Int J Softw Tools Technol Transfer (2014) 16:507–518
DOI 10.1007/s10009-014-0334-1

RERS

BDD-based software verification

Applications to event-condition-action systems

Dirk Beyer · Andreas Stahlbauer

Published online: 19 August 2014

Abstract In software model checking, most successful symbolic approaches use predicates as representation of the state space, and SMT solvers for computations on the state space; BDDs are often used as auxiliary data structure. Although BDDs are applied with great success in hardware verification, BDD representations of software state spaces were not yet thoroughly investigated, mainly because not all operations that are needed in software verification are efficiently supported by BDDs. We evaluate the use of a pure BDD representation of integer values, and focus on a particular class of programs: event-condition-action (ECA) programs with limited operations. A symbolic representation using BDDs seems appropriate for ECA programs under certain conditions. We configure a program analysis based on BDDs and experimentally compare it to four approaches to verify reachability properties of ECA programs: an explicit-value analysis, a symbolic bounded-loops analysis, a predicate-abstraction analysis, and a predicate-impact analysis. The results show that BDDs are efficient for a restricted class of programs, which yields the insight that BDDs could be used selectively for variables that are restricted to certain program operations (according to the variable's domain type), even in general software model checking. We show that even a naive portfolio approach, in which after a pre-analysis either a BDD-based analysis or a predicate-impact analysis is performed, outperforms all above-mentioned analyses.

Keywords Binary decision diagram · BDD · Symbolic model checking · Software model checking · Program analysis · Event-condition-action system · RERS challenge · Configurable software verification · CPA

An earlier version was published in Proc. MEMICS'12 [18].

D. Beyer (✉) · A. Stahlbauer
University of Passau, Passau, Germany

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) [24] are an efficient data structure for manipulating large sets, because they store 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 [26,27], for transition systems in general [21,43], for real-time systems [15,23,28], and push-down systems [36]. There are systems for relational programming [6] based on BDDs, and the data structure is used for points-to program analyses [4].

In this paper, we use BDDs as representation of state sets in the verification of C programs, with a focus on event-condition-action (ECA) systems that use a limited set of operations. Such ECA programs were used as benchmarks in recent verification challenges [39,40].[1] For a special subclass of ECA programs, BDDs seem to be a promising representation of state sets 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, the condition is a boolean combination of equalities and negated equalities between variables and values, and the action is a sequence of assignments 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. Sec-

[1] http://rers-challenge.org/.

ond, due to the complex control and data flow of these ECA 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 sometimes overwhelm the underlying SMT-solvers [14].

CPACHECKER [2] [13], an open-source framework for configurable software verification, provides several configurable program analyses, including an explicit-value analysis, a symbolic bounded-loops analysis, a predicate-abstraction analysis, a predicate-impact analysis, and a BDD analysis. We present the results of applying one enumerative analysis and several symbolic analyses to the verification of reachability properties of the ECA programs. We configure CPACHECKER in order to evaluate (1) an explicit-value analysis (enumerative, explicit-state model checking [16]), (2) a bounded-loop analysis (symbolic, bounded model checking, cf. [20]), (3) a predicate-abstraction analysis (lazy abstraction [38], Craig interpolation for predicate extraction [37], boolean abstraction [3], adjustable-block encoding [9,14]), (4) a predicate-impact analysis (interpolation instead of predicate abstraction [19,44]), and (5) a program analysis based on BDDs (integer variables encoded as BDDs [18], as described in Sect. 3).

1.1 Related work

The current state-of-the-art approaches to software verification [7,8] are either based on satisfiability (SAT) and SAT-modulo-theories (SMT) solving, or on abstract domains from data-flow analysis. BDD-based approaches were so far not thoroughly evaluated in this context as main representation for the state space of integer variables in C programs (only as auxiliary data structure). In preliminary comparisons, SAT-based approaches often outperformed BDD-based approaches [41].

1.1.1 Models of software

Binary decision diagrams-based model checking is a standard technique for verifying transition systems, and several state-of-the-art verifiers are available. SMV [43] is a tool for checking properties (temporal logic, CTL) of finite-state systems. SMV was developed to verify hardware designs and has later been applied to software as well [1]. SMV uses a BDD-based symbolic model-checking algorithm, in which the state graph of the model is represented using BDDs. NUSMV [30] is an alternative implementation of SMV and additionally supports LTL model checking. NUSMV 2 [29] integrates BDD-based model checking and SAT-based model checking. The IMPROVISO algorithm [42], which was implemented within NUSMV, uses BDDs and an efficient partial order reduction. LTSMIN [21] is a tool that uses BDDs to check safety properties of transition systems provided in high-level languages, such as Promela. RABBIT [15] is a BDD-based verifier for modular timed automata [5], and was used to model and verify controller software for real-time systems.

1.1.2 Auxiliary use

Several software verifiers that are based on predicate abstraction [10,13] use BDDs for storing truth values of predicates. Each predicate in the precision of the predicate-abstraction analysis corresponds to one propositional variable. In cartesian and boolean predicate abstraction [9,14], the strongest (cartesian or boolean, respectively) combination of predicates is stored as a BDD over the propositional variables that represent the predicates in the precision [12]. In data-flow analysis, there exist algorithms for pointer-alias analyses that store the points-to relations in BDDs [4].

1.1.3 BDD-based analysis of software

The verification tool MOPED [35] can verify Java byte-code, by constructing a boolean program (boolean variables and integer variables with a finite range of values can be represented as BDDs) that is internally represented as a symbolic pushdown system. A BDD-based implementation of interpolation [36] was integrated in the abstraction-refinement loop of MOPED. KRATOS [3] [31] verifies SystemC programs and also provides a BDD-based software-verification algorithm.[4] F-SOFT uses a BDD-based reachability analysis for unbounded verification of C programs [41]. The purpose of our study is to compare different abstract domains that are all implemented in the same verification framework (based on the same solver, libraries, and parser).

1.1.4 Domain types

It is possible to perform a static type analysis of all program variables and categorize the integer variables into more fine-grained *domain types* [2], for example, integer variables that are compared to zero (boolean), integer variables that are compared with constants or with other variables (equality), integer variables with addition, etc. It was shown that the choice of the abstract domain per variable based on a domain-type analysis makes a large difference in efficiency and effectiveness [2]. There exists a version of JAVA PATHFINDER that

[2] http://cpachecker.sosy-lab.org/.

[3] https://es-static.fbk.eu/tools/kratos/.

[4] Personal communication with the developers.

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 [48]. This can be seen as a domain-type-based analysis were the domain assignment is hard-coded to boolean variables that represent the feature selection.

1.2 Insights from the experiments

We experimentally evaluate a configurable program analysis (CPA) based on BDDs and compare it with several state-of-the-art techniques on verifying reachability properties of event-condition-action (ECA) programs. The contribution of this work is not to propose a purely BDD-based analysis for software verification as replacement for alternative approaches, but to experimentally show that using BDDs as representation for programs of a *certain category* (that use integer variables in a very restricted way: only equality comparisons and assignments) can be more efficient than other (more expressive, but also more expensive) encodings. The results of the participation at the RERS 2012 challenge with a BDD-based model checker [18] had motivated our work on domain types [2]. The more fine-grained domain-type analysis for each variable in a pre-analysis, followed by an appropriate assignment of an abstract domain for each domain type, is a promising approach to software verification.

