Efficient Predicate Abstraction of Program Summaries

Arie Gurfinkel, Sagar Chaki, and Samir Sapra

Carnegie Mellon University

Abstract. Predicate abstraction is an effective technique for scaling Software Model Checking to real programs. Traditionally, predicate abstraction abstracts each basic block of a program $\mathcal{P}$ to construct a small finite abstract model – a Boolean program BP, whose state-transition relation is over some chosen (finite) set of predicates. This is called Small-Block Encoding (SBE). A recent advancement is Large-Block Encoding (LBE) where abstraction is applied to a "summarized" program so that the abstract transitions of BP correspond to loop-free fragments of $\mathcal{P}$. In this paper, we expand on the original notion of LBE to promote flexibility. We explore and describe efficient ways to perform CEGAR bottleneck operations: generating and solving predicate abstraction queries (PAQs). We make the following contributions. First, we define a general notion of program summarization based on loop cutsets. Second, we give a linear time algorithm to construct PAQs for a loop-free fragment of a program. Third, we compare two approaches to solving PAQs: a classical AllSAT-based one, and a new one based on Linear Decision Diagrams (LDDs). The approaches are evaluated on a large benchmark from open-source software. Our results show that the new LDD-based approach significantly outperforms (and complements) the AllSAT one.

1 Introduction

Predicate abstraction is a well-established technique for scaling Software Model Checking to real systems [1]. Through predicate abstraction, model checking has been successfully applied to the verification of device drivers, hardware designs, and communication protocols. A core operation in predicate abstraction is the *predicate abstraction query* (PAQ): given a set of quantifier-free predicates P, and a quantifier-free formula e in some first-order theory, compute the strongest formula $\mathcal{G}_P(e)$ over P that is implied by e. It is used to over-approximate sets of states (when e and P are over program variables V), and transition relations (when e and P are over V and V').

Traditionally [1], PAQs are used to abstract transition relations of each individual basic block of an input program – this is called a Small-Block Encoding (SBE) [2]. Since transition relations of a basic blocks are simple (a few conjunctions of equalities) the corresponding PAQs are computationally simple as well [13]. Furthermore, SBE works well under a very coarse over-approximation of PAQs (e.g., via Cartesian abstraction [1] combined with an aggressive refinement [9]) simplifying PAQs even further. On the downside, SBE leads to a very

M. Bobaru et al. (Eds.): NFM 2011, LNCS 6617, pp. 131–145, 2011.
© Carnegie Mellon University, Software Engineering Institute 2011

large number of PAQs, a large number of predicates (often a different set for each basic block), and does not take advantage of the state-of-the-art in decision procedures. For example, a safety of a loop-free program can be proved with a single call to an SMT-solver, but with SBE often requires a large number of predicates and many iterations of the CounterExample Guided Abstraction Refinement (CEGAR) loop.

Beyer et al. [2] have proposed an alternative to SBE called the *Large-Block Encoding (LBE).* LBE lifts predicate abstraction to program summaries (i.e., loop-free program fragments). This leads to fewer PAQs, but the PAQs are more complex, harder to solve, and should not be over-approximated [2]. Overall, [2] shows that LBE is more efficient than SBE, and, even provably exponentially more efficient in some cases. While it is not clear whether LBE is preferable to SBE in all cases, LBE by itself presents three new problems for predicate abstraction. In this paper, we propose an expanded notion of LBE and present solutions to these problems:

(1) What types of program summaries are compatible with LBE? We show that LBE is compatible with a broad notion of a program summary. We argue that a *loop cutset summary* where where all loop-free fragments are summarized is a reasonable (but not the only) choice.

(2) How to efficiently generate PAQs? This problem is unique to LBE. We present a novel algorithm for generating queries for a summary that avoids constructing the summary itself. The algorithm takes a program in SSA form [8] and generates PAQs directly from the program's syntax. The size and the complexity of generating each query are linear in the size of the SSA.

(3) How to efficiently solve PAQs? With LBE, PAQs have a rich propositional structure. We present experiments with two algorithms: an AllSAT-based algorithm due to Lahiri et al. [16] (as implemented in MATHSAT4), and a novel algorithm based on Linear Decision Diagrams (LDDs) [5]. The two algorithms are evaluated on a benchmark derived from open-source programs. Surprisingly, we find that, on the whole benchmark, the LDD-based approach is superior to the AllSAT-based one. Interestingly, the approaches are complementary: we found that the "MIN" combination of the approaches (i.e., run both in parallel, stop as soon as one completes) is much more effective than either one in isolation.

To evaluate end-to-end performance of our approach in the CEGAR framework, we have built a safety checker for C and checked several classical examples from the literature. Our experiments indicate that LBE is more effective than SBE, and that the "MIN" combination of the AllSAT- and LDD-based approaches is most effective. We leave further comparison between LBE and SBE and the effect of LBE on the overall verification process to future work.

We envision that the algorithms proposed here will form a part of a complete CEGAR-based software analysis infrastructure. In particular, we do not argue for an exclusive use of any particular LBE or SBE. Instead, this work provides the flexibility necessary for an analyzer to (heuristically) choose a good block encoding and contributes efficient techniques to solve complex PAQs.

Related work. LBE for predicate abstraction was proposed by Beyer et al. [2]. They show that LBE significantly reduces the size of the abstract state space, the number of required predicates, and the verification time. They observe that success of LBE depends on precise predicate abstraction (as opposed to approximations such as Cartesian abstraction [1]). They use the AllSAT-based predicate abstraction [16] as implemented in MATHSAT4 [3]. We build on this work with a formal and general definition of LBE, new algorithms for efficiently constructing PAQs directly from an SSA program and for solving PAQs, and an extensive empirical evaluation on a large and challenging benchmark.

A naïve predicate abstraction algorithm – enumerating all satisfiable minterms – is exponential. Many heuristics have been proposed to improve its best-case complexity (e.g., [9,10]), and worst-case complexity at expense of completeness (e.g., [1,16]). For example, symbolic predicate abstraction [14,13] avoids exponentially many calls to an SMT solver by generating a symbolic proof from which the result is extracted by Boolean quantification, an exponential step.

Predicate abstraction is reducible to quantifier elimination. This leads to several solutions. In [16], the quantification is delegated to an AllSAT SMT solver. In [4], solutions are enumerated by a BDD and are discharged by an incremental SMT solver. Clarke et al. [6] use a SAT-solver for Boolean quantification for predicate abstraction over propositional logic. Lahiri et al. [15] give an algorithm for first-order logic via a reduction to propositional logic and Boolean quantification with either SAT- or BDD-based method.

