Algorithms for Software Model Checking: Predicate Abstraction vs. IMPACT

Dirk Beyer
University of Passau, Germany

Philipp Wendler
University of Passau, Germany

***Abstract*—CEGAR, SMT solving, and Craig interpolation are successful approaches for software model checking. We compare two of the most important algorithms that are based on these techniques: lazy predicate abstraction (as in BLAST) and lazy abstraction with interpolants (as in IMPACT). We unify the algorithms formally (by expressing both in the CPA framework) as well as in practice (by implementing them in the same tool). This allows us to flexibly experiment with new configurations and gain new insights, both about their most important differences and commonalities, as well as about their performance characteristics. We show that the essential contribution of the IMPACT algorithm is the reduction of the number of refinements, and compare this to another approach for reducing refinement effort: adjustable-block encoding (ABE).**

***Index Terms*—Formal Verification, Software Model Checking, Predicate Abstraction, Lazy Abstraction, Refinement Techniques, Interpolation, Large-Block Encoding**

I. INTRODUCTION

Software model checking has been successful for improving the quality of computer programs [4]. Several fundamental concepts were invented in the last decade which made it possible to scale the technology from tiny examples to real programs. Predicate abstraction [16] with counterexample-guided abstraction refinement (CEGAR) [14] and lazy abstraction [19] is one such technique. It was made popular by the tools SLAM [6] and BLAST [9], and is implemented in a number of other tools. Lazy abstraction with interpolants [20] is another approach, which is implemented in the tools IMPACT, WOLVERINE [24], and UFO [3]. More than half of the participants in the first competition on software verification [7], and almost all of those that are not based on bounded model checking, use one of these two concepts. Thus, we are interested in comparing the two concepts with each other, identifying their essential differences, and potentially learning new insights from them.

The contribution of our work is to systematically compare the two approaches. First, we re-implemented the IMPACT algorithm within the CPACHECKER framework. This is necessary in order to compare predicate abstraction with the IMPACT algorithm in the same framework: with the same parser frontend, SMT solver, and run-time environment. This verifies that our re-implementation shows all known characteristics in the comparison. Second, we present a unifying framework for predicate-based software model checking with an algorithm that can be configured (parametrized) such that it works like BLAST's predicate abstraction or IMPACT's approach. We show that the framework causes almost no overhead and the algorithms —when expressed in our framework— perform similarly to their original versions.

Now, we can conceptually and experimentally identify the differences of the algorithms. A performance comparison of our implementations of both algorithms (in the unified framework) shows that the key advantage of the IMPACT algorithm is the *forced covering* optimization that was presented by McMillan together with the algorithm [20]. This optimization effectively reduces the number of refinements and leads to a significant performance boost. However, without this optimization IMPACT does not perform better than predicate abstraction.

Another technique that has been shown to effectively reduce the number of refinements is adjustable-block encoding (ABE) [12] (a generalization of large-block encoding [8]), which was originally presented for predicate abstraction. We do not only compare the IMPACT algorithm to predicate abstraction with ABE, but also experiment with the combination of the IMPACT algorithm and ABE.

Availability of Data and Tools. We implemented all presented approaches (where not already existing) in the open-source verification framework CPACHECKER [11]. All experiments are based on publicly available benchmark programs from the last competition on software verification [7]. Our extensions of CPACHECKER are available under the Apache 2.0 license in the project repository via `http://cpachecker.sosy-lab.org`. Tables with our detailed results, as well as all benchmark programs, the configurations files, scripts, and a ready-to-run version of CPACHECKER are available on the supplementary webpage `http://www.sosy-lab.org/~dbeyer/cpa-uni`.

Related Work. A different approach to combine predicate abstraction and the IMPACT algorithm was presented by Albarghouthi et al. [1]. Their algorithm is similar to the IMPACT algorithm, but optionally computes an abstraction using predicates from previous refinements when creating new abstract states, instead of always setting these states to $true$. Furthermore, this approach represents the program counter symbolically (not explicitly), and does a single refinement for all error paths after the control-flow graph (CFA) has been completely unrolled into an abstract reachability graph (ARG) (instead of doing a separate refinement whenever a path to the error location was found). McMillan presented an application of the IMPACT principle to testing [21] and similarly proposed computing predicate abstractions in order to speed up the convergence of the algorithm.

978-0-9835678-2-0/12/$31.00 ©2012 FMCAD Inc.

Ermis et al. presented a technique for software verification which is also based on interpolation [15]. Instead of unrolling the CFA into an ARG by iteratively creating new abstract states at a frontier, they start with a path to the error location in the CFA and split all nodes along this path into two nodes, labeling one with the interpolant computed for this node, and the other with its negation [13]. Afterwards, all transitions between the newly created nodes and their neighbors are checked for feasibility and removed if appropriate. This is continued until no infeasible path to the error location is left. The authors compared their algorithm with the IMPACT algorithm, and applied large-block encoding to it. Complementing this work, we compare IMPACT with predicate abstraction. Ermis et al. support programs in the programming language Boogie and the tool cannot be directly applied to C benchmarks.