2 Preliminaries

In order to define a configurable 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 [11].

2.1 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.[5] 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 l_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 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 $\overset{g}{\to} \subseteq C \times \{g\} \times C$. The complete transition relation $\to$ is the union over all control-flow edges: $\to = \bigcup_{g \in G} \overset{g}{\to}$. We write $c \overset{g}{\to} c'$ if $(c, g, c') \in \to$, and $c \to c'$ if there exists a g with $c \overset{g}{\to} c'$. 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 a *feasible program path*. In order to define an efficient program analysis, we need to define abstract states and abstract transitions.

2.2 Configurable program analysis (CPA)

We use the framework of *configurable program analysis* [11] 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 development work 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, and is used to determine if the fixed point of the iteration algorithm is reached). 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. An abstract-domain element is also called *abstract state*.

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.

2.3 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 [11].

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

[5] The framework CPACHECKER [13], which we use to implement the analysis, accepts C programs and transforms them into a side-effect free form [45]; it also supports interprocedural program analysis.

Algorithm 1 *CPA*($\mathbb{D}$, e_0) (taken from [11])

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$,
an initial abstract state $e_0 \in E$, where E denotes
the set of elements of the semi-lattice of D
Output: a set of reachable abstract states
Variables: two sets reached and waitlist of abstract states

```
1: waitlist := {e_0};
2: reached := {e_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
```

$\mathcal{E}_{\mathbb{L}} = ((L \cup \{\top\}), \sqsubseteq)$, $l \sqsubseteq l'$ if $l = l'$ or $l' = \top$, $[\![\top]\!] = C$, and for all l in L, we have $[\![l]\!] = \{c \in C \mid c(pc) = l\}$.

2. The transfer relation $\rightsquigarrow_{\mathbb{L}}$ has the transfer $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$ if $g = (l, \cdot, l') \in G$.
3. The merge operator $\mathsf{merge}_{\mathbb{L}}$ does not combine abstract states when control flow meets: $\mathsf{merge}_{\mathbb{L}}(l, l') = l'$.
4. The termination check $\mathsf{stop}_{\mathbb{L}}$ returns *true* if the current abstract state is already in the reached set: $\mathsf{stop}_{\mathbb{L}}(l, R) = (l \in R)$.

2.4 Analysis algorithm

Algorithm 1 shows the core iteration algorithm that is used to run a configurable program analysis, as implemented[6] by tools like CPACHECKER. The algorithm is started with a CPA and an initial abstract state e_0. The algorithm terminates if the set waitlist is empty (i.e., all abstract states were processed) and returns the set reached. We start the algorithm with two singleton sets that contain only the initial abstract state (lines 1–2). In each iteration of the `while` loop, the algorithm processes and removes one abstract state e from the set waitlist, computes all abstract successor states for e, and further processes the successors 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 combined (e.g., where the 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.

2.5 Binary decision diagrams (BDD)

A binary decision diagram [24] is (definition taken from [17]) 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 every path from the root to a terminal node. For a given variable order, the BDD representation of a set of variable assignments is unique. The ordering of the variables affects the size of the resulting BDD [22].

A BDD represents a set of value assignments for a set of boolean variables, i.e., it is a compressed representation of a truth table. In our analysis, we need to consider integer variables: we encode integer values as bit vectors, and integer variables as vectors of boolean variables, and thus, can represent data states of integer programs by BDDs.

3 BDD-based program analysis

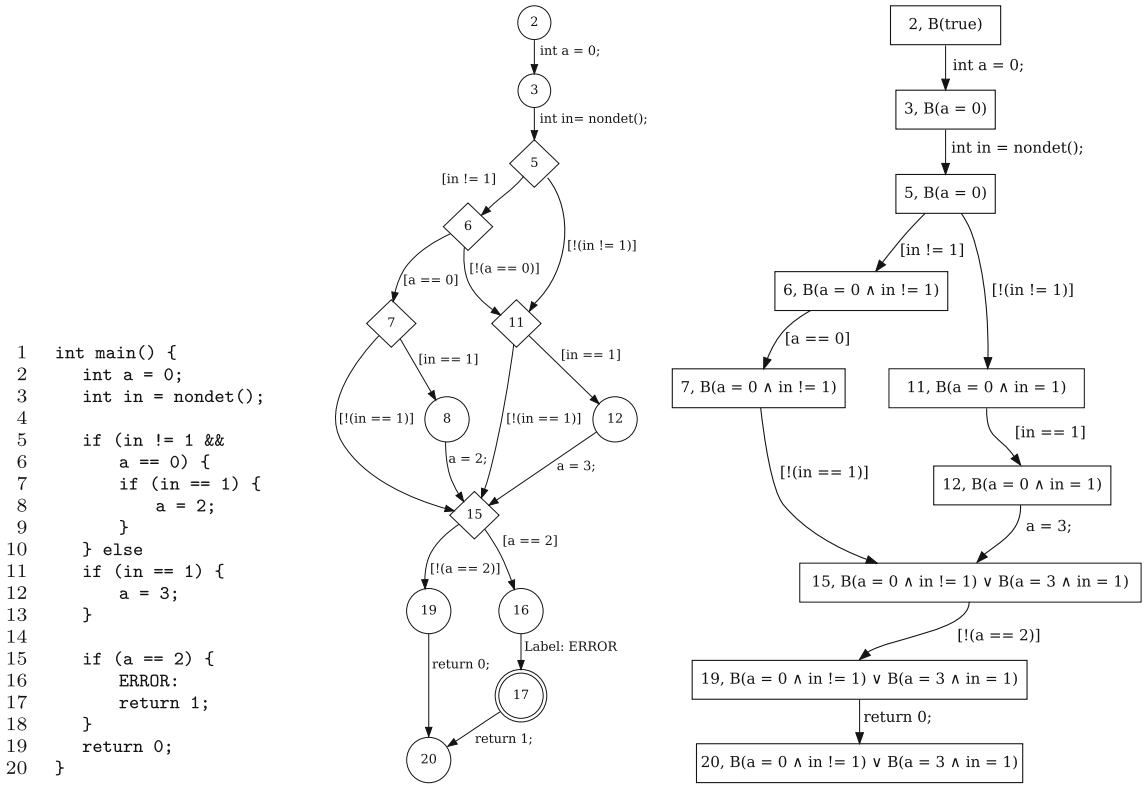
For implementing the BDD-based analysis, we define a configurable program analysis that uses BDDs to represent abstract states. We implement this analysis in the software-verification framework CPACHECKER.