In this paper, we propose another alternative: predicate abstraction is reduced to quantifier elimination over first-order logic, and the quantifiers are eliminated using LDDs [5]. On our benchmark this is much more efficient than the corresponding AllSAT-based solution.

The rest of the paper is structured as follows. Sec. 2 provides the necessary background. Sec. 3 describes program summarization. Sec. 4 presents algorithms to generate and solve PAQs. Sec. 5 presents experimental results. Sec. 6 concludes the paper.

2 Background

For a set of variables V, we write V' for $\{v' \mid v \in V\}$. For a binary relation ρ, we write $(s_1, s_2) \models \rho$ for $(s_1, s_2) \in \rho$. We write ρ^* for reflexive transitive closure, and $\rho \circ \rho$ for relational composition. We often represent sets and binary relations in the standard way by Boolean expressions over primed and unprimed variables. For an expression e, we write $e[V/V']$, or e', to mean the expression obtained by replacing each variable v in e with v'.

A *program* $\mathcal{P}$ is a tuple $(V, \mathcal{L}, \ell_0, \mathcal{T}, \mathcal{L}_\mathcal{E})$, where V is a set of variables, $\mathcal{L}$ a set of control locations, $\ell_0 \in (\mathcal{L} \setminus \mathcal{L}_\mathcal{E})$ a designated entry point, $\mathcal{T}$ a set of transitions, and $\mathcal{L}_\mathcal{E} \subset \mathcal{L}$ a set of exit locations. A *program state* is a valuation of all of the variables in V. The set of all states is denoted by Σ. Each *transition* $\tau \in \mathcal{T}$ is a triple (ℓ_1, ρ, ℓ_2), where $\ell_1, \ell_2 \in \mathcal{L}$ and $\rho \subseteq \Sigma \times \Sigma$ is a non-empty relation on program states. By convention, the entry location ℓ_0 and all exit locations in $\mathcal{L}_\mathcal{E}$ have no

incoming and outgoing transitions, respectively. The *control flow graph* (CFG) of $\mathcal{P}$, $CFG(\mathcal{P})$, is the graph $(\mathcal{L}, E)$, where $E = \{(\ell_1, \ell_2) \mid \exists \rho \cdot (\ell_1, \rho, \ell_2) \in \mathcal{T}\}$.

A *trace* in $\mathcal{P}$ is a finite sequence $\langle \ell_1, s_1 \rangle, \ldots, \langle \ell_n, s_n \rangle$ of location-state pairs such that $\forall 1 \leq i \leq (n-1) \cdot \exists \rho \cdot (\ell_i, \rho, \ell_{i+1}) \in \mathcal{T} \wedge (s_i, s_{i+1}) \models \rho$. A *computation*[1] is trace such that $\ell_1 = \ell_0$. A state s is *reachable* at location ℓ iff there exists a computation such that $\ell_n = \ell \wedge s_n = s$; a location ℓ is reachable iff there exists a state s reachable at ℓ; an *invariant* of $\mathcal{P}$ at ℓ is any superset of the states reachable at ℓ.

A predicate is any ground formula. A *cube* over a set of predicates P is a formula of the form $p_1 \wedge \cdots \wedge p_n \wedge \neg q_1 \wedge \cdots \wedge \neg q_m$, where $p_i, q_j \in P$ and every predicate appears at most once. A *minterm* is a cube of size $|P|$.

Let ψ be a quantifier-free first-order expression. A fundamental operation of predicate abstraction is to compute $\mathcal{G}_P(\psi)$ – a strongest Boolean combination of P that is implied by ψ. $\mathcal{G}_P(\psi)$ can be characterized as the set of all minterms that do not contradict ψ:

$$\mathcal{G}_P(\psi) = \bigvee \{c \mid c \text{ is a minterm over } P \text{ and } c \wedge \psi \text{ is satisfiable}\}.$$

$\mathcal{G}_P(\psi)$ can be computed by enumerating all minterms and using a decision procedure to decide satisfiability. Alternatively, the computation can be reduced to quantifier elimination as follows. With each $p \in P$ associate a unique Boolean variable b_p; let V be the set of all free variables in ψ and P; and let F_P be the formula $\psi \wedge (\bigwedge_{p \in P} b_p \Leftrightarrow p)$. Then, $\mathcal{G}_P(\psi)$ is given by the result of eliminating all existential quantifiers in $\exists V \cdot F_P$, and then replacing every b_p with the predicate p.

Let $\mathcal{P} = (V, \mathcal{L}, \ell_0, \mathcal{T}, \mathcal{L}_\mathcal{E})$ be a program, and μ a *predicate map* that assigns to each location ℓ a set of predicates denoted $\mu.\ell$. The *(most precise) predicate abstraction* of $\mathcal{P}$ with respect to μ is a program $\mathcal{P}_\mu = (V, \mathcal{L}, \ell_0, \mathcal{T}_\mu, \mathcal{L}_\mathcal{E})$, where

$$\mathcal{T}_\mu = \{(\ell_1, \mathcal{G}_P(\rho), \ell_2) \mid (\ell_1, \rho, \ell_2) \in \mathcal{T} \text{ and } P = \mu.\ell_1 \cup \mu.\ell_2'\}.$$

Note that if μ is finite for every program location, then $\mathcal{P}_\mu$ is finite as well.

3 Program Summary

Large-Block Encoding applies predicate abstraction to a *summary* of a program. The original definition of LBE [2] uses a specific notion of summary, which we call *rule summary*. In this section, we present a more general concept of summaries. In particular, we define a *loop cutset summary* as the most general summary that summarizes all loop-free program fragments. Cutset summaries subsume useful classes of summaries, including (as we show later) the rule summary.

Let $\mathcal{P} = (V, \mathcal{L}, \ell_0, \mathcal{T}, \mathcal{L}_\mathcal{E})$ be a program; let $\mathcal{L}' \subseteq \mathcal{L}$ such that $\ell_1, \ell_n \in \mathcal{L}'$. A trace $\langle \ell_1, s_1 \rangle, \ldots, \langle \ell_n, s_n \rangle$ of $\mathcal{P}$ is $\mathcal{L}'$*-free* iff $\mathcal{L}' \cap \{\ell_2, \ldots, \ell_{n-1}\} = \emptyset$. The $\mathcal{L}'$-free (ℓ_1, ℓ_n) fragment of $\mathcal{P}$ comprises locations appearing on $\mathcal{L}'$-free (ℓ_1, ℓ_n) traces of $\mathcal{P}$, with ℓ_1 and ℓ_n as entry and exit locations, respectively.

[1] In this paper, we only consider finite computations.