Extensions of the IMPACT approach have also been presented by Heizmann et al. [17] and Albarghouthi et al. [2]. These works add support for recursive programs.

II. BACKGROUND

We briefly provide some basic notions and concepts from the literature [9], and describe the two algorithms.

Programs. 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] A *program* is represented by a *control-flow automaton* (CFA), which consists of a set L of program locations (models the program counter l), an initial program location l_0, 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 of program variables that occur in operations from Ops is denoted by X. A *concrete state* of a program is a variable assignment $c : X \cup \{l\} \to \mathbb{Z}$ that assigns to each variable an integer value. The set of all concrete states of a program is denoted by C. A set $r \subseteq C$ of concrete states is called a *region*. Each edge $g \in G$ defines a (labeled) transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The complete transition relation $\to$ is the union over all control-flow edges: $\to = \bigcup_{g \in G} \xrightarrow{g}$. We write $c \xrightarrow{g} c'$ if $(c, g, c') \in \to$, and $c \to c'$ if there exists a g with $c \xrightarrow{g} c'$. A concrete state c_n is *reachable* from a region r, denoted by $c_n \in Reach(r)$, if there exists a sequence of concrete states $\langle c_0, c_1, \ldots, c_n \rangle$ such that $c_0 \in r$ and for all $1 \leq i \leq n$, we have $c_{i-1} \xrightarrow{g} c_i$.

Lazy Predicate Abstraction. Predicate abstraction in combination with CEGAR and lazy abstraction is a forward reachability analysis that unrolls the CFA into an abstract reachability graph (ARG) until a fixed point is reached. Abstract states are represented using predicates over program variables from a given set (the *precision*), which is initially empty. An abstract state is created by computing a boolean combination of these predicates that over-approximates the reachable concrete states. This abstraction computation is done using an SMT solver. When an abstract state that belongs to the error location is discovered, the concrete program path that leads to this state is reconstructed from the ARG and checked for feasibility. If the concrete path is infeasible, the current counterexample is said to be spurious, and the precision of the analysis needs to be refined in order to rule out this counterexample. This is done by computing a Craig interpolant [18] for each location on the path. The predicates contained in these interpolants are then added to the precision, and the analysis is restarted. This guarantees that all necessary predicates for proving program safety will be automatically discovered. For improved performance, the previously computed ARG is not completely deleted after refinement, but only those parts that need to be, are re-computed. Furthermore, the new predicates will not be used globally for all abstraction computations, but only in the part of the ARG and only at those locations of the CFA, for which they are relevant. The analysis terminates if either a non-spurious counterexample is found, or a fixed point is reached during unrolling the ARG (in which case the program is safe). In order to speed up the coverage checks between abstract states (which are necessary for determining whether the fixed point was reached), binary decision diagrams (BDDs) are used for representing the abstract states. This approach corresponds, e.g., to what is implemented in BLAST.

Lazy Predicate Abstraction with Adjustable Block-Encoding. Adjustable block-encoding (ABE) [12] aims at improving the performance of predicate abstraction by reducing the number of abstraction computations and refinements. It does not compute an abstraction for each new abstract state, but instead it groups abstract states into blocks and computes abstractions only once per block (at the end). Abstract states are now tuples of an abstract-state formula and a concrete path formula. The path formula of any abstract state always represents a set of concrete paths from the block entry to the location of this state. When a new state is created, the strongest post-condition of the previous state and the current edge is created and used as the concrete path formula. The abstract-state formula is copied from the previous state. If there exists already an abstract state with the same location inside the same block, both states are merged into one state by taking the disjunction of their path formulas. Only at the block end, an abstraction of the conjunction of the abstract-state formula and the concrete path formula of the current state is computed and used as the new abstract-state formula. The path formula is reset to $true$ at the block end. ABE does not only reduce the number of abstraction computations, but also the number of coverage checks (which are only done at block ends), and the size of the ARG (due to merging of abstract states). The latter is the reason for a vastly reduced number of refinements. During refinement, interpolants are computed only for those abstract states at the block ends, because only for those states, predicates are needed for computing abstractions.