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

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

1. The abstract domain $D_{\mathbb{BPA}} = (C, \mathcal{E}_{\mathcal{B}}, [\![\cdot]\!])$ is based on the semi-lattice $\mathcal{E}_{\mathcal{B}}$ of BDDs, i.e., every abstract state consists of a BDD. The concretization function $[\![\cdot]\!]$ assigns

[6] For ease of illustration, we show only a simplified version.



(a) Source code **(b)** Control-flow automaton **(c)** Abstract reachability graph

Fig. 1 Example C program (**a**) with its corresponding CFA (**b**) and the verification certificate (**c**); the program locations in (**b**) and (**c**) correspond to the line numbers in (**a**) before the line of code is executed. (**a**) Source code. (**b**) Control-flow automaton. (**c**) Abstract reachability graph

to an abstract state $\mathcal{B}$ the set $[\![\mathcal{B}]\!]$ of all concrete states that are represented by the BDD. Formally, the lattice $\mathcal{E}_{\mathcal{B}} = (\widehat{\mathcal{B}}, \sqsubseteq)$—where $\widehat{\mathcal{B}}$ is the set of all BDDs, $\mathcal{B}_{true}$ is the BDD that represents all concrete states (1-terminal), and $\mathcal{B}_{false}$ is the BDD that represents the empty set of states (0-terminal)— is induced by the partial order $\sqsubseteq$ that is defined as: $\mathcal{B} \sqsubseteq \mathcal{B}'$ if $[\![\mathcal{B}]\!] \subseteq [\![\mathcal{B}']\!]$. (The join operator $\sqcup$ of the lattice yields the least upper bound, which is the disjunction $\vee$; $\mathcal{B}_{true}$ is the top element $\top$ of the lattice.)

2. The transfer relation $\rightsquigarrow_{\mathbb{BPA}}$ has the transfer $\mathcal{B} \overset{g}{\rightsquigarrow} \mathcal{B}'$ if

$$\mathcal{B}' = \begin{cases} \mathcal{B} \wedge \mathcal{B}_p & \text{if } g = (l, \texttt{assume}(p), l') \\ & \text{and } \mathcal{B} \wedge \mathcal{B}_p \neq \mathcal{B}_{false} \\ (\exists w : \mathcal{B}) \wedge \mathcal{B}_{w=e} & \text{if } g = (l, \texttt{w} := \texttt{e}, l') \ . \end{cases}$$

3. The merge operator is defined by $\mathsf{merge}_{\mathbb{BPA}}(\mathcal{B}, \mathcal{B}') = \mathcal{B} \sqcup \mathcal{B}'$. (This merge is precise, not overapproximating, because the join is the disjunction of BDDs.)
4. The termination check is defined by $\mathsf{stop}_{\mathbb{BPA}}(\mathcal{B}, R) = \exists \mathcal{B}' \in R : \mathcal{B} \sqsubseteq \mathcal{B}'$.

We construct the composite 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. The resulting composite merge-operator $\mathsf{merge}_{\times}$ satisfies $e' \sqsubseteq \mathsf{merge}_{\times}(e, e')$: it merges two composite states $e = (l, \mathcal{B})$ and $e' = (l', \mathcal{B}')$ such that $(l', \mathcal{B} \vee \mathcal{B}')$ is returned if $l = l'$, and $(l', \mathcal{B}')$ is returned otherwise. For further details on CPA composition, we refer to the literature [11]. The abstract state of the composition analysis consists of the program location from CPA $\mathbb{L}$ and the abstract data state (BDD) from the CPA $\mathbb{BPA}$.

3.1 Example

Consider the program in Fig. 1a, which is represented by the control-flow automaton (CFA) in Fig. 1b. The error location (location 17, indicated by label ERROR) is not reachable in this simple example program, i.e., the program is safe. Figure 1c represents the corresponding abstract-reachability graph (ARG), which is also called 'verification certificate', for this verification task. The edges in the ARG represent successor computations along the control-flow edges of the corresponding CFA. The nodes in the ARG represent abstract states that are stored in the set reached by Algorithm 1, which are initial abstract states or constructed by computing abstract successor states according to the edges of the CFA, using the CPA algorithm and the composition of CPAs as described above. We label each node of the ARG with the program

location (which corresponds to the line number in Fig. 1a before the line is executed) and the BDD that represents the abstract data state. The set of states that are represented by the nodes of the ARG in Fig. 1c equals the set reached after the CPA algorithm has terminated.

The analysis starts at the initial program location $l_0 = 2$ with the initial abstract state $e_0 = (2, \mathcal{B}_{true})$. The analysis then computes the abstract successor states by applying the transfer relation $\rightsquigarrow$. In our example, the abstract data state for location 3 is computed by quantifying the assigned variable in the BDD of the previous abstract state, creating a BDD for the constraint of control-flow edge `int a = 0` (assignment), and conjuncting it with the former BDD. The transfer along the edge (3, `int in = nondet()`, 5) does not change the abstract data state because (1) the variable that is defined by this edge had an unknown value before, and (2) it does not restrict the possible concrete states since the return value of `nondet()` is non-deterministic, i.e., unknown. Successors for CFA edges whose operations are assumptions are computed by conjuncting the BDD of the abstract data state for the predecessor location with the BDD for the respective assumption. For example, consider the CFA edge (5, `[in != 1]`,6): the BDD $\mathcal{B}_{a=0}$ for the abstract predecessor state is conjuncted with the BDD $\mathcal{B}_{in\,!=1}$ for the operation, resulting in the BDD $\mathcal{B}_{a=0} \wedge \mathcal{B}_{in\,!=1}$ for the abstract successor state at location 6. Assignment operations are processed by first existential quantifying the variable that gets assigned a new value, and then the intermediate BDD is conjuncted with the BDD that represents the new value of the variable. For example, consider location 12, which has the BDD $\mathcal{B}_{a=0 \wedge in=1}$ as abstract data state, and process the control-flow edge (12, `a = 3`, 15) (assignment): First, the variable `a` is quantified and BDD $\mathcal{B}_{in=1}$ is obtained as intermediate result, and then the BDD $\mathcal{B}_{a=3}$ for the assignment operation is conjuncted to the intermediate result such that the abstract successor state contains the BDD $\mathcal{B}_{in=1} \wedge \mathcal{B}_{a=3}$ (which is merged with another BDD for location 15).

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 $\mathcal{B}_{(a=0 \wedge in\,!=1) \vee (a=3 \wedge in=1)}$ at location 15 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 successor computation of a given abstract state e stops (the abstract state is not added to the sets waitlist and reached for further processing), whenever an existing abstract state covers (i.e., is implied by) the abstract state e; this check is performed by the CPA operator stop. The analysis does not add abstract states to the set reached (not produced by the transfer relation) for the locations 8 and 16, because the BDDs evaluate to *false*. Thus, the error location 17 is not reachable.

4 Evaluation

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

4.1 Verification tasks

For the evaluation of our approach, we use the benchmark sets of reachability verification tasks from the RERS challenges 2012 and 2013 [39,40]. Many of these programs are in the restricted class of C programs for which a BDD-based verification is interesting and promising. Table 1 lists all programs from the RERS benchmark collection. In total, the collection consists of 2 760 verification tasks with reachability properties. The verification tasks were composed from 46 programs[7] and 60 reachability properties to verify.