Definition 1 (Summary). *A program $\mathcal{P}' = (V, \mathcal{L}', \ell_0, \mathcal{T}', \mathcal{L}_\mathcal{E})$ is a* summary *of a program $\mathcal{P} = (V, \mathcal{L}, \ell_0, \mathcal{T}, \mathcal{L}_\mathcal{E})$ iff: (i) $\mathcal{L}' \subseteq \mathcal{L}$, and (ii) $\forall \ell_1, \ell_n \in \mathcal{L}'$ there exists a $\mathcal{L}'$-free (ℓ_1, ℓ_n) trace $\langle \ell_1, s_1 \rangle, \ldots, \langle \ell_n, s_n \rangle$ of $\mathcal{P}$ iff $\exists \rho \centerdot (\ell_1, \rho, \ell_n) \in \mathcal{T}' \wedge (s_1, s_n) \models \rho$.*

A program and its summary share the same variables, entry and exit locations, and state space Σ. A summary also preserves reachability of locations, as stated by Theorem 1.

Theorem 1. *Let $\mathcal{P}' = (V, \mathcal{L}', \ell_0, \mathcal{T}', \mathcal{L}_\mathcal{E})$ be a summary of $\mathcal{P} = (V, \mathcal{L}, \ell_0, \mathcal{T}, \mathcal{L}_\mathcal{E})$. Then, $\forall \ell \in \mathcal{L}'$, $s \in \Sigma$ is reachable at ℓ in $\mathcal{P}$ iff s is reachable at ℓ in $\mathcal{P}'$.*

As a corollary, since an invariant is a set of states, a summary also preserves invariants: I is an invariant of $\mathcal{P}'$ at $\ell \in \mathcal{L}'$ iff it is an invariant of $\mathcal{P}$ at ℓ. Thus, any program summary can be used for LBE. Ideally, we want the smallest summary possible since it leads to smaller abstract models. In particular, we'd like a unique minimal summary – when $\mathcal{L}' = \{\ell_0\} \cup \mathcal{L}_\mathcal{E}$. Unfortunately, it is not computable since its computation requires summarizing program loops. Instead, we want the smallest summary that summarizes only loop-free program fragments.

Let $G = (V, E)$ be a graph. A set $S \subseteq V$ is a *cycle cutset* (or simply a cutset) of G iff S contains a vertex from every cycle in G, i.e., the graph $(V \setminus S, E \setminus ((S \times V) \cup (V \times S)))$ is acyclic. We call an element $s \in S$ a *cutpoint*.

Definition 2 (Loop Cutset Summary). *A program $\mathcal{P}' = (V, \mathcal{L}', \ell_0, \mathcal{T}', \mathcal{L}_\mathcal{E})$ is a* cutset summary *of $\mathcal{P}$ iff $\mathcal{P}'$ is a summary of $\mathcal{P}$ and $\mathcal{L}'$ is a cutset of $CFG(\mathcal{P})$.*

The cutset summary of a program is not unique. Finding a minimal one is hard since it requires solving the *minimal feedback vertex set*, which is known to be NP-complete [12]. However, in practice, a good approximation is obtained in polynomial time by letting $\mathcal{L}'$ be the set of destinations of all back-edges discovered by a DFS of $CFG(\mathcal{P})$, together with ℓ_0 and $\mathcal{L}_\mathcal{E}$. Given a cutset of $CFG(\mathcal{P})$, the corresponding cutset summary of $\mathcal{P}$ is effectively computable since, by definition, each edge in it corresponds to a loop-free fragment of $\mathcal{P}$.

In the rest of this section, we compare cutset summaries with rule summaries [2]. A rule summary is based on two program transformations, SEQ and CHOICE. Let $\mathcal{P} = (V, \mathcal{L}, \ell_0, \mathcal{T}, \mathcal{L}_\mathcal{E})$, and $\ell_1, \ell_2 \in \mathcal{L}$ be two locations. The preconditions of SEQ$(\mathcal{P}, \ell_1, \ell_2)$ are (a) $\ell_1 \neq \ell_2$, (b) there is an edge from ℓ_1 to ℓ_2, (c) ℓ_2 has no other incoming edges, and (d) ℓ_2 has at least one successor. The output is the program $\mathcal{P}' = (V, \mathcal{L}', \ell_0, \mathcal{T}', \mathcal{L}_\mathcal{E})$, where $\mathcal{L}' = \mathcal{L} \setminus \{\ell_2\}$ and

$$\mathcal{T}' = (\mathcal{T} \cup \{(\ell_1, \rho \circ \rho_i, \ell_i) \mid (\ell_2, \rho_i, \ell_i) \in \mathrm{out}(\ell_2)\}) \setminus (\mathrm{out}(\ell_2) \cup \mathrm{in}(\ell_2)) \ ,$$

where $\mathrm{out}(\ell)$ and $\mathrm{in}(\ell)$ are the sets of all outgoing and incoming transitions of ℓ, respectively. The precondition of CHOICE$(\mathcal{P}, \ell_1, \ell_2)$ is that there are two distinct edges (ℓ_1, ρ_1, ℓ_2) and (ℓ_1, ρ_2, ℓ_2) in $\mathcal{T}$. The output is $\mathcal{P}' = (V, \mathcal{L}, \ell_0, \mathcal{T}', \mathcal{L}_\mathcal{E})$, where

$$\mathcal{T}' = (\mathcal{T} \setminus \{(\ell_1, \rho_1, \ell_2), (\ell_1, \rho_2, \ell_2)\}) \cup \{(\ell_1, \rho_1 \cup \rho_2, \ell_2)\} \, .$$

Intuitively, SEQ removes a location with a single incoming edge, and CHOICE replaces multiple edges between the same locations with a single one.

Definition 3 (Rule Summary). *A* rule summary *of a program* $\mathcal{P} = (V, \mathcal{L}, \ell_0, \mathcal{T}, \mathcal{L}_\mathcal{E})$*, is a limit of the sequence* $\mathcal{P}_0, \mathcal{P}_1, \ldots$*, where* $\mathcal{P}_0 = \mathcal{P}$*, and* $\mathcal{P}_{i+1}$ *is* SEQ$(\mathcal{P}_i, \ell_1, \ell_2)$ *if* SEQ *is applicable,* CHOICE$(\mathcal{P}_i, \ell_1, \ell_2)$ *if* CHOICE *is applicable, and* $\mathcal{P}_i$ *otherwise.*

The advantage of a cutset summary is that it is not restricted to a particular cutset computation procedure. For example, suppose we construct a cutset of $\mathcal{P}$ by taking the destinations of all back-edges in a topological ordering of $CFG(\mathcal{P})$. Let us call this *back-edge* summary and compare it to rule summary. In both cases, the complexity of constructing a summary is polynomial in the size of $\mathcal{P}$. However, the locations of a rule summary of $\mathcal{P}$ subsumes those of a back-edge summary of $\mathcal{P}$. This is because: (i) the destination of a back-edge always has at least two incoming edges, and hence can never be removed by SEQ, and (ii) CHOICE never eliminates locations. Thus, back-edge summary is never larger than a rule summary. Thus, a rule summary is a custset summary as well.

