BDD-Based Software Model Checking with CPAchecker

Dirk Beyer and Andreas Stahlbauer

University of Passau, Germany

Abstract. In symbolic software model checking, most approaches use predicates as symbolic representation of the state space, and SMT solvers for computations on the state space; BDDs are sometimes used as auxiliary data structure. The representation of software state spaces by BDDs was not yet thoroughly investigated, although BDDs are successful in hardware verification. The reason for this is that BDDs do not efficiently support all operations that are needed in software verification. In this work, we evaluate the use of a pure BDD representation of integer variable values, and focus on a particular class of programs: event-condition-action systems with limited operations. A symbolic representation using BDDs seems appropriate for this particular class of programs. We implement a program analysis based on BDDs and experimentally compare three symbolic techniques to verify reachability properties of ECA programs. The results show that BDDs are efficient, which yields the insight that BDDs could be used selectively for some variables (to be determined by a pre-analysis), even in general software model checking.

1 Introduction

The internal representation of sets of reachable abstract states is an important factor for the effectiveness and efficiency of software model checking. Binary decision diagrams (BDD) [10] are an efficient data structure for manipulation of large sets, because they represent the sets in a compressed representation, and operations are performed directly on the compressed representation. BDDs are used, for example, to store the state sets in tools for hardware verification [11, 12], for transition systems in general [16], for real-time systems [8, 13], and push-down systems [14]. There are programming systems for relational programming [2] based on BDDs, and the data structure is used for points-to program analyses [1]. The current state-of-the-art approaches to software verification [3] are either based on satisfiability (SAT) and SAT-modulo-theories (SMT) solving, or on abstract domains from data-flow analysis. BDDs were so far not used as main representation for the state space of integer variables (only as auxiliary data structure). For example, software verifiers based on predicate analysis [4,6] use BDDs for storing truth values of predicates. There exists a version of Java PathFinder that supports the annotation of boolean variables in the program such that the analyzer can track the specified boolean variables using BDDs, which was shown to be efficient for the verification of software product lines [19].

A. Kučera et al. (Eds.): MEMICS 2012, LNCS 7721, pp. 1–11, 2013.
© Springer-Verlag Berlin Heidelberg 2013

This paper applies BDDs as representation of state sets in the verification of C programs, with a focus on event-condition-action (ECA) systems that use a very limited set of operations. Such ECA programs were used as benchmarks in a recent verification challenge [15] [1]. For such a special sub-class of ECA programs, BDDs seem to be promising as representation for two reasons. First, the programs that we consider consist of a single loop in which many conditional branches occur. In each of those branches, a condition is a boolean combination of equalities and negated equalities between variables and values, and an action is a sequence of assignments of values to variables. This means that all required operations are in fact efficiently supported by BDDs, and a symbolic representation using BDDs seems indeed appropriate for this particular class of programs. Second, due to the complex control and data flow of these programs, they are challenging verification tasks for traditional techniques. The formulas that are used as representation in predicate-based approaches represent many paths with a complicated control structure, which are sometimes overwhelming for the SMT solver.

Contribution. We implement a configurable program analysis (CPA) based on BDDs and experimentally compare three symbolic techniques to verify reachability properties of ECA programs. The contribution of this work is not to use BDDs for software verification (which was done before, e.g., in Moped [14]), but to experimentally show that using BDDs as representation for certain variables (which are used in a restricted way) can be more efficient than other (more expressive, but also more expensive) encodings. The insight is that it could be a promising approach to software verification to analyze the usage of each variable in a pre-analysis and then determine for each variable the most efficient representation based on the result.

2 Preliminaries

In order to define a verifier, we need an iteration algorithm and a configurable program analysis, which defines the abstract domain, the transfer relation, as well as the merge and stop operators. In the following, we provide the definitions of the used concepts and notions from previous work [5].

