Explicit-State Software Model Checking Based on CEGAR and Interpolation[⋆]

Dirk Beyer and Stefan Löwe

University of Passau, Germany

Abstract. Abstraction, counterexample-guided refinement, and interpolation are techniques that are essential to the success of predicate-based program analysis. These techniques have not yet been applied together to explicit-value program analysis. We present an approach that integrates abstraction and interpolation-based refinement into an explicit-value analysis, i.e., a program analysis that tracks explicit values for a specified set of variables (the precision). The algorithm uses an abstract reachability graph as central data structure and a path-sensitive dynamic approach for precision adjustment. We evaluate our algorithm on the benchmark set of the Competition on Software Verification 2012 (SV-COMP'12) to show that our new approach is highly competitive. We also show that combining our new approach with an auxiliary predicate analysis scores significantly higher than the SV-COMP'12 winner.

1 Introduction

Abstraction is one of the most important techniques to successfully verify industrial-scale program code, because the abstract model omits details about the concrete semantics of the program that are not necessary to prove or disprove the program's correctness. Counterexample-guided abstraction refinement (CEGAR) [13] is a technique that iteratively refines an abstract model using counterexamples. A counterexample is a witness of a property violation. In software verification, the counterexamples are error paths, i.e., paths through the program that violate the property. CEGAR starts with the most abstract model and checks if an error path can be found. If the analysis of the abstract model does not find an error path, the analysis terminates, reporting that no violation exists. If the analysis finds an error path, the path is checked for feasibility, i.e., if the path is executable according to the concrete program semantics. If the error path is feasible, the analysis terminates, reporting the violation of the property, together with the feasible error path as witness. If the error path is infeasible, the violation is due to a too coarse abstract model and the infeasible error path is used to automatically refine the current abstraction. Then the analysis proceeds. Several successful software verifiers are based on abstraction and CEGAR (e.g., [4, 6, 9, 14]). Craig interpolation is a technique from logics that yields for two contradicting formulas an interpolant that contains less information than the first formula, but still enough to contradict the second formula [15]. In software verification, interpolation can be used to extract information from infeasible error paths [19], where the resulting interpolants are used to refine

[⋆] An extended version of this article appeared as Tech. Report MIP-1205, University of Passau, 2012 [11].

```
1 extern int system_call();
2 int main(int x) {
3   int flag = 0, ticks = 0, result;
4   while(1) {
5     ticks  = ticks + 1; result = system_call();
6     if(result == 0 || ticks > x) { break; }
7   }
8   if(flag > 0) { ERROR: return 1; }
9 }
```

Listing 1.1. Example to illustrate the effectiveness of CEGAR-based explicit-value analysis

the abstract model. Predicate abstraction is a successful abstraction technique for software model checking [16], because its symbolic state representation blends well with strongest post-conditions, and abstractions can be computed efficiently with solvers for satisfiability modulo theories (SMT) [3]. CEGAR and lazy refinement [20] together with interpolation [19] effectively refine abstract models in the predicate domain. The competition on software verification (SV-COMP'12 [5], Table 3) shows that these advancements had a strong impact on the success of participating tools (e.g., [6, 9, 23]).

Despite the success of abstraction, CEGAR, and interpolation in the field of predicate analysis, these techniques have not yet been combined and applied together to explicit-value analysis. We integrate these three techniques into an explicit-value analysis, a rather unsophisticated analysis that tracks for each program variable its current value explicitly (like constant propagation [1], but without join). First, we have to define the notion of abstraction for the explicit-value domain, and the precision of the analysis (i.e., the level of abstraction) by a set of program variables that the analysis has to track. Second, in order to automatically determine the necessary precision (i.e., a *small* set of program variables that *need* to be tracked) we use CEGAR iterations to discover finer precisions from infeasible error paths. Third, we define interpolation for the explicit-value domain and use this idea to construct an algorithm that efficiently extracts such a parsimonious precision that is sufficient to eliminate infeasible error paths.

Example. Consider the simple example program in Listing 1.1. This program contains a $while$ loop in which a system call occurs. The loop exits if either the system call returns 0 or a previously specified number of iterations x was performed. Because the body of the function $system\_call$ is unknown, the value of $result$ is unknown. Also, the assumption $[ticks > x]$ cannot be evaluated to $true$, because x is unknown. This program is correct, i.e., the error location in line 10 is not reachable. However, a simple explicit-value model checker that always tracks every variable would unroll the loop, always discovering new states, as the expression $ticks = ticks + 1$ repeatedly assigns new values to variable $ticks$. Thus, due to extreme resource consumptions, the analysis would not terminate within practical time and memory limits, and is bound to give up on proving the safety property, eventually.

The new approach for explicit-value analysis that we propose can efficiently prove this program safe, because it tracks only those variables that are necessary to refute the infeasible error paths. In the first CEGAR iteration, the precision of the analysis is empty, i.e., no variable is tracked. Thus, the error location will be reached. Now, using our interpolation-inspired method to discover precisions from counterexample paths, the algorithm identifies that the variable $flag$ (more precisely, the constraint $flag = 0$) has to be tracked. The analysis is re-started after this refinement. Because $ticks$ is not in the precision (the variable is not tracked), the assignment $ticks = ticks + 1$ will not add new information. Hence, no new successors are added, and the analysis stops unrolling the loop. The assume operation $[flag > 0]$ is evaluated to $false$, thus, the error label is not reachable. Finally, the analysis terminates, proving the program correct.

In summary, the crucial effect of this approach is that only relevant variables are tracked in the analysis, while unimportant information is ignored. This greatly reduces the number of abstract states to be visited.

Contributions. We make the following contributions:

- We integrate the concepts of abstraction, CEGAR, and lazy abstraction refinement into explicit-value analysis.
- Inspired by Craig interpolation for predicate analysis, we define a novel interpolation-like approach for discovering relevant variables for the explicit-value domain. This refinement algorithm is completely self-contained, i.e., independent from external libraries such as SMT solvers.
- To further improve the effectiveness and efficiency of the analysis, we design a combination with a predicate analysis based on dynamic precision adjustment [8].
- We provide an open-source implementation of all our concepts and give evidence of the significant improvements by evaluating several approaches on benchmark verification tasks (C programs) from SV-COMP'12.

Related Work. The explicit-state model checker SPIN [21] can verify models of programs written in a language called Promela. For the verification of C programs, tools like MODEX can extract Promela models from C source code. This process requires to give a specification of the abstraction level (user-defined extraction rules), i.e., the information of what should be included in the Promela model. SPIN does not provide lazy-refinement-based CEGAR. JAVA PATHFINDER [18] is an explicit-state model checker for Java programs. There has been work [22] on integrating CEGAR into JAVA PATHFINDER, using an approach different from interpolation.