4 Predicate Abstraction of Program Fragments

A cutset C of a program $\mathcal{P}$ has a BACK-EDGE-AT-END property if for every $\ell_1, \ell_2 \in C$, ℓ_2 is the sole destination of all back-edges in the C-free fragment $\mathcal{P}_{\ell_1,\ell_2}$. Note that a cutset C of any back-edge (or rule) summary satisfies this since in any C-free fragment $\mathcal{P}_{\ell_1,\ell_2}$, $\{\ell_1, \ell_2\}$ are the only possible destinations of back-edges, but ℓ_1 has no incoming edges at all. Let $\mathcal{P}$ be a program and C its BACK-EDGE-AT-END cutset. In this section, we show how to compute a predicate abstraction of a cutset summary of $\mathcal{P}$ (w.r.t. C), without explicitly constructing the summary.

Our algorithm is called SummaryPA, and is shown in Fig. 1. We assume that $\mathcal{P}$ is given in Static Single Assignment (SSA), and work directly on its syntax. Function EdgeQuery takes $\mathcal{P}$ and two locations $\ell_1, \ell_2 \in C$ and generates a PAQ for the C-free (ℓ_1, ℓ_2) fragment, $\mathcal{P}_{\ell_1,\ell_2}$, of $\mathcal{P}$. The size of the PAQ is linear in the size of $\mathcal{P}_{\ell_1,\ell_2}$. Function Solve takes this query and returns a predicate abstraction of $\mathcal{P}_{\ell_1,\ell_2}$. These two functions are applied to every pair of connected cutpoints from C. The output of SummaryPA is the predicate abstraction of the summary of $\mathcal{P}$ w.r.t. C. In the rest of this section, we give a brief overview of SSA, describe the formula constructed by EdgeQuery, and give two strategies for Solve: one based on an AllSAT SMT-solver, and one based on LDDs.

4.1 Single Static Assignment

We give here a brief overview of SSA. More details can be found elsewhere [8]. A program is in SSA form if an assignment to each variable appears at most once in its syntax. Any program can be put efficiently into SSA. As an example, an SSA program corresponding to the C program in Fig. 2(a) is shown in Fig. 2(b).

In addition to normal assignments, SSA uses special *ϕ-assignments.* Their syntax is x `:=` `PHI`$(v_1 : \ell_1, \ldots, v_n : \ell_n)$, where x is a variable, $\ell_1, \ldots, \ell_n$ are locations, and $v_1, \ldots, v_n$ are values. The `PHI`-function evaluates to value v_i if it

1: **Input:** SSA program $\mathcal{P} = (V, \mathcal{L}, \ell_0, \mathcal{T}, \mathcal{L}_\mathcal{E})$; a cutset C of $\mathcal{P}$; a predicate map μ
2: **Output:** P_μ the most precise predicate abstraction of P w.r.t. μ
3: **function** SummaryPA $(\mathcal{P}, C, \mu)$
4: $\quad \mathcal{T}_\mu = \emptyset$
5: $\quad$ **for all** $\ell_1, \ell_2 \in C$ s.t. $\exists$ a C-free (ℓ_1, ℓ_2)-path in CFG of $\mathcal{P}$ **do**
6: $\quad\quad Q = \text{EdgeQuery}(\mathcal{P}, C, \ell_1, \ell_2, \mu.\ell_1, \mu.\ell_2)$
7: $\quad\quad \rho = \text{Solve}(Q); \quad \mathcal{T}_\mu = \mathcal{T}_\mu \cup \{(\ell_1, \rho, \ell_2)\}$
8: $\quad P_\mu = (V, C, \ell_0, \mathcal{T}_\mu, \mathcal{L}_\mathcal{E})$

Fig. 1. Algorithm SummaryPA

is reached via location ℓ_i. In our example, `PHI(0:0, x_0:4)` on line 2 evaluates to 0 when reached from location 0 and to `x_0` when reached from location 4.

We model an SSA program as a tuple $(V, \mathcal{L}, E, \phi, G, Act, \ell_0, \mathcal{L}_\mathcal{E})$ where: V, $\mathcal{L}$, ℓ_0, and $\mathcal{L}_\mathcal{E}$ are same as in programs, $E \subseteq \mathcal{L} \times \mathcal{L}$ is the set of control flow edges, Act maps locations to assignments, and ϕ and G map edges to ϕ-assignments and guards, respectively. Intuitively, each $\ell \in \mathcal{L}$ corresponds to a basic block; each basic block is a sequence of assignments terminated by a branch; the branch condition is stored on the edge, and each ϕ-assignment is replaced by the corresponding assignments on the edges. This is a variant of the traditional compiler SSA format, where ϕ-assignments and guards are pushed into the source and destination blocks of their edges, respectively. Fig. 2(c) graphically shows the SSA program from Fig. 2(b).

Operationally, an edge (ℓ_1, ℓ_2) in an SSA program is executed by: (a) executing the assignments $Act(\ell_1)$, (b) validating the guard $G(\ell_1, \ell_2)$, and (c) executing ϕ-assignments $\phi(\ell_1, \ell_2)$. Formally, for a set of assignments A, let $\alpha(A)$ be $\bigwedge_{v:=e \in A} v = e$, and $\alpha'(A)$ be $\bigwedge_{v:=e \in A} v' = e$. The semantics of an SSA program $\mathcal{P} = (V, \mathcal{L}, E, \phi, G, Act, \ell_0, \mathcal{L}_\mathcal{E})$ is a program $\mathcal{P}' = (V, \mathcal{L}, \ell_0, \mathcal{T}', \mathcal{L}_\mathcal{E})$, s.t. $(\ell_1, \rho, \ell_2) \in \mathcal{T}'$ iff $\rho = \alpha'(Act(\ell_1)) \wedge Skip(K) \wedge G(\ell_1, \ell_2)' \wedge \alpha(\phi(\ell_1, \ell_2))'$, where $K = \{v \in V \mid \neg\exists e \centerdot (v := e) \in A \vee (v := e) \in \phi(\ell_1, \ell_2)\}$ and $Skip(U)$ is $\bigwedge_{u \in U} u' = u$. For example, the semantics of SSA program in Fig. 2(c) is shown in Fig. 2(d). The semantics of the edge $(3, 2)$ is $y_0' = y + 1 \wedge y' = y_0' \wedge x' = x \wedge x_0' = x_0$. Note that this definition depends on several properties of the SSA: assignments in a block have no circular dependencies, guards do not depend on following ϕ-assignments, etc.