The programs were generated automatically by a tool for synthesizing verification problems [46,47], with the goal of investigating the performance of different verification approaches for event-condition-action (ECA) systems. All programs follow the same structure: There is a number of (state) variables (with an initial value) that model the state of the system. The main function consists of one single `while` loop; the loop has no termination condition, i.e., the program does not necessarily terminate (reactive system). In each iteration of this loop, an integer value is consumed from standard input, after which a function is called that computes the successor state of the program, based on the input value and the current state of the program. The safety property (program invariant) is checked in an iteration that does not change any state variable. Whenever an invariant check fails, the function `assert(0)` is called; these calls are labeled with different names to distinguish the different safety properties. Thus, each verification task is a pair of the program and the C label that represents the safety property to be verified.

Table 1 provides a detailed overview over some static and dynamic characteristics of the programs. In the first group of three columns, we quantify the size of the programs by size measures: we report the length of each program (in LOC), the number of state variables, and the number of different integer constants that are contained in the program. The second group of columns gives the number of logical operators: conjunctions, disjunctions, and negations in the program, respectively. The third group of columns gives the number of arithmetic operators: equalities and inequalities, strict and non-strict greater-than and less-than compar-

[7] There are Programs 1–19 and 28–54; there is a gap from number 20 to 27, for which no programs exist.

Table 1 Program classification and measures that characterize the programs (CPU time of `gcc` reported with up to two significant figures)

Program Number	Classification	Size Measures			Logical Operators			Arithmetic Operators				Domain Types [2]				`gcc -c -O0`	
		Lines of Code	State Variables	Integer Constants	Conjunctions	Disjunctions	Negations	==, !=	>, =>, <=, <	+, −	*, /, %	Bool	Eq	Add	Other	CPU Time, in s	Peak Memory, in MB
1	ECA-EQ	587	7	17	906	42	253	1 077	0	0	0	0	12	0	0	.03	58
2	ECA-EQ	608	6	18	795	50	231	993	0	0	0	0	11	0	0	.05	56
3	ECA-EQ	1 661	30	13	2 516	329	1 122	3 062	0	0	0	0	32	0	0	.11	106
4	ECA-EQ	4 809	8	19	7 369	768	1 140	8 950	0	0	0	0	22	0	0	.31	165
5	ECA-EQ	11 114	8	21	14 131	1 596	1 362	17 626	0	0	0	0	34	0	0	.58	264
6	ECA-EQ	9 463	30	19	12 525	2 295	4 402	15 922	0	0	0	0	47	0	0	.54	262
7	ECA-EQ	73 554	11	17	116 254	10 158	30 132	136 555	0	0	0	0	151	0	0	4.5	1 177
8	ECA-EQ	171 328	11	19	223 012	18 288	88 998	261 792	0	0	0	0	259	0	0	9.8	2 275
9	ECA-EQ	184 822	30	21	235 770	39 542	56 866	293 680	0	0	0	0	257	0	0	11	2 537
10	ECA-MUL	518	5	111	749	60	0	651	296	152	122	1	8	0	1	.06	64
11	ECA-MUL	891	6	336	1 296	156	78	1 025	626	450	369	0	9	0	2	.11	91
12	ECA-MUL	4 063	5	1 278	5 164	744	0	3 797	2 858	1 927	1 645	0	17	0	2	.36	178
13	ECA-MUL	4 975	6	1 914	6 168	745	421	4 302	3 516	3 247	2 825	0	18	0	2	.50	222
14	ECA-MUL	740	4	160	825	126	0	785	343	174	150	0	8	0	1	.06	69
15	ECA-MUL	1 547	4	854	1 830	240	0	622	1 774	1 252	1 122	0	9	0	3	.20	136
16	ECA-MUL	1 498	4	324	1 695	258	0	1 403	877	553	467	0	11	0	1	.11	103
17	ECA-MUL	2 294	5	1 418	3 042	360	0	804	3 070	2 181	1 835	0	9	1	3	.31	164
18	ECA-MUL	3 500	4	784	3 453	684	0	3 467	1 357	1 024	899	0	14	0	1	.25	134
19	ECA-MUL	8 273	5	3 872	9 150	1 212	0	5 376	6 503	5 333	4 796	0	22	0	3	.78	286
28	ECA-EQ	2 050	138	23	1 474	643	565	2 441	0	0	0	0	167	0	0	.11	69
29	ECA-MUL	1 737	121	394	1 478	566	404	1 694	637	459	677	0	125	2	23	.16	78
30	ECA-MUL	2 067	142	422	1 563	628	496	2 010	493	410	539	0	85	3	83	.16	81
31	ECA-EQ	7 396	227	21	3 640	1 916	1 725	6 373	0	0	0	0	330	0	0	.29	123
32	ECA-MUL	9 783	282	2 014	5 374	2 437	1 794	6 872	1 986	3 759	5 516	0	359	1	60	.86	229
33	ECA-MUL	11 030	340	1 359	5 709	2 699	2 127	7 807	1 740	2 107	3 091	0	182	1	300	.74	219
34	ECA-EQ	93 613	1 754	20	35 032	19 278	17 121	62 071	0	0	0	0	3 011	0	0	2.8	935
35	ECA-MUL	110 456	1 923	14 270	46 954	22 567	16 246	61 201	17 412	40 812	60 088	0	3 125	3	391	9.6	1 635
36	ECA-MUL	111 403	2 278	8 674	46 229	23 075	18 288	64 003	14 516	18 471	27 249	0	1 224	0	2 473	7.0	1 311
37	ECA-EQ	126 336	329	24	45 015	25 597	23 234	81 869	0	0	0	0	1 461	0	0	3.4	1 158
38	ECA-MUL	168 870	440	5 097	67 583	33 592	23 891	90 363	25 208	52 425	78 265	0	1 764	1	89	13	1 624
39	ECA-MUL	121 746	496	4 082	50 004	25 308	20 249	69 924	16 133	36 360	52 812	0	232	0	1 185	9.4	1 623
40	ECA-EQ	767 117	1 816	24	230 757	132 236	117 466	421 306	0	0	0	0	6 716	0	0	21	4 491
41	ECA-MUL	963 292	2 143	23 014	327 799	164 083	116 747	438 389	125 994	416 552	610 920	0	7 546	1	448	100	10 263
42	ECA-MUL	790 653	2 301	16 773	280 185	141 626	113 158	394 887	89 137	221 660	327 724	0	1 199	1	7 934	68	8 056
43	ECA-EQ	6 361 733	13 208	26	1 738 525	995 532	886 983	3 173 394	0	0	0	0	46 804	0	0	410	34 621
44	ECA-EQ	3 643 670	14 201	25	952 110	550 005	487 826	1 750 532	0	0	0	0	33 991	0	0	160	18 937
45	ECA-EQ	4 385 069	22 173	117	1 171 310	672 959	639 215	2 143 818	0	0	0	1	13 034	0	31 583	450	29 341
46	ECA-EQ	484 639	264	29	237 172	31 252	86 543	307 825	0	0	0	0	933	0	0	13	2 949
47	ECA-MUL	266 699	253	15 082	147 126	17 304	38 962	147 211	39 252	125 583	189 603	0	577	0	52	27	3 181
48	ECA-MUL	766 918	544	19 763	403 541	48 166	120 300	422 319	91 137	265 168	402 452	0	257	0	1 240	77	8 318
49	ECA-EQ	2 398 639	1 392	30	1 074 426	135 527	389 827	1 386 381	0	0	0	0	5 044	0	0	140	13 311
50	ECA-MUL	4 618 987	1 870	33 361	2 209 347	260 242	585 072	2 210 721	594 145	1 395 380	2 076 128	0	8 369	0	385	1 100	39 512
51	ECA-MUL	7 927 698	3 096	33 431	3 662 316	425 837	1 092 415	3 820 245	823 889	2 013 760	2 985 215	0	1 551	0	10 610	2 300	70 096
52	ECA-EQ	870 021	2 578	30	305 293	168 509	152 144	541 495	0	0	0	0	6 882	0	0	30	5 259
53	ECA-MUL	878 564	2 473	28 966	333 339	161 699	116 594	440 377	119 604	280 688	391 546	0	5 658	1	488	75	7 515
54	ECA-MUL	5 902 289	9 616	34 607	1 691 858	877 408	693 174	2 428 426	540 507	1 656 141	2 469 453	0	4 964	0	20 809	1 500	50 405