Dynamic precision adjustment [8] is an approach to fine-tune the precision of combined analyses on-the-fly, i.e., during the analysis run; the precision of one analysis can be increased based on a current situation in another analysis. For example, if an explicit-value analysis stores too many different values for a variable, then the dynamic precision adjustment can remove that variable from the precision of the explicit-value analysis and add a predicate about that variable to the precision of a predicate analysis. This means that the tracking of the variable is "moved" from the explicit to the symbolic domain. One configuration that we present later in Section 3 uses this approach.

The tool DAGGER [17] improves the verification of C programs by applying interpolation-based refinement to octagon and polyhedra domains. To avoid imprecision due to widening in the join-based data-flow analysis, DAGGER replaces the standard widen operator by a so called *interpolated-widen* operator, which increases the precision of the data-flow analysis and thus avoids false alarms. The algorithm VINTA [2] applies interpolation-based refinement to interval-like abstract domains. If the state exploration finds an error path, then VINTA performs a feasibility check using bounded model checking (BMC), and if the error path is infeasible, it computes interpolants. The interpolants are used to refine the invariants that the abstract domain operates on. VINTA requires an SMT solver for feasibility checks and interpolation.

More tools are mentioned in our evaluation section, where we compare (in terms of precision and efficiency) our verifier with verifiers of SV-COMP'12. There is, to the best of our knowledge, no work that integrates abstraction, CEGAR, lazy refinement, and interpolation into explicit-state model checking. We make those techniques available for the explicit-value domain.

2 Background

We use several existing concepts; this section reminds the reader of basic definitions.

Programs, Control-Flow Automata, States. We restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers [1]. The following definitions are taken from previous work [10]: A program is represented by a *control-flow automaton* (CFA). A CFA $A = (L, G)$ consists of a set L of program locations, which model the program counter, and a set $G \subseteq L \times Ops \times L$ of control-flow edges, which model the operations that are executed when control flows from one program location to another. The set of program variables that occur in operations from Ops is denoted by X. A *verification problem* $P = (A, l_0, l_e)$ consists of a CFA A, representing the program, an initial program location $l_0 \in L$, representing the program entry, and a target program location $l_e \in L$, which represents the error.

A *concrete data state* of a program is a variable assignment $cd : X \to \mathbb{Z}$, which assigns to each program variable an integer value. A *concrete state* of a program is a pair (l, cd), where $l \in L$ is a program location and cd is a concrete data state. The set of all concrete states of a program is denoted by $\mathcal{C}$, a subset $r \subseteq \mathcal{C}$ is called *region*. Each edge $g \in G$ defines a labeled transition relation $\xrightarrow{g} \subseteq \mathcal{C} \times \{g\} \times \mathcal{C}$. 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 an edge g with $c \xrightarrow{g} c'$.

An *abstract data state* represents a region of concrete data states, formally defined as abstract variable assignment. An *abstract variable assignment* is a partial function $v : X \to \mathbb{Z} \cup \{\top, \bot\}$, which maps variables in the definition range of function v to integer values or $\top$ or $\bot$. The special value $\top$ is used to represent an unknown value, e.g., resulting from an uninitialized variable or an external function call, and the special value $\bot$ is used to represent no value, i.e., a contradicting variable assignment. We denote the *definition range* for a partial function f as $\text{def}(f) = \{x \mid \exists y : (x, y) \in f\}$, and the *restriction* of a partial function f to a new definition range Y as $f_{|Y} = f \cap (Y \times (\mathbb{Z} \cup \{\top, \bot\}))$. An abstract variable assignment v represents the region $[\![v]\!]$ of all concrete data states cd for which v is valid, formally:

[1] The framework CPACHECKER operates on C programs; non-recursive function calls are supported.

$[\![v]\!] = \{cd \mid \forall x \in \text{def}(v) : cd(x) = v(x)$ or $v(x) = \top\}$. An *abstract state* of a program is a pair (l, v), representing the following set of concrete states: $\{(l, cd) \mid cd \in [\![v]\!]\}$.

Configurable Program Analysis with Dynamic Precision Adjustment. We use the framework of configurable program analysis (CPA) [7], extended by the concept of dynamic precision adjustment [8]. Such a CPA supports adjusting the precision of an analysis during the exploration of the program's abstract state space. A *composite* CPA can control the precision of its component analyses during the verification process, i.e., it can make a component analysis more abstract, and thus more efficient, or it can make a component analysis more precise, and thus more expensive. A CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ consists of (1) an abstract domain D, (2) a set Π of precisions, (3) a transfer relation $\rightsquigarrow$, (4) a merge operator merge, (5) a termination check stop, and (6) a precision adjustment function prec. Based on these components and operators, we can formulate a flexible and customizable reachability algorithm, which is adapted from previous work [7, 12].

Explicit-Value Analysis as CPA. We now define a component CPA that tracks explicit values for program variables. In order to obtain a complete analysis, a composite CPA is constructed that consists of the component CPA for explicit values and another component CPA for tracking the program locations (CPA for location analysis, as previously described [8]). For the composite CPA, the general definitions of the abstract domain, the transfer relation, and the other operators are given in previous work [8]; the composition is done automatically by the framework implementation CPACHECKER.

The *CPA for explicit-value analysis*, which tracks integer values for the variables of a program explicitly, is defined as $\mathbb{C} = (D_\mathbb{C}, \Pi_\mathbb{C}, \rightsquigarrow_\mathbb{C}, \mathsf{merge}_\mathbb{C}, \mathsf{stop}_\mathbb{C}, \mathsf{prec}_\mathbb{C})$ and consists of the following components [8]:

1. The abstract domain $D_\mathbb{C} = (C, \mathcal{V}, [\![\cdot]\!])$ contains the set C of concrete data states, and uses the semi-lattice $\mathcal{V} = (V, \top, \bot, \sqsubseteq, \sqcup)$, which consists of the set $V = (X \to \mathcal{Z})$ of abstract variable assignments, where $\mathcal{Z} = \mathbb{Z} \cup \{\top_\mathcal{Z}, \bot_\mathcal{Z}\}$ induces the flat lattice over the integer values (we write $\mathbb{Z}$ to denote the set of integer values). The top element $\top \in V$, with $\top(x) = \top_\mathcal{Z}$ for all $x \in X$, is the abstract variable assignment that holds no specific value for any variable, and the bottom element $\bot \in V$, with $\bot(x) = \bot_\mathcal{Z}$ for all $x \in X$, is the abstract variable assignment which models that there is no value assignment possible, i.e., a state that cannot be reached in an execution of the program. The partial order $\sqsubseteq \subseteq V \times V$ is defined as $v \sqsubseteq v'$ if for all $x \in X$, we have $v(x) = v'(x)$ or $v(x) = \bot_\mathcal{Z}$ or $v'(x) = \top_\mathcal{Z}$. The join $\sqcup : V \times V \to V$ yields the least upper bound for two variable assignments. The concretization function $[\![\cdot]\!] : V \to 2^C$ assigns to each abstract data state v its meaning, i.e., the set of concrete data states that it represents.