The block size can be freely chosen in ABE and does not need to be statically fixed (as in LBE [8]). If the block size is restricted to one single CFA edge (we name this *single-block encoding*, or SBE), an abstraction is computed

[1] Our implementation is based on CPACHECKER [11], which supports C programs in the CIL [22] subset of C and interprocedural program analysis.

for every new abstract state and the analysis behaves exactly like predicate abstraction in BLAST. Experiments have shown that for a good performance, the program structure should be taken into account when defining the block encodings. A good configuration is for example to define block ends at loop head locations of the program (ABE-Loops), such that the blocks will be the largest loop-free subgraphs of the CFA. Another suitable configuration with somewhat smaller blocks is to define block ends not only at loop heads but also at function entry and exit points (ABE-LF). This configuration is similar to large-block encoding [8]. ABE is implemented, for example, in CPACHECKER.

IMPACT *(Lazy Abstraction with Interpolants).* The IMPACT algorithm [20] similarly creates an unwinding of the CFA. However, it never performs abstraction computations, and instead initializes all new abstract-state formulas to $true$. This is similar to how predicate-abstraction algorithms work while the precision is still empty.

The algorithm consists of three basic steps, which are applied until no further change can be made. In theory, the steps can be executed in any order, but the right strategy is crucial for good performance. The steps are:

Expand(e). If the state e has no successors (i.e., it is a sink in the ARG) and is not covered, create the successor states using $true$ as their initial state formula, and add them to the ARG.

Refine(e). If e is an abstract state at the error location with a state formula different from $false$, compute inductive interpolants for the path from the ARG root to this state. For each state along this path, the state formula is strengthened by conjunctively adding the corresponding interpolant, and the state is marked as not covered. If the error path is infeasible, the state formula of the state at the error location is guaranteed to be $false$ (which marks unreachable states) after this step.

Cover(e_1, e_2). In this step, a state e_1 is marked as *covered* by another state e_2 if the following properties hold:

- e_2 is (and all of its ancestors are) not covered,
- both states belong to the same program location,
- the state formula of e_1 implies the one of e_2, and
- e_1 is not an ancestor of e_2.

If e_1 gets marked as covered, then (1) all states that are covered by e_1 or e_1's children are uncovered, and (2) all children of e_1 are implicitly considered as covered. Note that covered states never cover any other states themselves, i.e., no chains of coverage exist. In order to prevent an infinite loop of coverings and uncoverings, the step *Cover* may be applied only to pairs (e_1, e_2) where e_1 was created after e_2 (only older states can cover newer states, not vice versa).

The application order of the steps as proposed by McMillan is to expand nodes in a depth-first search. During the search, he keeps the invariant that the currently being expanded state and all its ancestors are not coverable by any other state (otherwise the current state would not need to be expanded). As soon as a state is found that belongs to the error location, the refinement procedure is run for this state. After a successful refinement, the invariant that no state on the path from the ARG root to the current state is coverable, is re-established by trying to cover all these states. (This can be optimized by checking only those states that have been strengthened during refinement.) This algorithm corresponds to the core algorithm of IMPACT, as presented by McMillan [20]. It is not available in IMPACT without the following optimization.

IMPACT *with Forced Covering.* When a new state is created in the IMPACT algorithm, its state formula is always $true$ and thus it can only be covered by another state at the same location with the same state formula. However, after some refinements, most states are expected to have stronger formulas, and thus coverage is unlikely, causing a large number of expansions and abstract states. As an optimization, one can try to strengthen the state formula of a new state such that this state can be covered by an existing state at the same location. This is called *forced covering*. In order to forcefully cover a state e_1 by another state e_2, the path from the nearest common ancestor of both states to e_1 is considered. If it can be proven that the state formula of e_2 holds at the location of e_1 after following this path from the nearest common ancestor, the state formula of e_2 can be set as the state formula of e_1. Thus, e_1 is immediately covered by e_2. Additionally, the states along the path from the nearest common ancestor to e_1 are strengthened by computing Craig interpolants for this path. This corresponds to the algorithm used for the benchmarks in the IMPACT article and to the tool implementation [20].

III. UNIFYING ALGORITHM

We formalize our unifying algorithm using the framework of *configurable program analysis* (CPA) [10]. A CPA specifies —independently of the analysis algorithm— the abstract domain and a set of operations that control the program analysis. Such a CPA can be plugged in as a component into the software-verification framework without the need to work on program parsers, exploration algorithms, and their general data structures. A CPA $\mathbb{C} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$ (which computes abstract successor states), a merge operator merge (which specifies if and how to merge abstract states when control flow meets), and a stop operator stop (which determines whether an abstract state is covered by another abstract state). The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete states, a semi-lattice $\mathcal{E}$ over abstract-domain elements (i.e., abstract states), and a concretization function that maps each abstract-domain element to the represented set of concrete states.

Using this framework, program analyses can be composed of several component CPAs. For example, we have defined and implemented separate CPAs for tracking the program counter, the call stack, and the successor-predecessor relationship of the ARG. Thus, we do not need to specify these aspects when defining a new core analysis.