4.2 Generating Predicate Abstraction Queries

EdgeQuery$(\mathcal{P}, C, \ell_1, \ell_2, P_1, P_2)$ takes an SSA program $\mathcal{P}$, a cutset C, locations ℓ_1, and ℓ_2 in C, and two sets of predicates P_1 and P_2, and generates a PAQ for C-free (ℓ_1, ℓ_2)-fragment, $\mathcal{P}_{\ell_1, \ell_2}$, of $\mathcal{P}$.

The result of EdgeQuery is similar to a typical "reachability query" , e.g., as in CBMC [7]. It is linear in the size of $\mathcal{P}_{\ell_1, \ell_2}$ and computable in linear time. However, EdgeQuery works directly on SSA (as opposed to a more expensive Gated SSA used in [7]). The resulting PAQ separates control- and data-flows,

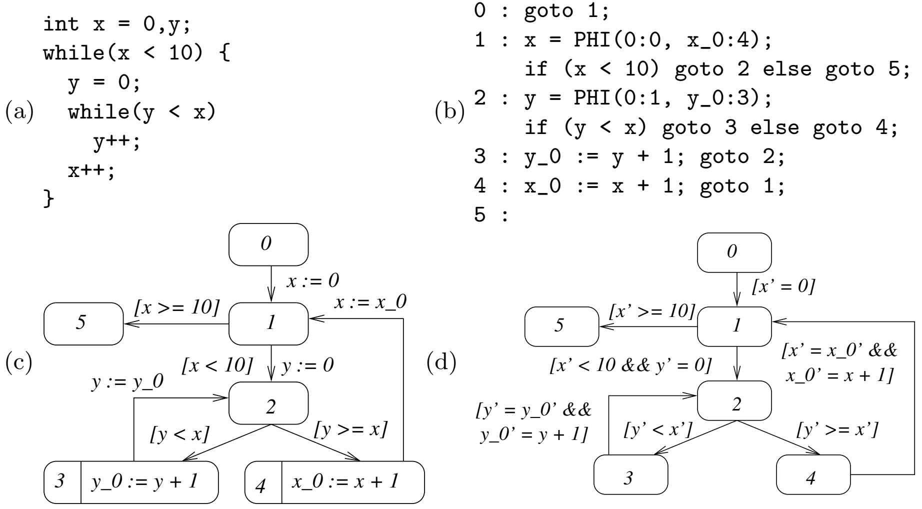


Fig. 2. Representation of a C program: (a) traditional, (b) SSA, (c) graphical SSA, (d) semantic. In (d), all expressions of the form $v' = v$ have been omitted.

and preservers control-flow structure in the query. These features are crucial for our approach to discharging PAQs (see *LDD-based approach* in Section 4.3).

For ease of understanding, we present the query in parts. Let $\mathcal{L}_f$ denote the set of all locations of $\mathcal{P}_{\ell_1,\ell_2}$. Let A be a formula for all of the simple assignments in the fragment, $A = \bigwedge_{\ell \in \mathcal{L}_f \setminus \{\ell_2\}} \alpha(Act(\ell))$, where α is as defined in Sec. 4.1. Intuitively, a complete satisfying assignment to A corresponds to executing, in parallel, all assignments of all of the locations in $\mathcal{L}_f$. A is always satisfiable because, by assumption, $\mathcal{P}_{\ell_1,\ell_2}$ has no back-edges (except possibly to ℓ_2), and hence no circularly dependent assignments.

For each $\ell \in \mathcal{L}$, let B_ℓ be a Boolean variable corresponding to ℓ, and V_ℓ the set of all such variables. Let R_ℓ be a formula defined for a location ℓ as follows:

$$R_\ell = \left(B_\ell \Rightarrow \bigvee_{\ell' \in Preds(\ell) \cap \mathcal{L}_f} B_{\ell'} \wedge G(\ell', \ell) \wedge \alpha(\phi(\ell', \ell)) \right) ,$$

where $Preds(\ell)$ is the set of all CFG-predecessors of ℓ. Intuitively, B_ℓ represents whether ℓ is visited in an execution, i.e., is reachable. R_ℓ states that if ℓ is reachable then at least one (but possibly more) of its predecessors ℓ' must be reachable, and the guards and the ϕ-assignments on the (ℓ', ℓ)-edge must be true.

For the final location ℓ_2, we need a variant of R_ℓ, denoted $\hat{R}_\ell$ and defined as:

$$\hat{R}_\ell = \left(B_\ell \Rightarrow \bigvee_{\ell' \in Preds(\ell) \cap \mathcal{L}_f} B_{\ell'} \wedge G(\ell', \ell) \wedge \alpha'(\phi(\ell', \ell)) \right) ,$$

where α' is as defined in Sec. 4.1. Since ℓ_2 can be the destination of a back-edge, the ϕ-assignment on that edge might be circularly dependent on another assignment in $\mathcal{P}_{\ell_1,\ell_2}$. Such dependencies are eliminated by using α' instead of α.

Next, we define a formula *CFG* as follows:

$$\mathit{CFG} = \left(B_{\ell_2} \wedge \hat{R}_{\ell_2} \wedge \bigwedge_{\ell \in \mathcal{L}_f \setminus \{\ell_1,\ell_2\}} R_\ell \right) .$$

Every satisfying assignment to *CFG* corresponds to one (or several) paths of $\mathcal{P}_{\ell_1,\ell_2}$. B_{ℓ_2} guarantees that ℓ_2 is visited, the implications in R_ℓ create the path, and the guard and ϕ-assignment constraints ensure that the path is feasible (i.e., can always be elaborated into a concrete computation).

Consider the formula $A \wedge \mathit{CFG}$. Each satisfying assignment to it corresponds to at least one concrete execution from ℓ_1 to ℓ_2. Furthermore, note that any assignment that corresponds to multiple non-contradicting executions can be transformed into a satisfying assignment for a single execution. This is done by picking one of the corresponding executions, setting B_ℓ to true for every location ℓ on that execution, and setting all other B_ℓ variables to false.

Next, we need formulas for predicates. With each predicate $p \in P_1$ we associate a Boolean variable b_p, and with each predicate $p \in P_2$ a Boolean variable b'_p. Let *Src* and *Dst* be formulas defined as:

$$\mathit{Src} = \left(\bigwedge_{p \in P_1} b_p \Leftrightarrow p \right) \qquad \mathit{Dst} = \left(\bigwedge_{p \in P_2} b'_p \Leftrightarrow \Phi(p) \right) ,$$

where $\Phi(p) = p[v/v' \mid \exists \ell \in (\mathit{Preds}(\ell_2) \cap \mathcal{L}_f) \,.\, v \in LHS(\phi(\ell, \ell_2))]$. Note that this renaming in *Dst* corresponds to the renaming in $\hat{R}_\ell$.