Programs. We consider only a simple imperative programming language, in which all operations are either assignments or assume operations, and all variables are of type integer.[2] We represent a *program* by a *control-flow automaton* (CFA), which consists of a set L of program locations (models the program counter pc), an initial program location pc_0 (models the program entry), and a set $G \subseteq L \times Ops \times L$ of control-flow edges (models the operation that is executed when control flows from one program location to another). The set X of program

[1] `http://leo.cs.tu-dortmund.de:8100/isola2012/`

[2] The framework CPAchecker [6], which we use to implement the analysis, accepts C programs and transforms them into a side-effect free form [18]; it also supports interprocedural program analysis.

variables contains all variables that occur in operations from *Ops*. A *concrete state* of a program is a variable assignment $c : X \cup \{pc\} \to \mathbb{Z}$ that assigns to each variable an integer value. The set of all concrete states of a program is denoted by C. A set $r \subseteq C$ of concrete states is called a *region*. Each edge $g \in G$ defines a (labeled) transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The complete transition relation $\to$ is the union over all control-flow edges: $\to = \bigcup_{g \in G} \xrightarrow{g}$. We write $c \xrightarrow{g} c'$ if $(c, g, c') \in \to$, and $c \to c'$ if there exists a g with $c \xrightarrow{g} c'$. A concrete state c_n is *reachable* from a region r, denoted by $c_n \in Reach(r)$, if there exists a sequence of concrete states $\langle c_0, c_1, \ldots, c_n \rangle$ such that $c_0 \in r$ and for all $1 \leq i \leq n$, we have $c_{i-1} \to c_i$. Such a sequence is called *feasible program path*. In order to define an efficient program analysis, we need to define abstract states and abstract transitions.

Configurable Program Analysis. We use the framework of *configurable program analysis* (CPA) [5] to formalize our program analysis. A CPA specifies the abstract domain and a set of operations that control the program analysis. A CPA is defined independently of the analysis algorithm, and can be plugged in as a component into the software-verification framework without working on program parsers, exploration algorithms, and other general data structures. A CPA $\mathbb{C} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$ (which specifies how to compute abstract successor states), a merge operator merge (which defines how to merge abstract states when control flow meets), and a stop operator stop (which indicates if an abstract state is covered by another abstract state). The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete states, a semi-lattice $\mathcal{E}$ over abstract-domain elements, and a concretization function that maps each abstract-domain element to the represented set of concrete states. The abstract-domain elements are also called *abstract states*.

Using this framework, program analyses can be composed of several component CPAs. We will now give the definition of a location analysis; our complete analysis will be the composition of the location analysis with the BDD-based analysis that we will define later.

CPA for Location Analysis. The CPA for *location analysis* $\mathbb{L} = (D_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, \mathsf{merge}_{\mathbb{L}}, \mathsf{stop}_{\mathbb{L}})$ tracks the program counter *pc* explicitly [5].

1. The domain $D_{\mathbb{L}}$ is based on the flat semi-lattice for the set L of program locations: $D_{\mathbb{L}} = (C, \mathcal{E}_{\mathbb{L}}, [\![\cdot]\!])$, with $\mathcal{E}_{\mathbb{L}} = ((L \cup \{\top\}), \sqsubseteq)$, $l \sqsubseteq l'$ if $l = l'$ or $l' = \top$, $[\![\top]\!] = C$, and for all l in L, $[\![l]\!] = \{c \in C \mid c(pc) = l\}$.

2. The transfer relation $\rightsquigarrow_{\mathbb{L}}$ has the transfer $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$ if $g = (l, \cdot, l')$.

3. The merge operator does not combine elements when control flow meets: $\mathsf{merge}_{\mathbb{L}}(l, l') = l'$.

4. The termination check returns true if the current element is already in the reached set: $\mathsf{stop}_{\mathbb{L}}(l, R) = (l \in R)$.

Analysis Algorithm. Algorithm 1 shows the core iteration algorithm that is used to run a configurable program analysis, as implemented by tools like CPAchecker. The algorithm is started with a CPA and two sets of abstract

Algorithm 1. *CPA*$(\mathbb{D}, R_0, W_0)$ (taken from [5])