Analysis Algorithm. We use the CPA algorithm for reachability analysis, which gets as input a CPA and two sets of abstract states: one is the set R_0 ($\mathsf{reached}$) of reachable abstract states, and one is the set W_0 ($\mathsf{waitlist}$) of abstract states that

the algorithm is told to process next. The algorithm loops until the set waitlist is empty (all abstract states completely processed) and returns the two sets reached and waitlist. In each iteration, the algorithm takes one state e from the waitlist, computes all abstract successors and processes each of them. The algorithm checks if there is an existing abstract state in the set of reached states with which the new state is to be merged (e.g., at join points where control flow meets after completed branching). If this is the case, then the new, merged abstract state is substituted for the existing abstract state in both sets reached and waitlist. The stop operator ensures that a new abstract state is inserted into the work sets only if this is needed, i.e., the state is not already covered by a state in the set reached.

In order to be able to use CEGAR, we modify this existing CPA algorithm such that it terminates whenever a target state (a state at the error location) is encountered. First, we run the algorithm with singleton sets containing the initial state as input. If the algorithm terminates with a non-empty waitlist (target state found), we start the refinement procedure, which may modify both sets. Then, if the refinement was successful (i.e., the counterexample was infeasible), we run the CPA algorithm again with the modified sets as input. The analysis terminates if either the CPA algorithm finishes due to an empty waitlist ('safe'), or the refinement procedure determines that a feasible error path was found ('bug').

Another modification of the algorithm is necessary to support forced covering. We define a new operator fcover, which is called before an abstract state is going to be expanded. It takes as input the current state and the set of reached states, and returns a new set of reached states. This operator may change the set reached only by strengthening some states if this leads to the current state being covered afterwards. If the current state is still in the set reached afterwards (i.e., was not replaced by a strengthened version), then this state is explored, otherwise we continue with the next state from the waitlist.

The modified algorithm is shown as Algorithm 1. Our changes can be seen in lines 4–5 and 17–18. The shortness and simplicity of these modifications show that using the CPA framework as basis for new approaches is a good idea.

Configurable Predicate Analysis. We use a previous definition of a CPA for predicate abstraction with ABE [12] and configure it. The CPA for predicate analysis $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge, and a stop operator stop, which are defined as follows. (Given a program $P = (L, l_0, G)$, we use X for denoting the set of program variables occurring in P, $\mathcal{P}$ for the set of quantifier-free predicates over variables from X, and $\Pi : L \to 2^{\mathcal{P}}$ for the precision of the predicate abstraction.) Note that this CPA is expected to be used in conjunction with separate CPAs for abstract domains like program counter and call-stack tracking.

1. The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ is a tuple that consists of a set C of concrete states, a semi-lattice $\mathcal{E} = (E, \top, \sqsubseteq, \sqcup)$, and a concretization function $[\![\cdot]\!] : E \to C$. The lattice

Algorithm 1 *CPA*$_{\mathsf{fcover}}(\mathbb{D}, R_0, W_0)$

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

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

elements $e \in E$ (or abstract states) are tuples $(\psi, l^{\psi}, \varphi) \in (\mathcal{P} \times (L \cup \{l_{\top}\}) \times \mathcal{P})$, where the *state formula* ψ is a boolean combination of predicates that occur in $\Pi(l^{\psi})$, l^{ψ} is the location at which ψ was computed, and φ is a disjunctive *path formula* representing some or all concrete paths from l^{ψ} to the location of state e. The top element of the lattice is the abstract state $\top = (true, l_{\top}, true)$. The partial order $\sqsubseteq \subseteq E \times E$ is defined such that for any two elements $e_1 = (\psi_1, l^{\psi}_1, \varphi_1)$ and $e_2 = (\psi_2, l^{\psi}_2, \varphi_2)$ from E the following holds:
$e_1 \sqsubseteq e_2 \Leftrightarrow (e_2 = \top) \vee \left((l^{\psi}_1 = l^{\psi}_2) \wedge (\psi_1 \wedge \varphi_1 \Rightarrow \psi_2 \wedge \varphi_2)\right)$
The join operator $\sqcup : E \times E \to E$ yields the least upper bound of the two operands, according to the partial order.