Finally, the PAQ produced by EdgeQuery is

$$\exists V, V', V_\ell \,.\; A \wedge \mathit{CFG} \wedge \mathit{Src} \wedge \mathit{Dst} .$$

This formula is linear in $|\mathcal{L}_f|$ and can be computed in linear time. Theorem 2 asserts the correctness of EdgeQuery.

Theorem 2. *Let $\rho \subseteq \Sigma \times \Sigma$ be the summary of $\mathcal{P}_{\ell_1,\ell_2}$. Then,* EdgeQuery$(\mathcal{P}, C, \ell_1, \ell_2, P_1, P_2)$ *is equivalent to:*

$$\exists V, V' \,.\, \rho \wedge \left(\bigwedge_{p \in P_1} b_p \Leftrightarrow p \right) \wedge \left(\bigwedge_{p \in P_2} b'_p \Leftrightarrow p' \right)$$

Example 1. Let $\mathcal{P}$ be the SSA program from Fig. 2(c) and $C = \{0, 1, 2, 5\}$ its loop cutset. Consider the C-free $(2, 2)$ fragment of $\mathcal{P}$ and predicates $y < 0$, $x < 0$.

$$\mathcal{L}_f = \{2, 3\} \qquad A = (y_0 = y + 1)$$
$$\mathit{Src} = (b_y \Leftrightarrow y < 0) \wedge (b_x \Leftrightarrow x < 0) \qquad \hat{R}_2 = (B_2 \Rightarrow B_3 \wedge y' = y_0)$$
$$\mathit{Dst} = (b'_y \Leftrightarrow y' < 0) \wedge (b'_x \Leftrightarrow x < 0) \qquad R_3 = (B_3 \Rightarrow B_2 \wedge y < x)$$

The overall predicate abstraction query is:

$$\exists y_0, y, y', x, B_2, B_3 \centerdot (y_0 = y + 1) \wedge (B_3 \Rightarrow B_2 \wedge y < x) \wedge (B_2 \Rightarrow B_3 \wedge y' = y_0) \wedge \\ B_2 \wedge (b_y \Leftrightarrow y < 0) \wedge (b'_y \Leftrightarrow y' < 0) \wedge (b_x \Leftrightarrow x < 0) \wedge (b'_x \Leftrightarrow x < 0)\,.$$

4.3 Solving Predicate Abstraction Queries

Solve takes a PAQ of the form $\exists V, V', V_\ell \centerdot \Psi$, eliminates the quantifiers, and then replaces the Boolean variables introduced by EdgeQuery by the corresponding predicates. In this section, we describe two strategies for the quantifier elimination step: *AllSAT-based* – based on the approach of Lahiri et al. [16], and *LDD-based* – based on a recently developed decision diagrams LDDs [5].

AllSAT-based approach. An SMT solver decides satisfiability of a quantifier-free first-order formula F (over a theory T). An AllSAT SMT solver takes a formula F and a subset V_{Imp} of *important* Boolean terms of F and returns the set M of all minterms over V_{Imp} that can be extended to a satisfying assignment to F.

A PAQ of the form $\exists V, V', V_\ell \centerdot \Psi$ is solved by giving an AllSAT solver a formula Ψ and setting V_{Imp} to the set of all Boolean variables b_p and b'_p in Ψ. The output is a set M of minterms such that $\bigvee M$ is equivalent to $\exists V, V', V_\ell \centerdot \Psi$.

The key advantage of this approach is that all of the reasoning is delegated to an AllSAT solver. Thus, it applies to queries in any SMT-supported theory and leverages advancements in SMT-solvers. The main limitation – it enumerates all minterms of the solution, which can be exponentially larger than the smallest DNF representation. We illustrate this limitation further in Sec. 5.

LDD-based approach. An LDD is a Binary Decision Diagram (BDD) with nodes labeled with atomic terms from Linear Arithmetic (LA). An LDD represents a LA formula in the same way a BDD represents a Boolean formula. LDDs support the usual Boolean operations (conjunction, disjunction, negation, etc.), Boolean quantification, and variable reordering. Additionally, they provide quantification over numeric variables via direct Fourier-Motzkin elimination on the diagram.

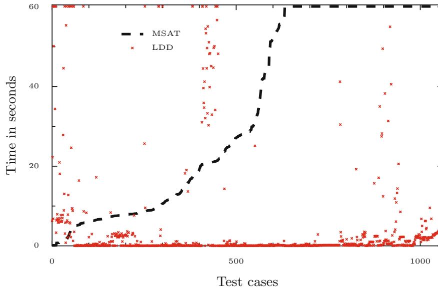
Let ℓ_1 and ℓ_k be two cutpoints. Recall that the query Q computed by EdgeQuery for (ℓ_1, ℓ_k) is of the form $\exists V, V', V_\ell \centerdot \Psi$, where

$$\Psi = A \wedge R_{\ell_2} \wedge \cdots \wedge \hat{R}_{\ell_k} \wedge B_{\ell_k} \wedge \mathit{Src} \wedge \mathit{Dst}\,,$$

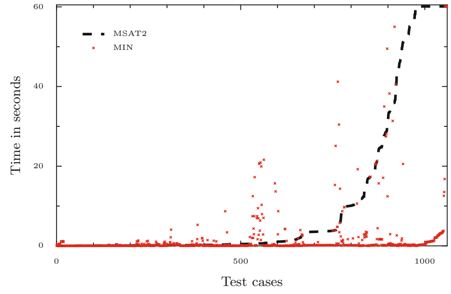
and each R_{ℓ_i} is of the form $B_{\ell_i} \Rightarrow \theta_i$. A naïve way to solve Q is to first compute and LDD for Ψ, and then use numeric and Boolean quantification to eliminate the variables in V, V', and V_ℓ. Note that the result is a BDD (since all of the remaining variables are Boolean).

Unfortunately, the naïve approach does not scale. The bottleneck is constructing a diagram for Ψ. In large part, this is due to the variables B_ℓ used in encoding control-flow constraints. The solution is to re-arrange the query to eliminate these variables as early as possible.