2. The set of precisions $\Pi_\mathbb{C} = 2^X$ is the set of subsets of program variables. A precision $\pi \in \Pi_\mathbb{C}$ specifies a set of variables to be tracked. For example, $\pi = \emptyset$ means that not a single program variable is tracked, and $\pi = X$ means that each and every program variable is tracked.

3. The transfer relation $\rightsquigarrow_{\mathbb{C}}$ has the transfer $v \overset{g}{\rightsquigarrow} (v', \pi)$ if
(1) $g = (\cdot, \texttt{assume}(p), \cdot)$ and for all $x \in X$:

$$v'(x) = \begin{cases} \bot_{\mathcal{Z}} & \text{if } (y, \bot_{\mathcal{Z}}) \in v \text{ for some } y \in X \text{ or the predicate } p_{/v} \text{ is unsatisfiable} \\ c & \text{if } c \text{ is the only satisfying assignment of the predicate } p_{/v} \text{ for variable } x \\ \top_{\mathcal{Z}} & \text{otherwise} \end{cases}$$

where $p_{/v}$ denotes the interpretation of p over variables from X for an abstract variable assignment v, that is, $p_{/v} = p \wedge \bigwedge_{x \in \text{def}(v), v(x) \in \mathbb{Z}} x = v(x) \wedge \neg \exists x \in \text{def}(v) : v(x) = \bot_{\mathcal{Z}}$
or

(2) $g = (\cdot, \texttt{w} := exp, \cdot)$ and for all $x \in X$: $v'(x) = \begin{cases} exp_{/v} & \text{if } x = \texttt{w} \\ v(x) & \text{if } x \in \text{def}(v) \\ \top_{\mathcal{Z}} & \text{otherwise} \end{cases}$

where $exp_{/v}$ denotes the interpretation of an expression exp over variables from X for an abstract value assignment v:

$$exp_{/v} = \begin{cases} \bot_{\mathcal{Z}} & \text{if } (y, \bot_{\mathcal{Z}}) \in v \text{ for some } y \in X \\ \top_{\mathcal{Z}} & \text{if } (y, \top_{\mathcal{Z}}) \in v \text{ or } y \notin \text{def}(v) \text{ for some } y \in X \text{ that occurs in } exp \\ c & \text{otherwise, where expression } exp \text{ evaluates to } c \text{ after replacing each} \\ & \text{occurrence of variable } x \text{ with } x \in \text{def}(v) \text{ by } v(x) \text{ in } exp \end{cases}$$

4. The merge does not combine states when control flow meets: $\mathsf{merge}_{\mathbb{C}}(v, v', \pi) = v'$.

5. The stop operator checks states individually: $\mathsf{stop}_{\mathbb{C}}(v, R, \pi) = (\exists v' \in R : v \sqsubseteq v')$.

6. The precision adjustment function computes a new abstract state with precision, based on the abstract state v and the precision π, by restricting the variable assignment v to those variables that appear in π, formally: $\mathsf{prec}(v, \pi, R) = (v_{|\pi}, \pi)$.

The precision of the analysis controls which program variables are tracked in an abstract state. In other approaches, this information is hard-wired in either the abstract-domain elements or the algorithm itself. The concept of CPA supports different precisions for different abstract states. A simple analysis can start with an initial precision and propagate it to new abstract states, such that the overall analysis uses a globally uniform precision. It is also possible to specify a precision individually per program location, instead of using one global precision. Our refinement approach in the next section will be based on location-specific precisions.

Predicate Analysis as CPA. In a predicate analysis [16], the precision is defined as a set of predicates, and the abstract states track the strongest set of predicates that are fulfilled (cartesian predicate abstraction) or the strongest boolean combination of predicates that is fulfilled (boolean predicate abstraction). This means, the abstraction level of the abstract model is determined by predicates that are tracked in the analysis. Predicate analysis is also implemented as a CPA in the framework CPAchecker, and a detailed description is available [10]. The precision is freely adjustable [8] also in the predicate analysis; we use this feature later in this article for composing a combined analysis.

Lazy Abstraction. The concept of lazy abstraction [20] proposes to refine the abstract states only where necessary along infeasible error paths in order to eliminate those paths. We implemented this using CPAs with dynamic precision adjustment, where the refinement procedure operates on location-specific precisions and the precision-adjustment operator always removes unnecessary information from abstract states.

Algorithm 1. $\mathsf{CPA}(\mathbb{D}, R_0, W_0)$, adapted from [8]

```
Input: CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
       set $R_0 \subseteq (E \times \Pi)$ of abstract states with precision,
       subset $W_0 \subseteq R_0$ of frontier abstract states with precision,
       where $E$ denotes the set of elements of the semi-lattice of $D$
Output: set of reachable abstract states with precision,
        subset of frontier abstract states with precision
Variables: sets reached and waitlist of elements of $E \times \Pi$
  reached := $R_0$; waitlist := $W_0$;
  while waitlist $\neq \emptyset$ do
    choose $(e, \pi)$ from waitlist; remove $(e, \pi)$ from waitlist;
    for each $e'$ with $e \rightsquigarrow (e', \pi)$ do
      $(\hat{e}, \hat{\pi})$ := prec($e'$, $\pi$, reached);          // precision adjustment
      if isTargetState($\hat{e}$) then
        return $(\mathsf{reached} \cup \{(\hat{e}, \hat{\pi})\}, \mathsf{waitlist} \cup \{(\hat{e}, \hat{\pi})\})$;
      for each $(e'', \pi'') \in$ reached do
        $e_{new}$ := merge($\hat{e}$, $e''$, $\hat{\pi}$);             // combine with existing abstract state
        if $e_{new} \neq e''$ then
          waitlist := $(\mathsf{waitlist} \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$;
          reached := $(\mathsf{reached} \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$;
      if $\neg\, \mathsf{stop}(\hat{e}, \{e \mid (e, \cdot) \in \mathsf{reached}\}, \hat{\pi})$ then  // add new abstract state?
        waitlist := waitlist $\cup \{(\hat{e}, \hat{\pi})\}$;     reached := reached $\cup \{(\hat{e}, \hat{\pi})\}$
  return (reached, $\emptyset$);
```