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$,
a set $R_0 \subseteq E$ of abstract states,
a subset $W_0 \subseteq R_0$ of frontier abstract states,
where E denotes the set of elements of the semi-lattice of D
Output: a set of reachable abstract states,
a subset of frontier abstract states
Variables: two sets reached and waitlist of elements of E

```
 1: reached := R_0;
 2: waitlist := W_0;
 3: while waitlist ≠ ∅ do
 4:    choose e from waitlist; remove e from waitlist;
 5:    for each e' with e⇝e' do
 6:       for each e'' ∈ reached do
 7:          // Combine with existing abstract state.
 8:          e_new := merge(e', e'');
 9:          if e_new ≠ e'' then
10:             waitlist := (waitlist ∪ {e_new}) \ {e''};
11:             reached := (reached ∪ {e_new}) \ {e''};
12:       // Add new abstract state?
13:       if ¬ stop(e', reached) then
14:          waitlist := waitlist ∪ {e'};
15:          reached := reached ∪ {e'};
16: return (reached, waitlist)
```

states as input: the set R_0 (reached) contains the so far reached abstract states, and the set W_0 (waitlist) contains abstract states that the algorithm needs to process. The algorithm terminates if the set waitlist is empty (i.e., all abstract states are processed) and returns the two sets reached and waitlist. We start the algorithm with two singleton sets that contain only the initial abstract state. In each iteration of the 'while' loop, the algorithm processes and removes one state e from the waitlist, by computing all abstract successors and further processing them as e'.

Next, the algorithm checks (lines 6–11) if there is an existing abstract state in the set of reached states with which the new state e' has to be merged (e.g., where control flow meets after completed branching). If this is the case, then the new, merged abstract state is substituted for the existing abstract state in both sets reached and waitlist. (This operation is sound because the merge operation is not allowed to under-approximate.) In lines 12–15, the stop operator checks if the new abstract state is covered by a state that is already in the set reached, and inserts the new abstract state into the work sets only if it is not covered.

Binary Decision Diagrams. A binary decision diagram (BDD) [10] represents a set of assignments for a set of boolean variables. In our analysis, we need to consider integer variables. We encode the integer assignments as bit vectors, and the integer variables as vectors of boolean variables, and thus, can represent data states of integer programs by BDDs.

A BDD is a rooted directed acyclic graph, which consists of decision nodes and two terminal nodes (called 0-terminal and 1-terminal). Each decision node is labeled by a boolean variable and has two children (called low child and high child). A BDD is maximally reduced according to the following two rules: (1) merge any isomorphic sub-graphs, and (2) eliminate any node whose two children are isomorphic. Every variable assignment that is represented by a BDD corresponds to a path from the root node to the 1-terminal. The variable of a node has the value 0 if the path follows the edge to the low child, and the value 1 if it follows the edge to the high child. A BDD is always ordered, which means that the variables occur in the same order on any path from the root to a terminal node. For a given variable order, the BDD representation of a set of variable assignments is unique.

3 BDD-Based Program Analysis

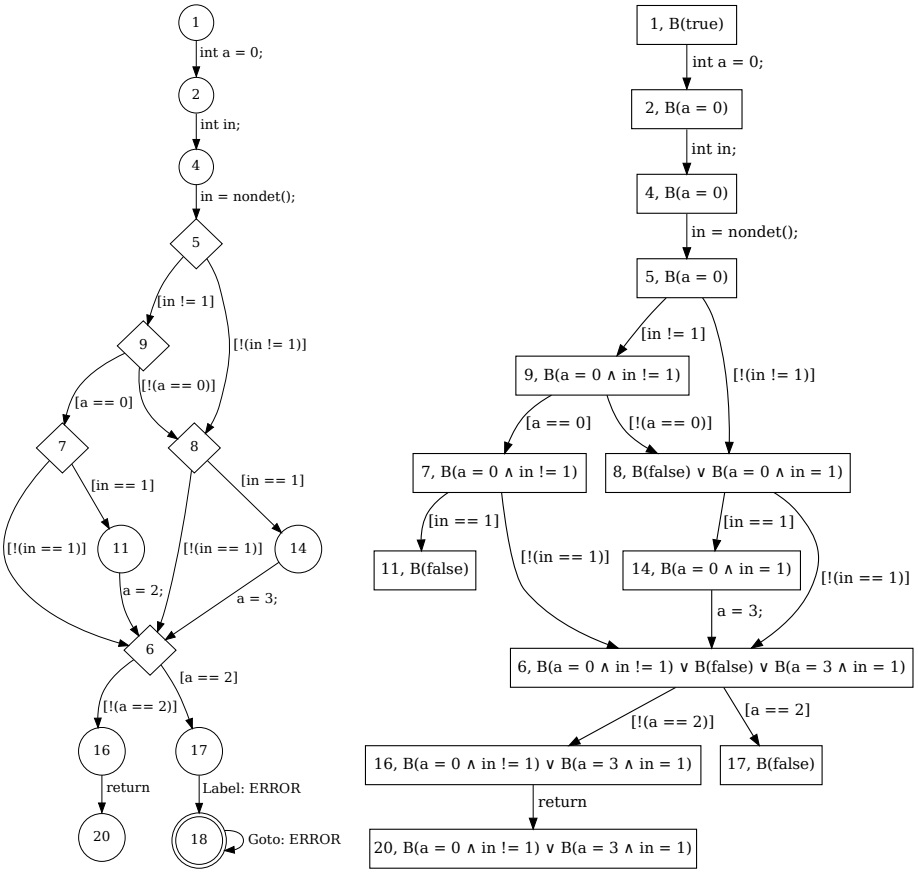
For implementing the BDD-based analysis, we define a configurable program analysis (CPA) that uses BDDs to represent abstract states, and implement it in the open-source tool CPAchecker.