2. The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E$ contains all tuples (e, g, e') with $e = (\psi, l^{\psi}, \varphi)$, $e' = (\psi', l^{\psi'}, \varphi')$ and $g = (l, op, l')$ for which the following holds:

$$\begin{cases} (\varphi' = true) \wedge \left(\psi' = (\mathsf{SP}_{op}(\varphi) \wedge \psi)^{\Pi(l')}\right) \wedge (l^{\psi'} = l') & \text{if } \mathsf{blk}(e, g) \\ (\varphi' = \mathsf{SP}_{op}(\varphi)) \wedge (\psi' = \psi) \wedge (l^{\psi'} = l^{\psi}) & \text{otherwise} \end{cases}$$

The 'mode' of the transfer relation, i.e., whether to compute an abstraction, is determined by a block-adjustment operator $\mathsf{blk} : E \times G \to \mathbb{B}$, which is given as parameter to the analysis. Inside each block (the second case) the successor states are created by purely syntactically assembling the strongest post-condition SP of the program code attached to the current edge. At the end of the current block (the first case), an *abstraction state* is created. For such a state, the path formula is reset to $true$, and the state formula is set to the result of an abstraction computation $(\cdot)^{\Pi(\cdot)}$ using the path formula and previous state formula as input. Thus, the choice of blk determines the block-encoding (i.e., how much to collect in the path formula before

abstraction). The precision of the predicate abstraction can vary between program locations (parsimonious precision [9]).

3. The *merge operator* $\mathsf{merge} : E \times E \to E$ for two abstract states $e_1 = (\psi_1, l^{\psi}_1, \varphi_1)$ and $e_2 = (\psi_2, l^{\psi}_2, \varphi_2)$ is defined as follows: $\mathsf{merge}(e_1, e_2) =$

$$\begin{cases} (\psi_2, l^{\psi}_2, \varphi_1 \vee \varphi_2) & \text{if } (\psi_1 = \psi_2) \wedge (l^{\psi}_1 = l^{\psi}_2) \\ e_2 & \text{otherwise} \end{cases}$$

This operator combines the two abstract states using a disjunctive path formula, if the abstraction formulas are equal and were computed at the same program location (i.e., if they belong to the same block).

4. The *stop operator* $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ checks if e is covered by a state in the set reached: $\mathsf{stop}(e, R) = \exists e' \in R : (e \sqsubseteq e')$

Our refinement procedure first reconstructs all program paths to the error state by traversing the ARG, and creates a concrete path formula for them. This formula is checked for satisfiability using an SMT solver. If it is satisfiable, a feasible error path was found. Otherwise we split the formula, such that one formula is for one block. This way the cut points exactly match the abstraction states in the ARG. We query the solver to produce an inductive Craig interpolant for each cut point.

Configuration / Instantiation. This framework can now be configured to behave similar to the lazy predicate abstraction of BLAST as well as the lazy abstraction with interpolants algorithm of IMPACT by defining the following three items: (1) how the state formula ψ of each abstract state $(\psi, l^{\psi}, \varphi)$ is represented and how the abstraction computation $(\cdot)^{\Pi(\cdot)}$ is defined, (2) how the partial order $\sqsubseteq$ of the lattice is implemented, and (3) how the interpolants are used to modify the ARG during refinement.

The configuration for BLAST-like lazy predicate abstraction works as follows: (1) Each state formula is represented by a binary decision diagram (BDD). It is computed by taking either the cartesian or the boolean abstraction [5] of the conjunction of the previous abstract-state formula and concrete path formula. Which abstraction mechanism is chosen needs to depend on the block size, to achieve a reasonable performance [8]. (2) Coverage checks are done by checking the entailment of the BDDs that represent the state formulas of the two abstraction states for which coverage is checked. (The path formulas of such states are always equal to $true$ and thus need not be considered.) (3) During refinement, the obtained interpolants are split into their basic atoms, and a predicate is created for each of these atoms. All those predicates are added to the precision for the program location for which the interpolant was computed. The first abstraction state in the paths to the error state, for which a new predicate was found, is identified. This state and all states in the ARG that are reachable from it are removed from the ARG (and from the sets reached and waitlist). Its predecessor state is re-added to the waitlist with the new precision. States that were covered by one of the removed states are also re-added to the waitlist.

To configure the framework as IMPACT algorithm, we use the following setup: (1) State formulas are represented by symbolic formulas. All abstraction states have $true$ as their initial state formula, and $\varphi^{\Pi(l)} = true$ for all formulas φ and locations l. (2) Coverage checks are done by querying an SMT solver whether the implication of the state formulas of the two states holds. (3) After the interpolants are computed during refinement, we conjunct them to the state formulas of the abstract states to which they belong. If a state is strengthened (i.e., the interpolants actually added a conjunct to the state formula), we need to re-check all coverage relations of this state. If a previously covered state is now uncovered, we re-add all sink states in the subgraph of the ARG that starts with this state, to the waitlist. We also check each of the strengthened states whether it is now covered by any other state at the same location. If this is successful, we mark the subgraph that starts with that state as covered and remove all leafs therein from the waitlist (we do not need to expand covered states). The only change to the set reached is the removal of all states whose state formula is $false$ and their successors. It is guaranteed that this is the case for the error state (if the error path is infeasible). This refinement procedure is similar to the function REFINE in the original presentation of the IMPACT algorithm [20].