isons, addition and subtraction, and finally multiplication, division, and modulo. These measures provide an overview of how the designers of the benchmarks were choosing the parameters for increasing the complexity: for example, for Programs 1–9, the number of disjunctions monotonically increases. Other measures (length, conjunctions, negations, equality checks) also indicate an increase of the designated complexity. The benchmark generation has different parameters [46,47], and those were varied throughout the benchmark set.

Considering the BDD-based representation, and mainly the well-known insight that BDDs do not scale well for the operation multiplication [25], we partition the set of programs into two partitions: the first partition ECA-EQ contains all programs in which no multiplication of integer variables occurs, and the second partition ECA-MUL contains all programs that do not have that restriction in terms of operations on integer variables. Our partitioning is given in the (second) column 'Classification'. The fourth group of columns gives the domain types that a domain-type analysis [2] would have assigned. Domain types are assigned to individual variables; much more fine-grained than our assignment to the two partitions ECA-EQ and ECA-MUL. Our classification can be derived from domain types in the following way: if all variables of a program are of domain type BOOL or EQ, then the program is in partition ECA-EQ, otherwise in partition ECA-MUL. In other words, ECA-EQ contains all programs whose variables are only used in equality expressions (`!=` and `==`) and not with other arithmetic operators. If any other arithmetic operation is used on a program variable, then the

program belongs to partition ECA-MUL. The classification of the programs is done by running a simple syntactical analysis.

Table 1 shows that some programs are extremely large (in terms of LOC and the number of state variables), and our verification infrastructure can not handle those. To quantify the effect, we report the run time (CPU time with up to two significant figures) and memory consumption of GCC in the last two columns of the table. Based on those numbers, we have excluded eight programs from the benchmark set for our experiments, because solving them within the given resource constraints is not feasible. We restrict ourselves to those programs that can be compiled with a standard compiler (GCC 4.6.3 without linking and without optimization: `gcc -c -O0`) with less than 100 s of CPU time and with less than 10 GB of RAM. The excluded programs are highlighted in the table by bold background color.

In total, we consider 2 280 (out of 2 760) verification tasks for our experimental evaluation. The set of benchmark verification tasks and tables with detailed results are publicly available on the supplementary web page.[8]

4.2 Experimental setup

All experiments were performed on machines with an Intel Core i7-2600 3.4 GHz CPU and 32 GB of RAM. OpenJDK 1.7.0_55 was used as the Java runtime environment and Linux 3.2.0-64 as operating-system kernel. We restricted the resources for each verification run to two CPU cores, 15 min of CPU time, and 15 GB of RAM. In order to leave sufficient RAM for the SMT solver, we configured a Java heap size to 10 GB. We assigned the subversion tag `cpachecker-1.3.4-sttt14` to the version of CPAchecker that we have used for our experiments. The subversion repository is publicly available.[9] We used MathSAT 5.2.10 [32] as SMT solver, and JavaBDD 1.0b2[10] as BDD package. We setup the JavaBDD package with an initial node table of size 100 million entries and a cache size of 500 thousand entries.

4.3 Compared verification approaches

We first compared five standard configurations based on fundamentally different verification techniques. The approaches are all implemented in the same verification tool, in order to eliminate influence of the used SMT solver, libraries, and parser. Then, based on the results, we created a combination analysis that first analyses the program syntactically and if the program is suitable for a BDD-based analysis, then we verify it using BDDs, otherwise we verify it with another approach (selection from portfolio).

The first configuration 'Explicit Value' is an explicit-value analysis with counterexample-guided abstraction refinement (CEGAR) [16]. An abstract state of this analysis describes the value of variables enumeratively (not symbolically) using value assignments. CEGAR with constraint-sequence interpolation [16] is used in order to track only those variables that are relevant to verify the safety property. The second configuration 'Bounded Loops (BMC)' is a standard bounded model-checking (BMC) configuration that unwinds loops up to a bound of 100. This analysis does not compute abstractions. The third configuration 'Predicate Abstraction' is a predicate analysis that computes boolean predicate abstractions based on adjustable-block encoding (ABE) [14]. The analysis is configured to compute abstractions only at loop heads. Since each program from the benchmark set contains exactly one loop, there is only one program location for which abstractions are computed. The predicates for the abstraction precision are derived from infeasible counterexample paths using CEGAR [33] and Craig interpolation [34,37]. The fourth configuration 'Predicate Impact' uses the IMPACT algorithm [19,44], which, in contrast to predicate abstraction, does not compute strongest-post conditions and abstracts those to more abstract formulas, but uses a conjunction of Craig interpolants as abstract states. A detailed conceptual and experimental comparison of 'Predicate Abstraction' and 'Predicate Impact' is available in the literature [19]. The fifth configuration 'BDD' is a BDD-based analysis that was described earlier in this paper.

The new configuration 'BDD + Predicate Impact' was introduced based on the results of the first set of experiments, and runs a different analysis depending on the program classification. If the input program belongs to class ECA-EQ, then it performs the BDD-based analysis, otherwise it performs the IMPACT-based analysis.

4.4 Results for ECA-EQ

First, we apply all five configurations to the 16 programs of class ECA-EQ[11], which yields a total of 960 verification tasks (16 programs × 60 properties). Table 2 presents the results: the first column specifies the program by its program number, for each of the five analyses, one group of columns represents the corresponding results. In each of the column groups, the first sub-column gives the number of solved verification tasks, the second and third sub-columns indicate the verification time (sum and mean, respectively), where all values are given in seconds of CPU time with two significant figures. The fourth sub-column gives the number of abstract states in the set of reached states (in 1 000 states).