Reachability Algorithm for CPA. Algorithm 1 keeps updating two sets of abstract states with precision: the set reached stores all abstract states with precision that are found to be reachable, and the set waitlist stores all abstract states with precision that are not yet processed, i.e., the frontier. The state exploration starts with choosing and removing an abstract state with precision from the waitlist, and the algorithm considers each abstract successor according to the transfer relation. Next, for the successor, the algorithm adjusts the precision of the successor using the precision adjustment function prec. If the successor is a target state (i.e., a violation of the property is found), then the algorithm terminates, returning the current sets reached and waitlist — possibly as input for a subsequent precision refinement, as shown below (cf. Alg. 2). Otherwise, using the given operator merge, the abstract successor state is combined with each existing abstract state from reached. If the operator merge results in a new abstract state with information added from the new successor (the old abstract state is subsumed) then the old abstract state with precision is replaced by the new abstract state with precision in the sets reached and waitlist. If after the merge step the resulting new abstract state with precision is covered by the set reached, then further exploration of this abstract state is stopped. Otherwise, the abstract state with its precision is added to the set reached and to the set waitlist. Finally, once the set waitlist is empty, the set reached is returned.

Counterexample-Guided Abstraction Refinement (CEGAR). CEGAR [13] is a technique for automatic stepwise refinement of an abstract model. CEGAR is based on three concepts: (1) a *precision*, which determines the current level of abstraction, (2) a *feasibility check*, deciding if an abstract error path is feasible, i.e., if there exists a

Algorithm 2. $\mathsf{CEGAR}(\mathbb{D}, e_0, \pi_0)$

```
Input:  CPA with dynamic precision adjustment $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
        initial abstract state $e_0 \in E$ with precision $\pi_0 \in \Pi$,
        where $E$ denotes the set of elements of the semi-lattice of $D$
Output:  verification result safe or unsafe
Variables: set reached $\subseteq E \times \Pi$, set waitlist $\subseteq E \times \Pi$, error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
  reached := $\{(e_0, \pi_0)\}$; waitlist := $\{(e_0, \pi_0)\}$; $\pi := \pi_0$;
  while $true$ do
    (reached, waitlist) := CPA($\mathbb{D}$, reached, waitlist);
    if waitlist = $\emptyset$ then
      return safe
    else
      $\sigma$ := extractErrorPath(reached);
      if isFeasible($\sigma$) then // error path is feasible: report bug
        return unsafe
      else // error path is not feasible: refine and restart
        $\pi := \pi \cup$ Refine($\sigma$);
        reached := $(e_0, \pi)$; waitlist := $(e_0, \pi)$;
```

corresponding concrete error path, and (3) a *refinement* procedure, which takes as input an infeasible error path and extracts a precision that suffices to instruct the exploration algorithm to not explore the same path again later. Algorithm 2 shows an outline of a generic and simple CEGAR algorithm. The algorithm starts checking a program using a coarse initial *precision* π_0. It uses Alg. 1 for computing the reachable abstract state space, returning the sets reached and waitlist. If the analysis has exhaustively checked all program states and did not reach the error, indicated by an empty set waitlist, then the algorithm terminates and reports that the program is safe. If the algorithm finds an error in the abstract state space, i.e., a counterexample for the given specification, then the exploration algorithm stops and returns the unfinished, incomplete sets reached and waitlist. Now the according abstract error path is extracted from the set reached using procedure extractErrorPath and analyzed for feasibility using the procedure isFeasible as *feasibility check*. If the abstract error path is feasible, meaning there exists a corresponding concrete error path, then this error path represents a violation of the specification and the algorithm terminates, reporting a bug. If the error path is infeasible, i.e., not corresponding to a concrete program path, then the precision was too coarse and needs to be refined. The algorithm extracts certain information from the error path in order to refine the precision based on that information using the procedure Refine for *refinement*, which returns a precision π that makes the analysis strong enough to refute the present infeasible error path in further state-space explorations. The current precision is extended using the precision returned by the refinement procedure and the analysis is restarted with this refined precision. Instead of restarting from the initial sets for reached and waitlist, we can also prune those parts of the abstract reachability graph (ARG) that need to be rediscovered with new precisions, and replace the precision of the leaf nodes in the ARG with the refined precision, and then restart the exploration on the pruned sets (cf. [11] for more details). Our contribution in the next section is to introduce new implementations for the feasibility check as well as for the refinement procedure.

Interpolation. For a pair of formulas φ^- and φ^+ such that $\varphi^- \wedge \varphi^+$ is unsatisfiable, a Craig interpolant ψ is a formula that fulfills the following requirements [15]: (1) the implication $\varphi^- \Rightarrow \psi$ holds, (2) the conjunction $\psi \wedge \varphi^+$ is unsatisfiable, and (3) ψ only contains symbols that occur in both φ^- and φ^+. Such a Craig interpolant is guaranteed to exist for many useful theories, e.g., the theory of linear arithmetic (implemented in SMT solvers). Interpolation-based CEGAR has been proven successful in the predicate domain. However, interpolants from the predicate domain, which consist of formulas, are not useful for the explicit domain. Hence, we need to develop a procedure to compute interpolants for the explicit domain, which we introduce in the following.

3 Refinement-Based Explicit-Value Analysis

The level of abstraction in our explicit-value analysis is determined by the precisions for abstract variable assignments over program variables. The CEGAR-based iterative refinement needs an extraction method to obtain the necessary precision from infeasible error paths. Our novel notion of interpolation for the explicit domain achieves this goal.

Explicit-Value Abstraction. We now introduce some necessary operations on abstract variable assignments, the semantics of operations and paths, and the precision for abstract variable assignments and programs, in order to be able to concisely discuss interpolation for abstract variable assignments and constraint sequences.

The operations *implication* and *conjunction* for abstract variable assignments are defined as follows: implication for v and v': $v \Rightarrow v'$ if $\text{def}(v') \subseteq \text{def}(v)$ and for each variable $x \in \text{def}(v) \cap \text{def}(v')$ we have $v(x) = v'(x)$ or $v(x) = \bot$ or $v'(x) = \top$; conjunction for v and v': for each variable $x \in \text{def}(v) \cup \text{def}(v')$ we have

$$(v \wedge v')(x) = \begin{cases} v(x) & \text{if } x \in \text{def}(v) \text{ and } x \notin \text{def}(v') \\ v'(x) & \text{if } x \notin \text{def}(v) \text{ and } x \in \text{def}(v') \\ v(x) & \text{if } v(x) = v'(x) \\ \bot & \text{if } \top \neq v(x) \neq v'(x) \neq \top \\ \top & \text{otherwise } (v(x) = \top \text{ or } v'(x) = \top) \end{cases}$$

Furthermore we define *contradiction* for an abstract variable assignment v: v is contradicting if there is a variable $x \in \text{def}(v)$ such that $v(x) = \bot$ (which implies $[\![v]\!] = \emptyset$); and *renaming* for v: the abstract variable assignment $v^{x \mapsto y}$, with $y \notin \text{def}(v)$, results from v by renaming variable x to y: $v^{x \mapsto y} = (v \setminus \{(x, v(x))\}) \cup \{(y, v(x))\}$.