Let X be the set of program variables. Given a first-order formula φ over X, we use $\mathbb{B}_\varphi$ to denote the BDD that is constructed from φ, and $[\![\varphi]\!]$ to denote all variable assignments that fulfill φ. Given a BDD $\mathbb{B}$ over X, we use $[\![\mathbb{B}]\!]$ to denote all variable assignments that $\mathbb{B}$ represents ($[\![\mathbb{B}_\varphi]\!] = [\![\varphi]\!]$).

The *BDD-based program analysis* is a configurable program analysis $\mathbb{BPA} = (D_{\mathbb{BPA}}, \rightsquigarrow_{\mathbb{BPA}}, \mathsf{merge}_{\mathbb{BPA}}, \mathsf{stop}_{\mathbb{BPA}})$ that represents the data states of the program symbolically, by storing the values of variables in BDDs. The CPA consists of the following components:

1. The abstract domain $D_{\mathbb{BPA}} = (C, \mathcal{E}, [\![\cdot]\!])$ is based on the semi-lattice $\mathcal{E}_\mathbb{B}$ of BDDs, i.e., every abstract state consists of a BDD. The concretization function $[\![\cdot]\!]$ assigns to an abstract state $\mathbb{B}$ the set $[\![\mathbb{B}]\!]$ of all concrete states that are represented by the BDD. Formally, the lattice $\mathcal{E}_\mathbb{B} = (\mathcal{B}, \sqsubseteq)$ —where $\mathcal{B}$ is the set of all BDDs, $\mathbb{B}_{true}$ is the BDD that represents all concrete states (1-terminal node), and $\mathbb{B}_{false}$ is the BDD that represents no concrete state (0-terminal node)— is induced by the partial order $\sqsubseteq$ that is defined as: $\mathbb{B} \sqsubseteq \mathbb{B}'$ if $[\![\mathbb{B}]\!] \subseteq [\![\mathbb{B}']\!]$. (The join operator $\sqcup$ yields the least upper bound; $\mathbb{B}_{true}$ is the top element $\top$ of the semi-lattice.)
2. The transfer relation $\rightsquigarrow_{\mathbb{BPA}}$ has the transfer $\mathbb{B} \overset{g}{\rightsquigarrow} \mathbb{B}'$ with $\mathbb{B}' = \begin{cases} \mathbb{B} \wedge \mathbb{B}_p & \text{if } g = (l, \mathtt{assume}(p), l') \\ (\exists w : \mathbb{B}) \wedge \mathbb{B}_{w=e} & \text{if } g = (l, \mathtt{w} := \mathtt{e}, l') \end{cases}$.
3. The merge operator is defined by $\mathsf{merge}_{\mathbb{BPA}}(\mathbb{B}, \mathbb{B}') = \mathbb{B} \vee \mathbb{B}'$.
4. The termination check is defined by $\mathsf{stop}_{\mathbb{BPA}}(\mathbb{B}, R) = \exists \mathbb{B}' \in R : \mathbb{B} \sqsubseteq \mathbb{B}'$.