[8] http://www.sosy-lab.org/~dbeyer/cpa-bdd/.

[9] https://svn.sosy-lab.org/software/cpachecker/tags/.

[10] http://javabdd.sourceforge.net/.

[11] Four ECA-EQ programs were removed due to GCC timeout.

Table 2 Verification tasks for programs of class ECA-EQ

Analysis	Explicit Value				Bounded Loops (BMC)				Predicate Abstraction				Predicate Impact				**BDD**			
Program Number	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)
1	60	550	9.2	29.3	13	120	9.5	7.0	60	680	11	8.5	60	600	10	6.7	60	440	7.3	5.8
2	60	520	8.7	16.3	7	45	6.5	2.7	60	590	9.9	5.7	60	550	9.2	6.1	60	380	6.4	2.3
3	60	5 400	91	877.6	13	200	15	17.4	60	1 300	22	28.1	43	930	22	23.9	60	840	14	12.4
4	60	12 000	200	1 301.3	24	6 400	270	213.2	60	4 500	75	181.2	60	4 900	81	96.8	60	2 100	35	197.4
5	41	21 000	520	4 015.1	24	1 400	60	162.6	60	6 800	110	168.2	36	4 300	120	171.9	60	2 100	35	110.7
6					25	1 500	62	106.1	57	5 400	94	129.0	16	1 100	68	102.2	60	1 500	26	74.0
7					23	7 100	310	1 012.9					59	23 000	380	946.7				
8																				
9																				
28	28	880	32	436.7	28	470	17	14.2	56	2 800	49	11.8	47	1 900	40	9.2	60	810	13	22.7
31	13	1 500	120	1 245.0	29	1 000	35	38.4	51	11 000	220	22.1	47	6 100	130	26.0	29	550	19	16.1
34	7	1 700	250	2 718.2					1	300	300	148.7	11	3 600	320	236.9	27	18 000	680	232.7
37	1	25	25	99.0	26	9 200	360	372.2	1	220	220	198.1	1	71	71	99.5	27	740	27	8.5
40																	29	6 400	220	55.2
46																	29	1 800	62	19.0
52	3	900	300	2 444.6																
Total solved	333	44 000	130	1 059.9	212	28 000	130	219.3	466	34 000	73	70.8	440	46 000	110	171.9	**561**	36 000	**64**	61.8

The last line of the table, 'Total solved', summarizes the overall, solved results: The BDD-based analysis outperforms the other analyses in both relevant measures. The analysis is the most effective: it solves 561 verification tasks. The analysis is also the most efficient: it has the lowest average CPU time consumption of 64 s per verification task. All other approaches solve fewer verification tasks and need significantly more time (on average per task).

Figure 2 illustrates the results using a quantile plot (cf. [7] for more details on this type of plots). For each approach, we plot a graph (a series of data points). One data point (x, y) in such a graph indicates that x verification tasks were successfully verified in up to y seconds of CPU time each, by the corresponding configuration. The integral below each graph illustrates the accumulated verification time for all solved verification tasks. The plot in Fig. 2 illustrates that the BDD-based analysis outperforms the other analyses both in terms of effectiveness and efficiency.

A more detailed picture of the results of the BDD-based analysis is shown in Table 3. The results can be interpreted with respect to the structure and size of the programs. In general, the performance of the BDD-based analysis decreases with a growing number of BDD nodes. The number of BDD nodes depends on the number of variables that are encoded (and on the ordering of the variables, but we do not discuss variable orderings here). For example, Program 34 has a large number of state variables (compared to Programs 1–7), and thus, the BDD-based analysis needs significantly more time for verification tasks of Program 34. Programs 34, 40, and 52 use more than 1 000 state variables. Due to the large state space of those programs, the analysis does not succeed in

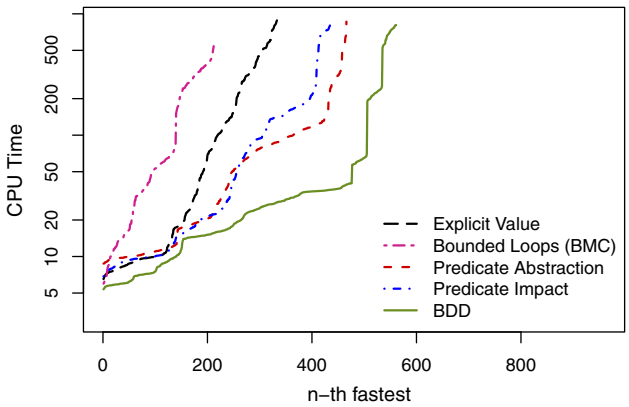


Fig. 2 Verification tasks for programs of class ECA-EQ, CPU time given in seconds, y-axis uses logarithmic scale

correctness proofs (for which the whole state space must be inspected). The BDD-based analysis can find violations of properties, because it is not necessary to inspect the complete state space in those cases.

Also the range of values per variable matters: one integer variable in the program is encoded by several boolean variables in the BDD; the number of boolean variables depends on the number of different values that a program variable can have. Since programs from the category ECA-EQ contain only the operators == and !=, the number of different values can be computed from the number of different integer constants that are used in combination with the variables. Also, combining the BDD-based analysis with constant propagation might be an optimization for reducing the number of variables to be stored in the BDDs.

Table 3 BDD analysis: details for verification tasks for programs of type ECA-EQ

Program Number	Program structure and relevant operators						BDD Analysis					
	State Variables	Integer Constants	==, !=	Conjunctions	Disjunctions	Negations	Solved 'True'	Solved 'False'	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)	Number of BDD nodes, in 1 000 nodes (mean)
1	7	17	1 077	906	42	253	47	13	440	7.3	5.8	138.4
2	6	18	993	795	50	231	53	7	380	6.4	2.3	58.6
3	30	13	3 062	2 516	329	1 122	47	13	840	14	12.4	242.0
4	8	19	8 950	7 369	768	1 140	36	24	2 100	35	197.4	7 460.0
5	8	21	17 626	14 131	1 596	1 362	36	24	2 100	35	110.7	6 214.7
6	30	19	15 922	12 525	2 295	4 402	35	25	1 500	26	74.0	2 965.7
7	11	17	136 555	116 254	10 158	30 132						
8	11	19	261 792	223 012	18 288	88 998						
9	30	21	293 680	235 770	39 542	56 866						
28	138	23	2 441	1 474	643	565	32	28	810	13	22.7	467.9
31	**227**	21	6 373	3 640	1 916	1 725	**0**	29	550	19	16.1	1 811.6
34	**1 754**	20	62 071	35 032	19 278	17 121	**0**	27	18 000	**680**	232.7	70 856.2
37	**329**	24	81 869	45 015	25 597	23 234	**0**	27	740	27	8.5	1 510.0
40	**1 816**	24	421 306	230 757	132 236	117 466	**0**	29	6 400	**220**	55.2	43 818.8
46	**264**	29	307 825	237 172	31 252	86 543	**0**	29	1 800	62	19.0	1 991.8
52	**2 578**	30	541 495	305 293	168 509	152 144						