The *semantics of an operation* $op \in Ops$ is defined by the strongest post-operator $\mathsf{SP}_{op}(\cdot)$ for abstract variable assignments: given an abstract variable assignment v, $\mathsf{SP}_{op}(v)$ represents the set of data states that are reachable from any of the states in the region represented by v after the execution of op. Formally, given a set X of program variables, an abstract variable assignment v, and an assignment operation $s := exp$, we have $\mathsf{SP}_{s:=exp}(v) = v_{|X \setminus \{s\}} \wedge v_{s:=exp}$ with $v_{s:=exp} = \{(s, exp_{/v})\}$, where $exp_{/v}$ denotes the interpretation of expression exp for the abstract variable assignment v (cf. definition of $exp_{/v}$ in Section 2). That is, the value of variable s is the result of the arithmetic evaluation of expression exp, or $\top$ if not all values in the expression are known, or $\bot$ if no value is possible (an abstract data state in which a variable is assigned to $\bot$ does not represent any concrete data state). Given an abstract variable assignment v and an assume operation $[p]$, we have $\mathsf{SP}_{[p]}(v) = v'$ with for all $x \in X$

we have $v'(x) = \bot$ if $(y, \bot) \in v$ for some variable $x \in X$ or the formula $p_{/v}$ is unsatisfiable, or $v'(x) = c$ if c is the only satisfying assignment of the formula $p_{/v}$ for variable x, or $v'(x) = \top$ in all other cases; the formula $p_{/v}$ is defined as in Section 2.

A *path* σ is a sequence $\langle (op_1, l_1), ..., (op_n, l_n) \rangle$ of pairs of an operation and a location. The path σ is called *program path* if for every i with $1 \leq i \leq n$ there exists a CFA edge $g = (l_{i-1}, op_i, l_i)$ and l_0 is the initial program location, i.e., σ represents a syntactic walk through the CFA. Every path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$ defines a *constraint sequence* $\gamma_\sigma = \langle op_1, ..., op_n \rangle$. The *semantics of a program path* $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$ is defined as the successive application of the strongest post-operator to each operation of the corresponding constraint sequence γ_σ: $\mathsf{SP}_{\gamma_\sigma}(v) = \mathsf{SP}_{op_n}(...\mathsf{SP}_{op_i}(..\mathsf{SP}_{op_1}(v)..)...)$. The set of concrete program states that result from running σ is represented by the pair $(l_n, \mathsf{SP}_{\gamma_\sigma}(v_0))$, where $v_0 = \{\}$ is the initial abstract variable assignment that does not map any variable to a value. A program path σ is *feasible* if $\mathsf{SP}_{\gamma_\sigma}(v_0)$ is not contradicting, i.e., $\mathsf{SP}_{\gamma_\sigma}(v_0)(x) \neq \bot$ for all variables x in def($\mathsf{SP}_{\gamma_\sigma}(v_0)$). A concrete state (l_n, cd_n) is *reachable* from a region r, denoted by $(l_n, cd_n) \in \mathit{Reach}(r)$, if there exists a feasible program path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$ with $(l_0, v_0) \in r$ and $cd_n \in [\![\mathsf{SP}_{\gamma_\sigma}(v_0)]\!]$. A location l is reachable if there exists a concrete state c such that (l, c) is reachable. A program is SAFE if l_e is not reachable.

The *precision for an abstract variable assignment* is a set π of variables. The *explicit-value abstraction* for an abstract variable assignment is an abstract variable assignment that is defined only on variables that are in the precision π. For example, the explicit-value abstraction for the variable assignment $v = \{x \mapsto 2, y \mapsto 5\}$ and the precision $\pi = \{x\}$ is the abstract variable assignment $v^\pi = \{x \mapsto 2\}$.

The *precision for a program* is a function $\Pi : L \to 2^X$, which assigns to each program location a precision for an abstract variable assignment, i.e., a set of variables for which the analysis is instructed to track values. A *lazy explicit-value abstraction* of a program uses different precisions for different abstract states on different program paths in the abstract reachability graph. The explicit-value abstraction for a variable assignment at location l is computed using the precision $\Pi(l)$.

CEGAR for Explicit-Value Model Checking. We now instantiate the three components of the CEGAR technique, i.e., precision, feasibility check, and refinement, for our explicit-value analysis. The precisions that our CEGAR instance uses are the above introduced precisions for a program (which assign to each program location a set of variables), and we start the CEGAR iteration with the empty precision, i.e., $\Pi_{init}(l) = \emptyset$ for each $l \in L$, such that no variable will be tracked.

The feasibility check for a path σ is performed by executing an explicit-value analysis of the path σ using the full precision $\Pi(l) = X$ for all locations l, i.e., all variables will be tracked. This is equivalent to computing $\mathsf{SP}_{\gamma_\sigma}(v_0)$ and check if the result is contradicting, i.e., if there is a variable for which the resulting abstract variable assignment is $\bot$. This feasibility check is extremely efficient, because the path is finite and the strongest post-operations for abstract variable assignments are simple arithmetic evaluations. If the feasibility check reaches the error location l_e, then this error can be reported. If the check does not reach the error location, because of a contradicting

Algorithm 3. $\mathsf{Interpolate}(\gamma^-, \gamma^+)$

Input: two constraint sequences γ^- and γ^+, with $\gamma^- \wedge \gamma^+$ is contradicting
Output: a constraint sequence Γ, which is an interpolant for γ^- and γ^+
Variables: an abstract variable assignment v

```
$v := \mathsf{SP}_{\gamma^-}(\emptyset)$;
for each $x \in \text{def}(v)$ do
    if $\mathsf{SP}_{\gamma^+}(v_{|\text{def}(v) \setminus \{x\}})$ is contradicting then
        $v := v_{|\text{def}(v) \setminus \{x\}}$;        // $x$ is not relevant and should not occur in the interpolant
$\Gamma := \langle\rangle$;                              // start assembling the interpolating constraint sequence
for each $x \in \text{def}(v)$ do
    $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;   // construct an assume constraint for $x$
return $\Gamma$
```

abstract variable assignment, then a refinement is necessary because at least one constraint depends on a variable that was not yet tracked.