We construct the complete program analysis by composing the CPA $\mathbb{BPA}$ for BDD-based analysis with the CPA $\mathbb{L}$ for location analysis, in order to also track the program locations. For further details on CPA composition, we refer to [5].



(a) Control-flow autom. (CFA) (b) Abstract reachability graph (ARG)

Fig. 1. Example program with verification certificate

Example. Consider the program represented by the control-flow automaton (CFA) in Fig. 1(a). The error location (location 18, indicated by label 'ERROR') is not reachable in this simple example program, i.e., the program is safe. Figure 1(b) represents the corresponding abstract-reachability graph (ARG), which could serve as verification certificate for this analysis. The nodes in the ARG represent abstract states, which are initial abstract states or constructed by computing abstract successor states according to the edges of the CFA, using the CPA algorithm and composition of CPAs as described above. The edges in the ARG represent successor computations along the control-flow edges of the corresponding CFA. We label each node of the ARG with the program location and the BDD that represents the abstract data state. The set of states that are represented by the nodes of the ARG shown in Fig. 1(b) equals the set reached after the CPA algorithm has terminated.

The analysis starts at the initial program location $pc_0 = 1$ with the initial abstract data state e_0, which is represented by the BDD $\mathbb{B}_{true}$. The analysis then computes the abstract successor states by applying the transfer relation $\rightsquigarrow$; in our example the abstract data state for location $pc = 2$ is computed by quantifying the assigned variable in the BDD of the previous abstract state, create a BDD for the constraint of control-flow edge `int a=0` (assignment) and conjunct it with the former BDD. The transition along the edge $(2, \texttt{int in}, 4)$ does not change the abstract data state because the variable that is declared by this edge was not known before; also the transition along $(4, \texttt{in = nondet()}, 4)$ does not change the data state because it does not restrict the possible concrete states (the return value of `nondet()` is non-deterministic). Transitions whose operations are assumptions, for example, $(5, \texttt{[in != 1]}, 9)$ are encoded by conjuncting the BDD $\mathbb{B}$ of the abstract data state of the predecessor location ($pc = 5$) with the BDD for the respective assumption (`in != 1`), i.e., the successor state $\mathbb{B}'$ is computed as $\mathbb{B}' = \mathbb{B}_{a=0} \wedge \mathbb{B}_{in\,!=1}$. Now we consider, for example, location $pc = 14$, which has the BDD $\mathbb{B}_{a=0 \wedge in=1}$ as abstract data state, and process the control-flow edge $(14, \texttt{a = 3}, 6)$ (assignment). Assignment operations are processed by first existential quantifying the variable that gets a new value assigned (`a`); then the intermediate BDD $\mathbb{B}_{in=1}$ is conjuncted with the BDD that represents the new value of the variable: $\mathbb{B}' = \mathbb{B}_{in=1} \wedge \mathbb{B}_{a=3}$.

Abstract states that were computed for the same program location are —as defined by the CPA operator merge— joined by computing the disjunction of the BDDs; the abstract data state $\mathbb{B}_{(a=0 \wedge in\,!=1) \vee (false) \vee (a=3 \wedge in=1)}$ at location $pc = 6$ is such a result of a join. After the analysis has terminated, the set reached of reached states contains at most one abstract state for each program location.

The computation of successors of a given abstract state e stops (the abstract state is not added to the sets waitlist and reached for further processing), whenever the abstract data state is already covered by (implies) an existing abstract data state; this check is performed by the CPA operator stop. The analysis does not process successors of locations 11 and 17, because the BDDs evaluate to *false*. Thus, the error location 18 is not reached.

4 Evaluation

In order to demonstrate that the BDD-based analysis yields a significant performance improvement on a set of C programs with restricted operations on integer variables, we compare our simple analysis with two other approaches for symbolic software model checking.

Experimental Setup. All experiments were performed on machines with a 3.4 GHz Quad Core CPU and 16 GB of RAM. The operating system was Ubuntu 12.04 (64 bit), using Linux 3.2.0-30 and OpenJDK 1.6.0_24. A time limit of 5 min and a memory limit of 15 GB were used. We took CPAchecker from revision 6607 of the trunk in the repository, and MathSAT 4.2.17 as SMT solver; for the BDD-based analysis we configured it with a Java heap size of 13 GB, for the other analyses we configured it with 10 GB, in order to leave RAM for the SMT solver.