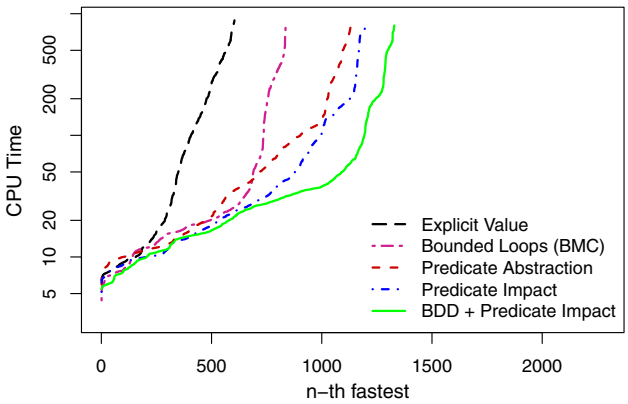


Fig. 3 All verification tasks (all programs)

4.5 Results for ECA-MUL

The benchmark set ECA-MUL contains 1 320 verification tasks for 22 programs[12] that use multiplications among the arithmetic operations in the programs (cf. Table 1). BDDs are known to be inefficient for representing multiplications [25]. Therefore, we do not discuss the configuration that is 'purely' based on BDDs any further for programs in class ECA-MUL.

We start presenting the overall picture: Figure 3 shows that on the full set of all 38 programs, the configuration 'Predicate Impact' is the best individual analysis, followed by 'Predicate Abstraction'. However, a configuration that chooses an appropriate analysis technique automatically, based on the insights from the previous experiments, can considerably increase the effectiveness and efficiency of the verification process (cf. right-most graph for 'BDD + Predicate Impact').

Table 4 presents the details, where the column structure is similar to Table 2, just with the new column group for 'BDD + Predicate Impact' instead of 'BDD'. The predicate-impact analysis can solve 1,206 verification tasks, which is 68 verification tasks more than the predicate-abstraction-based analysis can solve. The approach that is based on bounded model checking solves 837 verification tasks, and the approach based on explicit-state model checking with abstraction can solve only 604 verification tasks.

The combination analysis 'BDD + Predicate Impact' is superior in our context: it verifies 1,329 verification tasks and is thus the most *effective* verification approach; and with an average of 54 s of CPU time per verification task, the analysis is also the most *efficient* approach.

Our results illustrate that it is beneficial to classify programs according to the operations that occur in the program and to run an appropriate analysis that can provide the results in the most effective and efficient way. The IMPACT-based analysis performs quite well, independent from the program class, but can be significantly improved by combining it with the BDD-based analysis.

5 Conclusion

We implemented a BDD-based verification approach for software model checking [18] and explored the application of a purely BDD-based analysis to software, namely the programs from the RERS challenge [39]. We compared the effective-

[12] Four ECA-MUL programs were removed due to GCC timeout.

Table 4 Overall results for all verification tasks (all programs)

Analysis	Explicit Value				Bounded Loops (BMC)				Predicate Abstraction				Predicate Impact				**BDD + Predicate Impact**			
Program Number	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)	Solved	Time (sum)	Time (mean)	Size of reached set, in 1 000 states (mean)
1	60	550	9.2	29.3	13	120	9.5	7.0	60	680	11	8.5	60	600	10	6.7	60	450	7.5	4.9
2	60	520	8.7	16.3	7	45	6.5	2.7	60	590	9.9	5.7	60	550	9.2	6.1	60	380	6.4	2.1
3	60	5 400	91	877.6	13	200	15	17.4	60	1 300	22	28.1	43	930	22	23.9	60	830	14	10.4
4	60	12 000	200	1 301.3	24	6 400	270	213.2	60	4 500	75	181.2	60	4 900	81	96.8	60	2 100	35	156.1
5	41	21 000	520	4 015.1	24	1 400	60	162.6	60	6 800	110	168.2	36	4 300	120	171.9	60	2 100	35	95.5
6					25	1 500	62	106.1	57	5 400	94	129.0	16	1 100	68	102.2	60	1 500	25	62.2
7					23	7 100	310	1 012.9					59	23 000	380	946.7				
8																				
9																				
10	39	330	8.5	22.6	27	170	6.5	2.8	60	540	9.0	4.3	60	470	7.8	4.0	60	530	8.9	4.0
11	15	240	16	106.3	60	450	7.5	3.2	60	710	12	4.3	60	600	10	4.3	60	660	11	4.3
12	24	7 200	300	2 133.2	60	1 100	18	17.9	60	2 400	39	22.3	60	1 500	25	22.3	60	1 700	28	22.3
13	9	3 300	360	795.0	60	1 100	18	17.8	56	5 200	93	23.9	60	2 600	44	24.7	60	2 800	46	24.7
14	29	370	13	172.2	41	460	11	9.1	60	820	14	9.4	60	750	13	8.7	60	820	14	8.7
15	25	760	30	196.1	60	680	11	6.2	60	2 000	34	8.5	60	990	16	8.5	60	1 100	18	8.5
16	5	590	120	2 198.3	40	1 100	28	32.0	60	1 800	30	27.0	60	1 800	30	22.3	60	1 900	32	22.3
17	24	2 500	110	495.4	60	900	15	11.0	60	19 000	320	12.6	60	1 600	27	12.6	60	1 700	29	12.6
18	18	4 400	240	1 832.8	50	1 100	21	31.2	60	2 900	48	32.0	60	2 000	33	30.2	60	2 100	35	30.2
19	11	5 000	450	8 680.8	60	1 300	22	33.6	59	16 000	270	46.3	60	8 100	140	46.4	60	8 100	140	46.4
28	28	880	32	436.7	28	470	17	14.2	56	2 800	49	11.8	47	1 900	40	9.2	60	780	13	14.4
29	15	580	39	589.0	31	490	16	6.3	32	1 900	59	4.3	56	4 100	73	6.0	57	4 200	74	6.0
30	21	750	36	177.8	38	600	16	6.4	36	2 800	79	4.3	59	4 700	79	5.5	60	5 100	85	5.6
31	13	1 500	120	1 245.0	29	1 000	35	38.4	51	11 000	220	22.1	47	6 100	130	26.0	29	530	18	9.2
32	10	420	42	438.2	12	1 800	150	20.6	4	2 000	500	19.6	14	980	70	17.8	14	1 100	76	17.8
33	8	1 300	160	1 451.8	20	6 700	340	41.1	1	16	16	.2	23	1 700	74	23.3	23	1 800	80	23.3
34	7	1 700	250	2 718.2					1	300	300	148.7	11	3 600	320	236.9	27	14 000	520	118.9
35	6	270	45	422.6									6	3 900	650	172.0	6	3 800	630	172.0
36	2	200	99	1 219.3									4	2 000	490	139.7	4	1 900	480	139.7
37	1	25	25	99.0	26	9 200	360	372.2	1	220	220	198.1	1	71	71	99.5	27	720	27	4.4
38					2	810	400	39.4												
39	3	78	26	.3	1	26	26	1.5	1	38	38	.1	1	38	38	.1	1	41	41	.1
40																	29	6 000	210	26.5
42	3	410	140	.5	2	390	200	9.4	2	840	420	.6	2	840	420	.6	2	960	480	.6
46																	29	1 800	61	10.5
47																				
48	3	340	110	.3	1	100	100	1.6	1	120	120	.3	1	110	110	.3	1	140	140	.3
52	3	900	300	2 444.6																
53	1	170	170	639.0																
Total solved	604	74 000	120	1 008.7	837	47 000	56	67.8	1 138	93 000	82	39.2	1 206	85 000	71	74.4	**1 329**	72 000	**54**	29.7

ness and efficiency of the BDD-based symbolic analysis to one enumerative technique and four other symbolic techniques, all implemented in the state-of-the-art verification framework CPACHECKER.