We define the last component of the CEGAR technique, the refinement, after we introduced the notion of interpolation for variable assignments and constraint sequences.

Interpolation for Variable Assignments. For each infeasible error path in the above mentioned refinement operation, we need to determine a precision that assigns to each program location on that path the set of program variables that the explicit-value analysis needs to track in order to eliminate that infeasible error path in future explorations. Therefore, we define an interpolant for abstract variable assignments.

An *interpolant* for a pair of abstract variable assignments v^- and v^+, such that $v^- \wedge v^+$ is contradicting, is an abstract variable assignment $\mathcal{V}$ that fulfills the following requirements: (1) the implication $v^- \Rightarrow \mathcal{V}$ holds, (2) the conjunction $\mathcal{V} \wedge v^+$ is contradicting, and (3) $\mathcal{V}$ only contains variables in its definition range which are in the definition ranges of both v^- and v^+ ($\text{def}(\mathcal{V}) \subseteq \text{def}(v^-) \cap \text{def}(v^+)$).

Lemma. For a given pair (v^-, v^+) of abstract variable assignments, such that $v^- \wedge v^+$ is contradicting, an interpolant exists. Such an interpolant can be computed in time $O(m+n)$, where m and n are the sizes of v^- and v^+, respectively.

Proof. The variable assignment $v^-_{|\text{def}(v^+)}$ is an interpolant for the pair (v^-, v^+).
The above-mentioned interpolant that simply results from restricting v^- to the definition range of v^+ (common definition range) is of course not the best interpolant. Interpolation for assignments is a first idea to approach the problem, but since we need to extract interpolants for paths, we next define interpolation for constraint sequences.

Interpolation for Constraint Sequences. A more expressive interpolation is achieved by considering constraint sequences. The *conjunction* $\gamma \wedge \gamma'$ of two constraint sequences $\gamma = \langle op_1, ..., op_n \rangle$ and $\gamma' = \langle op'_1, ..., op'_m \rangle$ is defined as their concatenation, i.e., $\gamma \wedge \gamma' = \langle op_1, ..., op_n, op'_1, ..., op'_m \rangle$, the *implication* of γ and γ' (denoted by $\gamma \Rightarrow \gamma'$) as $\mathsf{SP}_\gamma(v_0) \Rightarrow \mathsf{SP}_{\gamma'}(v_0)$, and γ is *contradicting* if $[\![\mathsf{SP}_\gamma(v_0)]\!] = \emptyset$, with $v_0 = \{\}$.

An *interpolant* for a pair of constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting, is a constraint sequence Γ that fulfills the three requirements: (1) the implication $\gamma^- \Rightarrow \Gamma$ holds, (2) the conjunction $\Gamma \wedge \gamma^+$ is contradicting, and (3) Γ contains in its constraints only variables that occur in the constraints of both γ^- and γ^+.

Algorithm 4. Refine(σ)

```
Input: infeasible error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
Output: precision $\Pi$
Variables: interpolating constraint sequence $\Gamma$
  $\Gamma := \langle\rangle$; $\Pi(l) := \emptyset$, for all program locations $l$;
  for $i := 1$ to $n - 1$ do
    $\gamma^+ := \langle op_{i+1}, ..., op_n \rangle$
    $\Gamma := \mathsf{Interpolate}(\Gamma \wedge op_i, \gamma^+)$      // inductive interpolation
    // extract variables from variable assignment that results from $\Gamma$
    $\Pi(l_i) := \{x | (x, z) \in \mathsf{SP}_\Gamma(\emptyset)$ and $\bot \neq z \neq \top\}$
  return $\Pi$
```

Lemma. For a given pair (γ^-, γ^+) of constraint sequences, such that $\gamma^- \wedge \gamma^+$ is contradicting, an interpolant exists. Such an interpolant is computable in time $O(m \cdot n)$, where m and n are the sizes of γ^- and γ^+, respectively.

Proof. Algorithm Interpolate (Alg. 3) returns an interpolant for two constraint sequences γ^- and γ^+. The algorithm starts with computing the strongest post-condition for γ^- and assigns the result to the abstract variable assignment v, which then may contain up to m variables. Per definition, the strongest post-condition for γ^+ of variable assignment v is contradicting. Next we try to eliminate each variable from v, by testing if removing it from v makes the strongest post-condition for γ^+ of v contradicting (each such test takes n SP steps). If it is contradicting, the variable can be removed. If not, the variable is necessary to prove the contradiction of the two constraint sequences, and thus, should occur in the interpolant. Note that this keeps only variables in v that occur in γ^+ as well. The rest of the algorithm constructs a constraint sequence from the variable assignment, in order to return an interpolating constraint sequence, which fulfills the three requirements of an interpolant. A naive implementation can compute such an interpolant in $O((m + n)^3)$.

Refinement Based on Explicit-Interpolation. The goal of our interpolation-based refinement for explicit-value analysis is to determine a location-specific precision that is strong enough to eliminate an infeasible error path in future explorations. This criterion is fulfilled by the property of interpolants. A second goal is to have a precision that is as weak as possible, by creating interpolants that have a definition range as small as possible, in order to be parsimonious in tracking variables and creating abstract states.

We apply the idea of interpolation for constraint sequences to assemble a precision-extraction algorithm: Algorithm Refine (Alg. 4) takes as input an infeasible program path, and returns a precision for a program. A further requirement is that the procedure computes *inductive* interpolants [6], i.e., each interpolant along the path contains enough information to prove the remaining path infeasible. This is needed in order to ensure that the interpolants at the different locations achieve the goal of providing a precision that eliminates the infeasible error path from further explorations. For every program location l_i along an infeasible error path σ, starting at l_0, we split the constraint sequence of the path into a constraint prefix γ^-, which consists of the constraints from the start location l_0 to l_i, and a constraint suffix γ^+, which consists of the path from the location l_i to l_e. For computing inductive interpolants, we replace the constraint prefix