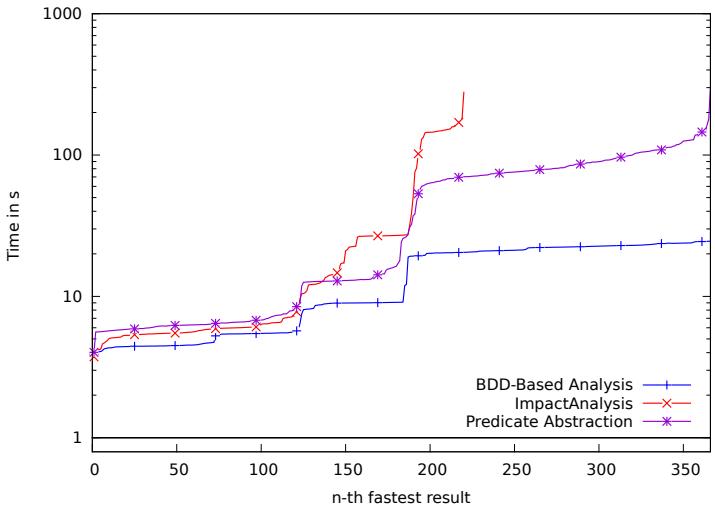


Fig. 2. Quantile functions for the three different approaches

The configuration of the BDD-based analysis is specified in the configuration file `fsmBddAnalysis.properties`.

Verification Tasks. For the evaluation of our approach, we use Problems 1 to 6 from the recent RERS challenge, because those programs are in the restricted class of C programs that we described earlier. Tables with detailed results and the benchmark programs are publicly available on the accompanying web page at `http://www.sosy-lab.org/~dbeyer/cpa-bdd`.

Compared Verification Approaches. We restrict the comparison to three symbolic techniques that are all implemented in the same verification tool, in order to eliminate influence of the used solver, libraries, parser front-ends, etc. The first approach is an Impact-based analysis [17]. This analysis is based on counterexample-guided abstraction refinement (CEGAR) and computes abstractions using interpolation along infeasible error paths. In contrast to predicate abstraction, this analysis does not compute strongest post-conditions and abstracts those to more abstract formulas, but uses a conjunction of the obtained interpolants as abstract states. A detailed comparison of the approach with predicate abstraction can be found in the literature [9]. The second approach is based on CEGAR and predicate abstraction, together with adjustable-block encoding [7]. The third approach is the BDD-based analysis that was introduced in this paper.

Discussion. Figure 2 gives an overview over the results using a quantile plot of the verification times (all verification tasks, no separation between satisfied and violated properties). A quantile plot orders, for each approach separately, the verification runs by the run time that was needed to obtain the correct verification result on the x-axis (n-th fastest result). A data point (x, y) of the graph means that x verification tasks were successfully verified each in under y

Table 1. Detailed results for the verification tasks with result 'UNSAFE'

		Impact Algorithm			Predicate Abstraction			BDD-Based Analysis		
Problem	# Properties	Solved properties	Time (total)	Time (mean)	Solved properties	Time (total)	Time (mean)	Solved properties	Time (total)	Time (mean)
Problem 1	14	14	86	6.2	14	100	7.4	14	65	4.6
Problem 2	8	8	37	4.7	8	48	6.0	8	33	4.2
Problem 3	14	10	120	12	14	190	13	14	110	8.2
Problem 4	25	1	14	14	25	2600	100	25	490	20
Problem 5	25	25	3600	150	25	3200	130	25	520	21
Problem 6	26	2	100	52	26	2200	85	26	520	20

Table 2. Detailed results for the verification tasks with result 'SAFE'

