of satisfiability queries) with quantifiers is preventing this reuse.

The results make four things clear. First, if it is possible to design a targeted abstraction as the linear abstraction is for Memcad, it is worth it. The performance is reliable and the precision is predictable. Second, if it is not clear what the constraints may be, BDDs provide a good alternative that gives excellent (if not perfect due to the insufficient cardinality reasoning) precision with the risk of less reliable performance. Third, much of the risk can be eliminated through the use of functors. For equality heavy loads, the equality functor provides a significant benefit. The packing functor seems to reliably improve performance by simply lowering the cost of each BDD operation without any measurable impact on precision. Lastly, unless the content-centric reasoning of QUIC graphs is necessary, it does not make sense to use it due to both unreliable performance and precision. Similarly, with the current state of SMT, this is not an appropriate use. It may be possible to fix this, but today it remains impractical for performance reasons.

7 Conclusions and Related Work

The problem of creating scalable, precise, and predictable abstractions for sets remains challenging. This paper introduced several ways of approaching this problem and showed that for symbolic set abstractions, binary decision diagrams offer good performance, precision, and predictability trade-offs. However, it is preferable to craft a custom abstraction such as the linear abstraction. This offers more predictable performance by only having the necessary precision.

There are other set abstractions available. They all offer different functionality at different costs. The QUIC graphs abstraction [8, 10] focuses on combining reasoning about contents with symbolic set reasoning. This comes at the cost of performance, precision, and predictability when it comes to purely symbolic set reasoning. The FixBag abstraction [19] attacks the problems of multisets or bags offering cardinality reasoning as well as symbolic set reasoning. Similar to QUIC graphs, it exchanges performance, precision, and predictability for this functionality. The linear and the BDD-based abstractions we present here are designed to be scalable, precise, and predictable rather than complex.

There are several decision procedures for sets. Bradley et al. [3] introduced a decision procedure for set contents and relationships (without cardinality). BAPA [13, 14] is a decision procedure for sets with cardinality. Z3 [11] also includes a decision procedure for sets with contents. None of these decision procedures are designed for invariant generation. It is possible that interpolation procedures [18] could be designed based upon these procedures, but to our knowledge this has not been done. Regardless, without invariant generation that is compatible with static analysis, it is difficult to use this work as a component of an existing analysis.

Due to the prevalence of Boolean algebra in the algorithms presented here, there is a natural correspondence to hardware model checking [4] and predicate abstraction [15]. However, one significant difference is the composability of the abstractions presented here. The equality and packing functors alter the underlying abstractions, making problems that were previously intractable, tractable. Additionally, because these are abstract domains, there is no conflation of control flow with data flow and as a result, many of the analysis problems are changed.

Additionally, the use of BDDs is similar to [17], where BDDs are extended to be possibly-cyclic graphs. These are used to represent tree structures.

As a result, we find that for now, abstractions that construct normal forms, such as the linear abstraction and binary decision diagrams, offer the best way of handling sets in static analysis. We have shown that depending on the application, both of these techniques offer sufficient performance and precision, especially when combined with functors for performing packing and managing equalities. The end result is that these abstractions are scalable, precise, and predictable in their behavior.

Acknowledgements. This material is based upon work supported in part by a Chateaubriand Fellowship, by the National Science Foundation under Grant Numbers CCF-1055066 and CCF-1218208, and by the European Research Council under the FP7 grant agreement 278673 (Project MemCAD).

References

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software.

In *PLDI*, 2003.

[2] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, 2012.

[3] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, 2006.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, 8(2), 1986.

[5] O. Coudert and J. C. Madre. A new method to compute prime and essential prime implicants of boolean functions. In *Advanced research in VLSI and Parallel Systems*. MIT, 1992.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[7] A. Cox, B. E. Chang, and X. Rival. Desynchronized multi-state abstractions for open programs in dynamic languages. In *ESOP*, 2015.

[8] A. Cox, B. E. Chang, and S. Sankaranarayanan. QUICr: A reusable library for parametric abstraction of sets and numbers. In *CAV*, 2014.

[9] A. Cox, B.-Y. E. Chang, and X. Rival. Automatic analysis of open objects in dynamic language programs. In *SAS*, 2014.

[10] A. Cox, B.-Y. E. Chang, and S. Sankaranarayanan. Quic graphs: Relational invariant generation for containers. In *ECOOP*, 2013.

[11] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[12] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.

[13] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, 2007.

[14] V. Kuncak, H. H. Nguyen, and M. C. Rinard. Deciding boolean algebra with presburger arithmetic. *J. Autom. Reasoning*, 36(3), 2006.

[15] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *CAV*, 2003.

[16] H. Li, B.-Y. E. Chang, and X. Rival. Shape analysis for unstructured sharing. In *SAS*, 2015.

[17] L. Mauborgne. *Representation of Sets of Trees for Abstract Interpretation*. PhD thesis, École Polytechnique, 1999.

[18] K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, 2003.

[19] T. Pham, M. Trinh, A. Truong, and W. Chin. Fixbag: A fixpoint calculator for quantified bag constraints. In *CAV*, 2011.

[20] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE, 2002.

[21] F. Somenzi. Binary decision diagrams. In *Calculational System Design*. IOS Press, 1999.

[22] F. Somenzi. Cudd: Cu decision diagram package, version 2.5.0. http://vlsi.colorado.edu/~fabio/CUDD/, 2012.