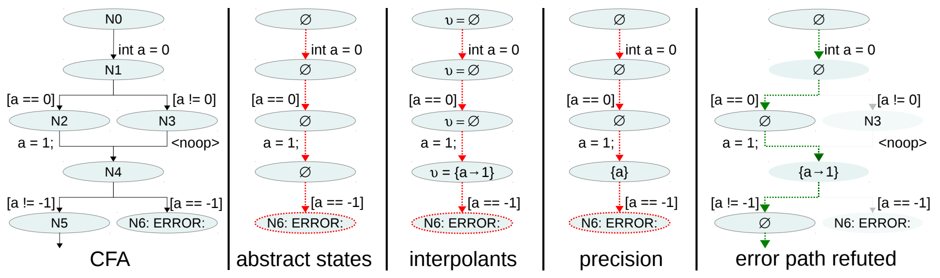

Fig. 1. Illustration of one refinement iteration; simple example CFA, infeasible error path with the abstract states annotated in the nodes (precision empty; nothing tracked), interpolants (i.e., variable assignments), precision extracted from interpolants, abstract states according to new precision after error path is refuted

by the conjunction of the last interpolant and the current constraint. The precision is extracted by computing the abstract variable assignment for the interpolating constraint sequence and assigning the relevant variables as precision for the current location l_i, i.e., the set of all variables that are necessary to be tracked in order to eliminate the error path from future exploration of the state space. This algorithm can be directly plugged-in as refinement routine of the CEGAR algorithm (cf. Alg. 2). Figure 1 illustrates the interpolation process on a simple example.

Auxiliary Predicate Analysis. As an optional further improvement, we implemented a combination with a predicate analysis (cf. [8]): If the explicit-value analysis finds an error path, this path is checked for feasibility in the predicate domain. If feasible, the result is *unsafe* and the error path is reported; if infeasible, the explicit-value domain is not expressive enough to analyze that program path (e.g., due to inequalities). We then ask the predicate analysis to refine its abstraction along that path, which yields a refined predicate precision that eliminates the error path but considering the facts along that path in the (more precise, and more expensive) predicate domain. We need to parsimoniously use this feature because the post-operations of the predicate analysis are much more expensive than the post-operations of the explicit-value analysis. In general, after a refinement step, either the explicit-value precision is refined (preferred) or the predicate precision is refined (only if explicit does not succeed). We also remove variables from the precision in the explicit-value domain if the number of different values on a path exceeds a certain threshold. A later refinement will then add predicates about such variables to the precision in the predicate domain. Note that this refinement-based, parallel composition of explicit-value and predicate analysis is strictly more powerful than a mere parallel product of the two analyses, because the explicit domain tracks exactly what it can efficiently analyze, while the predicate domain takes care of everything else.

4 Experiments

In order to demonstrate that our approach yields a significant improvement of verification efficiency and effectiveness, we implemented our algorithms and compared our new techniques to existing tools for software verification. We show that the application of abstraction, CEGAR, and interpolation to the explicit-value domain considerably improves the number of solved instances and the run time. Combinations of the new explicit-value analysis with a predicate-based analysis can further increase the number of solved instances. All experiments were performed using rules and hardware identical to SV-COMP'12 [5], restricting each verification task to the same run time and memory limits (900 s, 15 GB), such that our results are comparable to all results obtained there.

Compared Verification Approaches. For presentation, we restrict the comparison of our new approach to the SV-COMP'12 participants BLAST [6], SATABS [14], and the competition winner CPA-MEMO [23], all of which are based on predicate abstraction and CEGAR. Furthermore, to investigate performance differences in the same tool environment, we also compare with different configurations of CPACHECKER. BLAST won the category "DeviceDrivers64" in the SV-COMP'12, and got bronze in another category. SATABS got silver in the categories "SystemC" and "Concurrency", and bronze in another category. CPA-MEMO won the category "Overall", got silver in two more categories, and bronze in another category. We implemented our new concepts in CPACHECKER [9], a software-verification framework based on CPA. We compare with the existing explicit-value analysis (without abstraction, CEGAR, and interpolation) and with the existing predicate analysis [10]. We used the trunk version of CPACHECKER[2] in revision 6615.

Verification Tasks. For the evaluation of our approach, we use all SV-COMP'12[3] verification tasks that do not involve concurrency properties (all categories except category "Concurrency"). All obtained experimental data as well as the tool implementation are available at `http://www.sosy-lab.org/~dbeyer/cpa-explicit`.

Quality Measures. We compare the verification results of all verification approaches based on three measures for verification quality: First, we take the run time, in seconds, of the verification runs to measure the *efficiency* of an approach. Obviously, the lower the run time, the better the tool. Second, we use the number of correctly solved instances of verification tasks to measure the *effectiveness* of an approach. The more instances a tool can solve, the more powerful the analysis is. Third, and most importantly, we use the scoring schema of the SV-COMP'12 as indicator for the quality of an approach. The scoring schema implements a community-agreed weighting schema, namely, that it is more difficult to prove a program correct compared to finding a bug and that a wrong answer should be penalized with double the scores that a correct answer would have achieved. For a full discussion of the official rules and benchmarks of the SV-COMP'12, we refer to the competition report [5]. Besides the data tables, we use plots of quantile functions [5] for visualizing the number of solved instances and the verification time. The quantile function for one approach contains all pairs (x, y) such that the maximum run time of the x fastest results is y. We use a logarithmic scale for the time range from 1 s to 1000 s and a linear scale for the time range between 0 s and 1 s.

Improvements of Explicit-Value Analysis. In the first evaluation, we compare two different configurations of the explicit-value analysis: CPA-EXPL refers to the existing implementation of a standard explicit-value analysis without abstraction and refinement, and CPA-EXPL$_{\text{itp}}$ refers to the new approach, which implements abstraction, CEGAR, and

[2] `http://cpachecker.sosy-lab.org`
[3] `http://sv-comp.sosy-lab.org/2012`

Table 1. Comparison with purely explicit, non-CEGAR approach

Category	CPA-EXPL			CPA-EXPLitp		
	points	solved	time	points	solved	time
ControlFlowInt	124	81	8400	123	79	780
DeviceDrivers	53	37	63	53	37	69
DeviceDrivers64	5	5	660	33	19	200
HeapManipul	1	3	5.5	1	3	5.8
SystemC	34	26	1600	34	26	1500
Overall	217	152	11000	**244**	**164**	**2500**

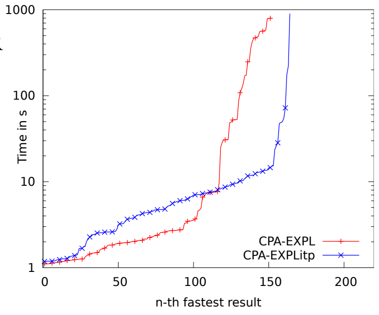

Fig. 2. Purely explicit analyses

Table 2. Comparison with predicate-based configurations

Category	CPA-PRED			CPA-EXPLitp			CPA-EXPL-PRED			CPA-EXPLitp-PRED		
	score	solved	time	score	solved	time	score	solved	time	score	solved	time
ControlFlowInt	103	70	2500	123	79	780	131	85	2600	141	91	830
DeviceDrivers	71	46	80	53	37	69	71	46	82	71	46	87
DeviceDrivers64	33	24	2700	33	19	200	10	11	1100	37	24	980
HeapManipul	8	6	12	1	3	5.8	6	5	11	8	6	12
SystemC	22	17	1900	34	26	1500	62	45	1500	61	44	3700
Overall	237	163	7100	244	164	**2500**	280	192	5300	**318**	**211**	5600