		Impact Algorithm			Predicate Abstraction			BDD-Based Analysis		
Problem	# Properties	Solved properties	Time (total)	Time (mean)	Solved properties	Time (total)	Time (mean)	Solved properties	Time (total)	Time (mean)
Problem 1	47	47	270	5.8	47	310	6.7	47	260	5.5
Problem 2	53	53	320	6.0	53	320	6.0	53	240	4.5
Problem 3	47	17	230	14	47	660	14	47	420	9.0
Problem 4	36	36	950	27	36	2500	70	36	820	24
Problem 5	36	6	790	130	36	2700	75	36	800	22
Problem 6	35	1	51	51	35	2800	80	35	840	24

seconds of CPU time. The integral below the graph illustrates the accumulated verification time for all solved verification tasks. The Impact-based analysis is not able to solve all verification tasks (it solves 220 instances), the predicate-abstraction-based analysis can verify each property within 300 s of CPU time. The BDD-based analysis, which we introduced earlier in this paper, is able to solve each of the properties within 25 s.

Table 1 shows more detailed results for the violated properties, i.e., the verification tasks for which the verification result is 'UNSAFE', and Table 2 shows the details for the satisfied properties. The verification time (total and mean) values are given in seconds of CPU time with two significant digits. The Impact-based analysis can solve the verification tasks of the programs Problem 1 and Problem 2

completely; performance and precision decrease dramatically for Problems 3 to 6. The predicate-abstraction-based analysis and the BDD-based analysis can both verify all properties; but the BDD-based analysis is significantly more efficient. The BDD-based analysis scales best with the problem size (assuming that the verification tasks for the program 'Problem $n+1$' are harder than the tasks for the program 'Problem n').

5 Conclusion

We extended the standard software-verification tool CPAchecker by a configurable program analysis (CPA) that uses BDDs as data structure to represent sets of data states (variable assignments). We have compared the effectiveness and efficiency of this analysis to other approaches that use symbolic techniques: a program analysis that computes abstract successor states using predicate abstraction after every successor computation [7], and a program analysis that computes abstract states along paths using interpolation [9, 17] — both being state-of-the-art approaches for symbolic software verification.

The experiments show that the BDD-based approach is the most efficient verification approach (by an order of magnitude) for the considered class of programs. However, as soon as the programs use more general operations, BDDs would be prohibitively less efficient. This means that BDDs can be more efficient than other representations for certain types of variables (the ones that are involved in simple operations only). This is an important insight and motivation for future work: It would be promising to pre-analyze the program in order to find out for each variable how it is used, and then determine —based on its usage-type— the most efficient abstract domain to track this variable. The other variables can be analyzed by an explicit-value analysis or a predicate-analysis; using a configuration program analysis (CPA) with adjustable precisions; such combinations of program analyses are easy to construct in CPAchecker.

References

1. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to Analysis using BDDs. In: Proc. PLDI, pp. 103–114. ACM (2003)
2. Beyer, D.: Relational Programming with CrocoPat. In: Proc. ICSE, pp. 807–810. ACM (2006)
3. Beyer, D.: Competition on Software Verification (SV-COMP). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker Blast. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)
5. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)

6. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
7. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: Proc. FMCAD, pp. 189–197 (2010)
8. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: A Tool for BDD-Based Verification of Real-Time Systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 122–125. Springer, Heidelberg (2003)
9. Beyer, D., Wendler, P.: Algorithms for Software Model Checking: Predicate Abstraction vs. IMPACT. In: Proc. FMCAD (2012)
10. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
11. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L.: Sequential Circuit Verification using Symbolic Model Checking. In: Proc. DAC, pp. 46–51. ACM (1990)
12. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. In: Proc. LICS, pp. 428–439. IEEE (1990)
13. Campos, S.V.A., Clarke, E.M.: The Verus Language: Representing Time Efficiently with BDDs. In: Rus, T., Bertrán, M. (eds.) ARTS 1997. LNCS, vol. 1231, pp. 64–78. Springer, Heidelberg (1997)
14. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 489–503. Springer, Heidelberg (2006)
15. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS Grey-Box Challenge 2012: Analysis of Event-Condition-Action Systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 608–614. Springer, Heidelberg (2012)
16. McMillan, K.L.: The SMV System. Technical Report CMU-CS-92-131, Carnegie Mellon University (1992)
17. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
18. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
19. von Rhein, A., Apel, S., Raimondi, F.: Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In: Proc. Java Pathfinder Workshop (2011)