Let $\mathcal{L}_f = \langle \ell_1, \ldots, \ell_k \rangle$ be the set of locations in the cutpoint-free – and hence, loop-free – (ℓ_1, ℓ_k) fragment of $\mathcal{P}$. Let *Topo* be a topological order on $\mathcal{L}_f$. Without



(a) Running times for experiments MSAT and LDD



(b) Running times for experiments MSAT2 and MIN

Fig. 3. Running times

loss of generality, assume that the locations are numbered such that $i \leq j$ iff ℓ_i precedes ℓ_j in some fixed linearization of *Topo*. Then, a variable B_{ℓ_i} appears in a constraint R_{ℓ_j} iff $i \leq j$. Therefore, Q is equivalent to $\exists V, V' \centerdot \Psi'$, where

$$\Psi' = A \wedge Src \wedge Dst \wedge \left(\exists B_{\ell_1} \centerdot \exists B_{\ell_2} \centerdot R_{\ell_2} \wedge \cdots \wedge \exists B_{\ell_k} \centerdot \hat{R}_{\ell_k} \wedge B_{\ell_k}\right) .$$

In summary, our overall solution is to compute an LDD for Ψ', and then use numeric quantification to eliminate V and V' variables. Note that it is possible to apply early quantification to the numeric variables as well. However, we did not explore this direction.

The main advantage of our approach is that the solution is computed directly as an LDD. Thus, its running time is independent of the number of minterms in the solution. Unlike the AllSAT-based approach, it is limited to Linear Arithmetic and does not directly benefit from advances in SMT-solving. However, in our experiments, it significantly outperformed the AllSAT-based approach.

We are not the first to use decision diagrams for predicate abstraction. However, previous approaches use BDDs by reducing numeric reasoning to propositional reasoning. This reduction introduces a large number of Boolean variables, which makes the problem hard for BDDs. For example, Lahiri et al. [15] find a SAT-based approach superior to a BDD-based one. In contrast, we use decision diagrams that are aware of Linear Arithmetic. This avoids the need for additional constraints, and makes the solution very competitive.

5 Experimental Results

We evaluated our approach on a large benchmark of PAQs and as a part of a software model checker. We used the MATHSAT4 SMT-solver [3] for the AllSAT-based solution, and our implementation of LDDs [5] for the LDD-based solution. All PAQs were restricted to two-variables-per-inequality logic (TVPI), i.e., linear constraints with at most two variables. The benchmark and our tools are available at `lindd.sf.net`.

The benchmark. To evaluate our approach on large queries, we constructed the benchmark from C programs using the following technique: (1) convert a program into LLVM bitcode [17] and optimize with loop unrolling and inlining; (2) for each function, use all loop headers as the cutset summary; (3) over-approximate the semantics of statements by TVPI constraints (e.g., loads from memory and function calls are replaced by non-determinism); (4) for each location ℓ, take the atomic formulas that appear in the weakest precondition of some conditional branch reachable from ℓ as the predicates at ℓ; (5) for each pair of locations ℓ and ℓ' in the summary, generate a PAQ, as described in Sec. 4.2, using the predicates at ℓ and ℓ'.

In our view, the benchmark is quite realistic: steps 1-3 are a common pre-processing techniques in program analysis; the choice of predicates is guided by our experience with predicate abstraction.

The benchmark consists of over 20K PAQs. We report the results on the top 1061 cases (exactly the ones that required $\geq$ 5s to solve with at least one approach). These PAQs are from `bash`, `bison`, `ffmpeg`, `gdb`, `gmp`, `httpd`, `imagemagick`, `mplayer`, and `tar`. As formulae in SMT-LIB format, they range in size from 280B to 57KB (avg. 11KB, med. 8KB). The number of predicates per query ranges from 10 to 56 (avg. 22, med. 19). Each experiment was limited to 60s CPU and 512MB of RAM, and was done on a 3.4GHz Pentium D with 2GB of RAM.

The experiments. The results of the experiments are summarized in the first three rows of Table 4a. The first column indicates the experiment as follows – *MSAT*: queries are solved using MATHSAT4; *LDD*: queries are solved using LDDs with dynamic variable order (DVO); and *LDD2*: queries are solved using LDDs with static variable order (SVO). For LDD, diagram nodes were reordered by the diagram manager based on memory utilization. For LDD2, a static order was selected such that terms that appeared earlier in the *query AST* would appear earlier in the diagram order. A query AST is $((A \wedge CFG) \wedge Src \wedge Dst)$.

For each experiment, we report the total time to solve all 1061 queries (*Total*), number of unsolved cases (*Failed*), average time per a solved instance (*Avg. per Solved*), total time for all solved instances (*Total Solved*), total time for all instances solved by MATHSAT4 (*Total MSAT Solved*), and total time for all instances solved by MATHSAT4 with predicates in each query restricted to those that appear in the support of the solution computed by LDD (*Total MSAT2 Solved*). All "Total" times include 60s for each failure.

Surprisingly, the AllSAT-based approach is the worst. It was only able to solve 60% of queries and was 7 times slower compared to the LDD-based solutions. Even restricted to queries that it could solve, it is almost 4 times slower than LDD, and 9 times slower than LDD2. Fig. 3a shows a detailed comparison between the MSAT and LDD experiments. In the chart, test case indices are on the x-axis (sorted first by MSAT time, and then by LDD time), time per test case is on the y-axis. There are several exceptional cases where MATHSAT4 significantly outperforms LDD. However, overall, most test-cases appear to be easy for LDD (solved in under 5s), but are much more evenly distributed for MATHSAT4.

Name	Total *(min)*	Failed	Avg. per Solved *(sec)*	Total Solved *(min)*	Total MSAT Solved *(min)*	Total MSAT2 Solved *(min)*
MSAT	610.00	429	17.12	180.29	180.29	523.86
LDD	83.54	35	2.84	48.48	60.64	72.15
LDD2	83.98	64	1.19	19.81	44.34	72.79
MIN	**28.40**	**6**	1.27	22.39	10.64	**19.98**
LDD3	85.66	74	**0.70**	**11.51**	57.64	77.08
MSAT2	188.14	91	6.87	102.00	**9.04**	102.00

(a) PAQ benchmark

Name	LBE						SBE			
	T			It	Pr	CP	T	It	Pr	BB
	LDD	MSAT	MIN							
`floppy.ok`	0.18	0.16	0.16	1	0	3	0.44	4	6	83
`tst_lck_50`	0.5	0.48	0.5	1	0	3	++	++	++	255
`diamond-4`	2.0	++	1.7s	4	42	4	++	++	++	24
`ssl-srv-D`	98.96	6.26	5.65	5	60	4	++	++	++	155

(b) End-to-end. T = times in sec; It = # of CEGAR iterations; CP = # of cutpoints; BB = # of blocks; Pr = total # of preds.