Table 3. Comparison with three existing tools

Category	BLAST			SATABS			CPA-MEMO			CPA-EXPLitp-PRED		
	score	solved	time	score	solved	time	score	solved	time	score	solved	time
ControlFlowInt	71	51	9900	75	47	5400	140	91	3200	141	91	830
DeviceDrivers	72	51	30	71	43	140	51	46	93	71	46	87
DeviceDrivers64	55	33	1400	32	17	3200	49	33	500	37	24	980
HeapManipul	–	–	–	–	–	–	4	9	16	8	6	12
SystemC	33	23	4000	57	40	5000	36	30	450	61	44	3700
Overall	231	158	15000	235	147	14000	280	209	**4300**	**318**	**211**	5600

interpolation. Table 1 and Fig. 2 show that the new approach uses less time, solves more instances, and obtains more points in the SV-COMP'12 scoring schema.

Improvements of Combination with Predicate Analysis. In the second evaluation, we compare the refinement-based explicit analysis against a standard predicate analysis and to a combination of predicate analysis with CPA-EXPL and CPA-EXPLitp, respectively: CPA-PRED refers to a standard predicate analysis that CPACHECKER offers (ABE-lf, [10]), CPA-EXPL-PRED refers to the combination of CPA-EXPL and CPA-PRED, and CPA-EXPLitp-PRED refers to the combination of CPA-EXPLitp and CPA-PRED. Table 2 and Fig. 3 show that the new combination approach outperforms the approaches CPA-PRED and CPA-EXPLitp in terms of solved instances and score. The comparison with column CPA-EXPL-PRED is interesting because it shows that the combination of the two analyses is an improvement even without refinement in the explicit-value analysis, but switching on the refinement in both domains makes the new combination significantly more effective.

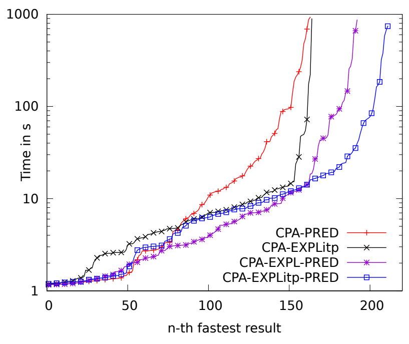

Fig. 3. Comparison with predicate-based configs

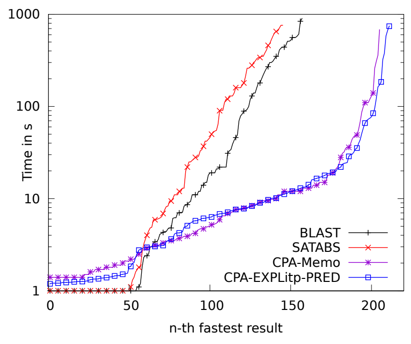

Fig. 4. Comparison with three existing tools

Comparison with State-of-the-Art Verifiers. In the third evaluation, we compare our new combination approach with three established tools: BLAST refers to the standard BLAST configuration that participated in the SV-COMP'12, SATABS also refers to the respective standard configuration, CPA-MEMO refers to a special predicate abstraction that is based on block-abstraction memoization, and CPA-EXPL$_{itp}$-PRED refers to our novel approach, which combines a predicate analysis (CPA-PRED) with the new explicit-value analysis that is based on abstraction, CEGAR, and interpolation (CPA-EXPL$_{itp}$). Table 3 and Fig. 4 show that the new approach outperforms BLAST and SATABS by consuming considerably less verification time, more solved instances, and a better score. Even compared to the SV-COMP'12 winner, CPA-MEMO, our new approach scores higher. It is interesting to observe that the difference in scores is much higher than the difference in solved instances: this means CPA-MEMO had many incorrect verification results, which in turn shows that our new combination is significantly more precise.

5 Conclusion

The surprising insight of this work is that it is possible to achieve —without using sophisticated SMT-solvers during the abstraction refinement— a performance and precision that can compete with the world's leading symbolic model checkers, which are based on SMT-based predicate abstraction. We achieved this by incorporating the ideas of abstraction, CEGAR, lazy abstraction refinement, and interpolation into a simple, standard explicit-value analysis. We further improved the performance and precision by combining our refinement-based explicit-value analysis with a predicate analysis, in order to benefit from the complementary advantages of the methods. The combination analysis dynamically adjusts the precision [8] for an optimal trade-off between the precision of the explicit analysis and the precision of the auxiliary predicate analysis. This combination out-performs state-of-the-art model checkers, witnessed by a thorough comparison on a standardized set of benchmarks.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig Interpretation. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 300–316. Springer, Heidelberg (2012)

3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
4. Ball, T., Rajamani, S.K.: The Slam project: Debugging system software via static analysis. In: Proc. POPL, pp. 1–3. ACM (2002)
5. 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)
6. 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)
7. 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)
8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008)
9. 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)
10. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
11. Beyer, D., Löwe, S.: Explicit-value analysis based on CEGAR and interpolation. Technical Report MIP-1205, University of Passau / ArXiv 1212.6542 (December 2012)
12. Beyer, D., Wendler, P.: Algorithms for software model checking: Predicate abstraction vs. IMPACT. In: Proc. FMCAD, pp. 106–113. FMCAD (2012)
13. 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)
14. Clarke, E., Kröning, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
15. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. 22(3), 250–268 (1957)
16. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
17. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically Refining Abstract Interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
18. Havelund, K., Pressburger, T.: Model checking Java programs using JAVA PATHFINDER. Int. J. Softw. Tools Technol. Transfer 2(4), 366–381 (2000)
19. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL, pp. 232–244. ACM (2004)
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002)
21. Holzmann, G.J.: The SPIN model checker. IEEE Trans. Softw. Eng. 23(5), 279–295 (1997)
22. Păsăreanu, C.S., Dwyer, M.B., Visser, W.: Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 284–298. Springer, Heidelberg (2001)
23. Wonisch, D.: Block Abstraction Memoization for CPACHECKER. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 531–533. Springer, Heidelberg (2012)