One last configuration option of our framework is the choice of the fcover operator. For the IMPACT algorithm, we may decide to use interpolation-based forced covering. For predicate abstraction, we use an implementation that always returns the set of reached states that was given as parameter (i.e., no change).

This unification makes the essential differences between these two algorithms explicit and removes those differences that have no impact on the performance. We show in our experimental evaluation in Sect. IV that our version of the IMPACT algorithm has similar performance to the original one.

Discussion. Now that we have identified the important differences, we can evaluate them and discuss their meaning. One main difference is that lazy predicate abstraction computes costly abstractions in order to have cheap coverage checks later on. This is an eager technique: computing effort is spent ahead, not knowing whether this will actually pay off. For example, along a long path within a single loop we might compute abstractions for every state, but check coverage only for the states at the loop head. On the other hand, the IMPACT algorithm delays all computation effort until it is actually needed, which means that whenever some information is needed about a state, a costly SMT-solver query is needed.

A further difference is how the coverage relationship is determined. In order to find as much coverage situations as possible and guarantee termination, the IMPACT algorithm may check coverage for a single node several times, specifically whenever it starts expanding nodes in the subtree below this state. Predicate abstraction checks coverage only once directly after the state has been created. However, states are deleted during refinement and might get rediscovered, where they are again checked for coverage.

For predicate abstraction, two choices exist for how to compute an abstraction when creating a state, cartesian abstraction and boolean abstraction. It was shown that if using

single-block encoding, boolean abstraction is too slow to be useful and only cartesian abstraction is feasible. However, the latter is imprecise if there are disjunctions in the formulas that represent program operations, because it can infer truth values only for predicates independently from each other. Disjunctions occur, e.g., if pointer-alias information is encoded in the formulas, and thus predicate abstraction with cartesian abstraction may fail to prove properties that rely on this. Boolean abstraction can handle all boolean combinations of predicates and is thus more precise, but is only usable with large blocks. The IMPACT algorithm does not have this problem: it uses the interpolant directly and never loses precision.

Implementation. In order to effectively compare the performance of two algorithms, it is important to implement them in the same tool. Separate tools typically differ in many ways, which have an impact on the performance, for example the programming language, the parser frontend, the used SMT solver, support for additional features like function pointers or pointer aliasing, and optimizations like constant propagation, which are independent from the core algorithm. As a basis for our implementation we took the open-source software verification platform CPACHECKER [11]. It supports verifying C programs, is based on the CPA framework and already has an implementation of lazy predicate abstraction with CEGAR and adjustable-block encoding. We took the existing CPA for predicate abstraction, made the state-formula representation configurable (providing a BDD-based and a symbolic representation with their respective forms of coverage checks), and added an IMPACT-like refinement strategy. We also extended the CPA algorithm to support forced coverings. Thus the implementations of both algorithms differs only in the points listed above; everything else is the same code. The common code includes for example parsing, the traversal algorithm, the encoding of C code into SMT formulas, and the SMT solver.

For comparison, we also implemented the unchanged IMPACT algorithm as described by McMillan [20]. All code that is not related to the algorithm itself and the representation of abstract states is still shared with the other algorithms, so the same parser, formula encoding, and SMT solver are used.

Basic optimizations like caching queries to the SMT solver were implemented in the common code and are thus used by all algorithms. For both versions of the IMPACT algorithm (the original and ours), an optional implementation of the forced-covering optimization was added.

IV. EXPERIMENTAL EVALUATIONS

Benchmark Programs. For our experimental evaluation, we took all 277 C programs from the last competition on software verification [7], out of which 119 programs contain a known specification violation.

Experimental Setup. All experiments were performed on machines with a 3.4 GHz Quad Core CPU and 16 GB of RAM. The operating system was Ubuntu 10.04 (64 bit), using Linux 2.6.35 and OpenJDK 1.6. A time limit of 15 minutes and a memory limit of 15 GB were used. We took CPACHECKER from revision 6013 of the 'forced-covering' branch in the repository, and configured it with a Java heap size of 12 GB and MathSAT 4.2.17 as SMT solver. For comparison, we also executed benchmarks with BLAST 2.7 [23] (a tool for lazy predicate abstraction) and WOLVERINE 0.5c [24] (a tool implementing the IMPACT algorithm). For both tools we used the version and the configuration parameters which were submitted to the last software-verification competition. Unfortunately, the original IMPACT tool was not available for benchmarking. We also did benchmarks with UFO 0.1 [2] [3]. For this tool, the programs needed to be pre-processed with a special variant of CIL, compiled with LLVM, and optimized (we ignored the run time necessary for this). This pre-processing failed for 11 benchmarks.