Fig. 4. Summary of experimental results

The two LDD-based experiments clearly highlight the virtues and vices of DVO: DVO makes an LDD-approach more robust (35 failures for LDD v.s. 64 for LDD2) but less efficient (about twice as slow on average). Out of 64 failures for LDD2, 39 where due to memory running out. Coincidentally, with our choice of using 60s for each failure, faster running times balance out more failures for LDD2, and its overall time is very similar to that of LDD.

In our benchmark, LDD-based solution significantly outperforms the AllSAT-based one. We conjecture that the two are complementary: AllSAT-based solution performs well when number of models to enumerate is small, and LDD-based solution performs well when the intermediate (and final) diagrams are small. To validate this conjecture, we computed the best-of time needed to solve a test-case by either of the three techniques. This is equivalent to running the three approaches in parallel and stopping as soon as one was successful. The results are summarized in the fourth row (MIN) of Table 4a. The combination is extremely effective: taking only 28 minutes (3 times better than previous best) for the benchmark and solving all but 6 instances. The improvement is even more significant when restricted to instances that MATHSAT4 could solve.

Oracle experiments. To put our results into perspective, we conducted two experiments against "oracle" solvers. Finding good variable ordering is the bottleneck for LDD-based solution. With DVO most time is spend reordering, but without it many cases run out of memory. We experimented with using the last ordering found by DVO during LDD experiment as a static ordering. The results are shown in the fifth row (LDD3) of Table 4a. We classify this experiment as "oracle" since we don't know how to achieve this variable order other than by repeating the LDD experiment.

Interestingly, the order did not help as much as we expected. The average time per solved test-case did drop to 0.7s (2× and 4× better than LDD2 and LDD, respectively). However, fewer instances could be solved, with 55 out of the 74 failures being memory outs. We believe this indicates that an order that is good for the final result is not necessarily good for the intermediate steps (and hence, overall) of the computation.

The bottleneck for the AllSAT-based solution is in enumerating all the minterms (as opposed to cubes or prime implicants). We found that in many cases that were hard for MATHSAT4, many of the predicates did not appear in

the support of the LDD-based solution. That is, many predicates were not part of any prime implicant. To evaluate the effect of this on MATHSAT4, we repeated the MSAT experiment, but restricted the predicates in each query to those that appeared in the support of the solution (as computed by LDD). The results are shown in the last row (MSAT2) of Table 4a. Note that determining variables in the support of a Boolean formula is NP-complete. We do not know how to compute the support other than by solving the problem with LDDs first.

Overall, the running time has improved dramatically. There is a significant improvement on the cases solved by MSAT, even compared to LDD-based solutions. However, overall it is much slower than any of the LDD-based solutions, even when restricted to cases it could solve. Overall, there are 91 failures (all timeouts). Fig. 3b shows the details from the MSAT2 and MIN experiments. There are two interesting points. First, the best-of LDD and MSAT is significantly better than the idealized AllSAT-based solution. Second, there are cases where the idealized AllSAT-based solution is an order-of-magnitude better.

End-to-end experiments. To evaluate the end-to-end performance of our approach, we implemented a CEGAR-based safety checker for C programs following Jhala et al. [11]. Fig. 4b is a sample of our results: `floppy.ok` is derived from a device driver, `test_locks_50` is based on the example from Beyer et al. [2], `diamond-4` is a program with a "diamond-shaped" CFG, and `ssl-srv-D` is derived from OpenSSL. We observe that LBE scales much better than SBE. The performances of LDD and AllSAT are more evenly balanced. LDD scales better for diamond-4. For ssl-srvr-D, LDD by itself is much worse than AllSAT. This is due to a single PAQ that is very hard for LDD. However, LDD outperforms AllSAT elsewhere, as seen by the MIN column.

Summary. Overall, our results show that the AllSAT-based solution is not competitive for solving PAQs of a large program fragment, while the LDD-based solution performs surprisingly well. Moreover, the MIN of the LDD- and the AllSAT-based approaches is the clear winner, even compared to an oracle-based solution.

6 Conclusion

Large-Block Encoding (LBE) [2] is a flavor of predicate abstraction applied to a summarized program. In this paper, we present solutions to three problems for predicate abstraction in the context of LBE. First, we define a general notion of program summarization, called *a loop cutset summary*, that is compatible with LBE and is efficiently computable. We show that it generalizes the rule-based summary of Beyer et al. [2]. Second, we present a linear time algorithm to construct PAQs for a loop-free program fragment. Our algorithm works directly on the SSA representation of the program, and constructs a query that separates control- and data-flow, while preserving both in its structure. Third, we study two approaches to solving PAQs: a classical AllSAT-based, and a new based on LDDs. The approaches are evaluated on a benchmark from open-source software.

Our approach builds on many existing components: SSA, loop-free program fragments, and early quantification – all well known; LDDs are used in [5] for

image computation. However, the combination of the techniques is novel, the benchmarks are realistic and challenging, and the results show that our new LDD-based solution outperforms (and complements) the AllSAT-based one.

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
2. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software Model Checking via Large-Block Encoding. In: FMCAD 2009 (2009)
3. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT4 SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123. Springer, Heidelberg (2008)
4. Cavada, R., Cimatti, A., Franzén, A., Kalyanasundaram, K., Roveri, M., Shyamasundar, R.K.: Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In: FMCAD 2007 (2007)
5. Chaki, S., Gurfinkel, A., Strichman, O.: Decision Diagrams for Linear Arithmetic. In: FMCAD 2009 (2009)
6. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate Abstraction of ANSI-C Programs using SAT. FMSD 25(2-3) (2004)
7. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. TOPLAS 13(4) (1991)
9. Das, S., Dill, D.: Successive Approximation of Abstract Transition Relations. In: LICS 2001, pp. 51–60 (2001)
10. Flanagan, C., Qadeer, S.: Predicate Abstraction for Software Verification. In: POPL 2002, pp. 58–70 (2002)
11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions From Proofs. In: POPL 2004 (2004)
12. Karp, R.M.: Reducibility Among Combinatorial Problems. In: Complexity of Computer Computations, pp. 85–103 (1972)
13. Kroening, D., Sharygina, N.: Approximating Predicate Images for Bit-Vector Logic. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 242–256. Springer, Heidelberg (2006)
14. Lahiri, S.K., Ball, T., Cook, B.: Predicate Abstraction via Symbolic Decision Procedures. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 24–38. Springer, Heidelberg (2005)
15. Lahiri, S.K., Bryant, R.E., Cook, B.: A Symbolic Approach to Predicate Abstraction. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 141–153. Springer, Heidelberg (2003)
16. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT Techniques for Fast Predicate Abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 424–437. Springer, Heidelberg (2006)
17. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO 2004 (2004)