The experiments show that the BDD-based approach is the most efficient verification approach for a restricted class of programs. Based on this insight from our first set of evaluations, we experimented further with a simple combination analysis, where an appropriate verification approach is chosen based on the static characteristics of the program. This combination analysis shows a significant improvement in effectiveness and efficiency.

This was an important insight for related work: It is promising to pre-analyze the program in order to find out for each variable how it is used, and then determine—based on its type of use—the most efficient abstract domain to track this variable [2].

References

1. Anderson, R.J., Beame, P., Burns, S., Chan, W., Modugno, F., Notkin, D., Reese, J.D.: Model checking large software specifications. In: Proc. FSE, pp. 156–166. ACM (1996)
2. Apel, S., Beyer, D., Friedberger, K., Raimondi, F., Rhein, A.V.: Domain types: Abstract-domain selection based on variable usage. In: Proc. HVC, LNCS 8244, pp. 262–278. Springer, Berlin (2013)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstractions for model checking C programs. In: Proc. TACAS, LNCS 2031, pp. 268–283. Springer, Berlin (2001)
4. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: Proc. PLDI, pp. 103–114. ACM (2003)
5. Beyer, D.: Improvements in BDD-based reachability analysis of timed automata. In: Proc. FME, LNCS 2021, pp. 318–343. Springer, Berlin (2001)
6. Beyer, D.: Relational programming with CrocoPat. In: Proc. ICSE, pp. 807–810. ACM (2006)
7. Beyer, D.: Second competition on software verification. In: Proc. TACAS, LNCS 7795, pp. 594–609. Springer, Berlin (2013)

8. Beyer, D.: Status report on software verification. In: Proc. TACAS, LNCS 8413, pp. 373–388. Springer, Berlin (2014)
9. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD, pp. 25–32. IEEE (2009)
10. 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)
11. Beyer, D., Henzinger, T. A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV, LNCS 4590, pp. 504–518. Springer, Berlin (2007)
12. Beyer, D., Henzinger, T. A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008)
13. Beyer, D., Keremoglu, M. E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV, LNCS 6806, pp. 184–190. Springer, Berlin (2011)
14. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
15. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: A tool for BDD-based verification of real-time systems. In: Proc. CAV, LNCS 2725, pp. 122–125. Springer, Berlin (2003)
16. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE, LNCS 7793, pp. 146–162. Springer, Berlin (2013)
17. Beyer, D., Noack, A.: Can decision diagrams overcome state space explosion in real-time verification? In: Proc. FORTE, LNCS 2767, pp. 193–208. Springer, Berlin (2003)
18. Beyer, D., Stahlbauer, A.: BDD-based software model checking with CPAchecker. In: Proc. MEMICS, LNCS 7721, pp. 1–11. Springer, Berlin (2013)
19. Beyer, D., Wendler, P.: Algorithms for software model checking: Predicate abstraction vs. Impact. In: Proc. FMCAD, pp. 106–113. FMCAD (2012)
20. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS, LNCS 1579, pp. 193–207. Springer, Berlin (1999)
21. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and symbolic reachability. In: Proc. CAV, LNCS 6174, pp. 354–359. Springer, Berlin (2010)
22. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. **45**(9), 993–1002 (1996)
23. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine. S.: KRONOS: A model-checking tool for real-time systems. In: Proc. CAV, LNCS 1427, pp. 546–550. Springer, Berlin (1998)
24. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **C-35**(8):677–691 (1986)
25. Bryant, R.E.: On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. IEEE Trans. Comput. **40**(2), 205–213 (1991)
26. 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)
27. 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)
28. Campos, S.V.A., Clarke, E.M.: The Verus language: representing time efficiently with BDDs. In: Proc. ARTS, LNCS 1231, pp. 64–78. Springer, Berlin (1997)
29. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An open-source tool for symbolic model checking. In: Proc. CAV, LNCS 2404, pp. 359–364. Springer, Berlin (2002)
30. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Proc. CAV, LNCS 1633, pp. 495–499. Springer, Berlin (1999)
31. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos: A software model checker for SystemC. In: Proc. CAV, LNCS 6806, pp. 310–316. Springer, Berlin (2011)
32. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS, LNCS 7795, pp. 93–107. Springer, Berlin (2013)
33. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
34. Craig, W.: Linear reasoning. A new form of the Herbrand–Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957)
35. Esparza, J.: Building a software model checker. In: Proc. Formal logical methods for system security and correctness, pp. 53–87. IOS Press (2008)
36. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. In: Proc. TACAS, LNCS 3920, pp. 489–503. Springer, Berlin (2006)
37. Henzinger, T. A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL, pp. 232–244. ACM (2004)
38. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002)
39. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS grey-box challenge 2012: Analysis of event–condition–action systems. In: Proc. ISoLA, LNCS 7609, pp. 608–614. Springer, Berlin (2012)
40. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D., Păsăreanu, C.S.: Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. Int. J. Softw. Tools Technol. Transfer. doi:10.1007/s10009-014-0337-y (2014)
41. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M.K., Kahlon, V., Wang, C., Yang, Z.: Model checking C programs using F-Soft. In: Proc. ICCD, pp. 297–308. IEEE (2005)
42. Lerda, F., Sinha, N., Theobald, M.: Symbolic model checking of software. Electr. Notes Theor. Comput. Sci. **89**(3), 480–498 (2003)
43. McMillan, K.L.: The SMV system. Technical Report CMU-CS-92-131. Carnegie Mellon University (1992)
44. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV, LNCS 4144, pp. 123–136. Springer, Berlin (2006)
45. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of C programs. In: Proc. CC, LNCS 2304, pp. 213–228. Springer, Berlin (2002)
46. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation. In: Proc. SPIN, LNCS 7976, pp. 341–357. Springer, Berlin (2013)
47. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation: Synthesizing programs of realistic structure. Int. J. Softw. Tools Technol. Transfer. doi:10.1007/s10009-014-0336-z (2014)
48. von Rhein, A., Apel, S., Raimondi, F.: Introducing binary decision diagrams in the explicit-state verification of Java code. In: Proceedings of Java Pathfinder Workshop (2011)