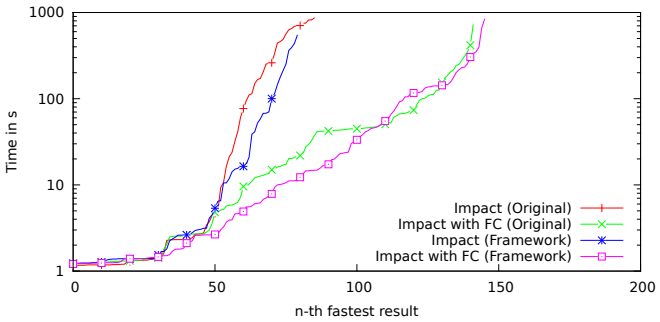

Fig. 1. Original IMPACT algorithm and our framework version; both implemented in CPACHECKER; quantile functions for verification results

Tables with the detailed results, as well as all benchmark programs, the used configurations, scripts, and a ready-to-run version of CPACHECKER are available on the supplementary webpage `http://www.sosy-lab.org/~dbeyer/cpa-uni`.

Variants of the IMPACT Algorithm. As a first set of benchmarks, we compare our implementation of the original IMPACT algorithm with the IMPACT algorithm expressed in our unifying framework, both without and with forced covering enabled. The original version is able to solve 86 and 142 instances, respectively, whereas the unifying version is able to solve 80 and 146 instances. The few differences are due to some out-of-memory conditions in the configuration. Figure 1 shows the performance for all successful verification runs of all four configurations using a plot of the quantile functions. The function graph for a configuration yields the maximum run time y (measured as used CPU time) for the xth fastest computed correct results. For example, a time of 10 s for the 100th fastest result would mean that this configuration could successfully verify 100 programs in under 10 s each, and took longer than that for all remaining programs. The x-value for which a graph ends at the top gives the maximal number of successfully verified programs for the configuration. The area below a graph (its integral) represents the accumulated verification time that the configuration needed for all programs that it could verify.

From these results we draw the conclusion that the original version of the IMPACT algorithm and our variant perform

[2]Taken from `http://www.cs.utoronto.ca/~aws/ufo/`

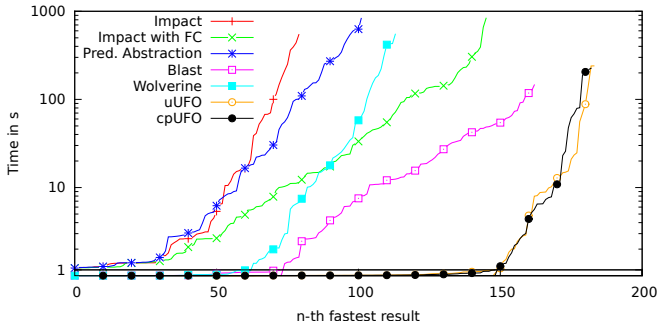

Fig. 2. SBE-based CPACHECKER configurations, BLAST, WOLVERINE, and UFO; quantile functions for verification results

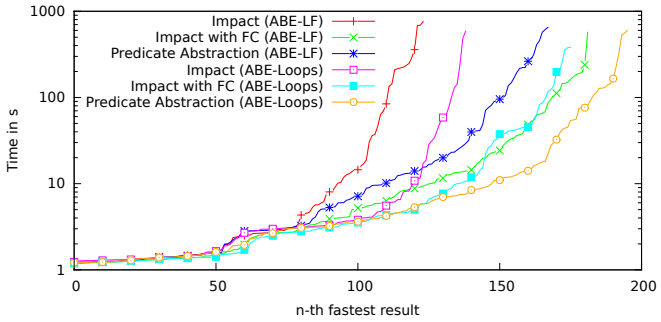

Fig. 3. Large-block CPACHECKER configurations to reduce the number of refinements; quantile functions for verification results

TABLE I
CHARACTERISTICS OF DIFFERENT CPACHECKER CONFIGURATIONS

a) Number of successfully verified programs

	SBE	ABE-LF	ABE-Loops
IMPACT	80	124	139
IMPACT with Forced Covering	146	182	176
Predicate Abstraction	102	168	196

b) Average number of refinements for successfully verified programs

	SBE	ABE-LF	ABE-Loops
IMPACT	513	304	42.5
IMPACT with Forced Covering	52.2	19.6	14.6
Predicate Abstraction	887	79.5	8.47

similar enough and we are able to further experiment with our unifying framework only.

Benchmarks using Single-Block Encoding. Now we compare the configurations of our analysis framework against each other: the IMPACT algorithm and lazy predicate abstraction. The former is run with and without the forced-covering optimization. For reference, we also run benchmarks with other tools that implement one of these algorithms: BLAST, WOLVERINE, and UFO. The latter is run in two configurations, without abstraction computations (uUFO, similar to IMPACT) and with cartesian predicate abstraction (cpUFO). All configurations except the two UFO configurations use single-block encoding, i.e., the former do not group several program statements into larger blocks.

Figure 2 shows the performance of these configurations and tools. The number of solved instances for the CPACHECKER configurations can also be seen in the column 'SBE' of Table I a). Comparing the IMPACT algorithm (1st row) with predicate abstraction (3rd row), we can see that the latter can solve 22 more programs, and is somewhat faster (cf. graph). This indicates that the eagerness of predicate abstraction pays off, and the amount of work spent for computing abstractions is worth the effort. Omitting the abstraction computation and delegating the coverage checks to an SMT solver needs more time, although the SMT solver queries are cached. However, when forced covering is enabled for the IMPACT algorithm, it can solve 66 more programs, and is much faster than predicate abstraction. These results show that reducing the number of paths in the ARG and the number of refinements is worthwhile, even if substantial effort is needed. One forced covering consists of a satisfiability check and an interpolation query, and can thus be similarly expensive as a refinement. However, the formulas used during a check for forced covering are smaller than those during refinements, and a single successful forced covering can prevent the expansion of a whole subgraph of the ARG and thus save several refinements.

Comparing these results to the results of the other tools that are also shown in the graph is difficult, because the performance characteristics of tools written in Java, OCaml, and C/C++ are typically quite different, different SMT solvers are used, and the amount of work that was put into the tools for adding optimizations and performance tuning differs vastly. This can be seen, for example, by the fact, that in this comparison the lazy predicate abstraction implementations of CPACHECKER and BLAST have a significant performance difference, although they are conceptually the same.

Benchmarks with Large Blocks. We have already identified that the most important performance factor is the number of refinements, and not the algorithm itself. Thus, we are interested in seeing how both algorithms perform when adjustable-block encoding (ABE) is used to group many program statements into larger blocks. This approach is known to vastly reduce the number of refinements [12]. One important advantage of our unified framework is that we can now use ABE with both algorithms without any further work, although it was originally only designed and implemented for predicate abstraction. Of course, ABE does not save abstraction computations if the IMPACT algorithm is used, but it still does allow to merge paths and does reduce the number of refinements, and it saves coverage checks as these are only done at block ends. The first results have shown that the original version of IMPACT behaves similarly to our framework version, therefore, we did not implement adjustable-block encoding for it.

For the larger blocks, we use two different block sizes: ABE-LF (block ends at function entries/exits and loop heads) and ABE-Loops (block ends only at loop heads). The number of solved instances is shown in Table I a) and the performance results in Fig. 3. First of all, the results show the expected improvement in performance and number of solved instances if the block size is increased from SBE to ABE-LF and further to ABE-Loops. This holds for all three configurations with one exception which we will discuss below. The results further confirm that the IMPACT algorithm without forced covering is the slowest of those configurations regardless of the block

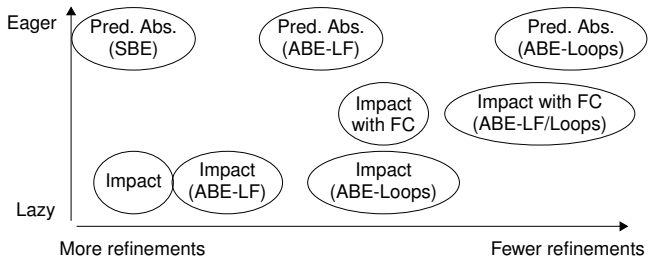

Fig. 4. Classification of various combinations of lazy predicate abstraction and IMPACT with forced covering and adjustable-block encoding

size. For ABE-LF (the medium block size), forced covering provides a performance benefit for the IMPACT algorithm similar to when SBE is used, making it faster than predicate abstraction. However, for ABE-Loops (the largest block size) the results differ: The performance improvement of forced covering still exists, but is smaller than for the other block sizes. The IMPACT algorithm is now slower than predicate abstraction, even with forced covering enabled. Furthermore, this configuration solves fewer instances with ABE-Loops than with ABE-LF, although for all other configurations an increase in the block size also leads to a significant performance increase. The reason for this is that the ABE-Loops block size has reduced the number of refinements already so much that the forced-covering optimization has little chance to achieve a further reduction. Because the only abstraction states that remain belong to loop-head locations, forced covering is attempted only for such states. However, for loop heads the abstract states need to be annotated with loop invariants in order to reach the fixed point, and those invariants are only discovered by refinements, not by forced-covering attempts. The overhead for the unsuccessful attempts then leads to a performance decrease. Table I b) shows the average number of refinements for each successful verification run and confirms this. For SBE and ABE-LF, forced covering can reduce the number of refinements by one order of magnitude, however, for ABE-Loops it only manages to reduce it to one third.

V. CONCLUSION

We have presented a new unifying framework for predicate-based model checking, and expressed the two most successful existing approaches in this framework. This allowed us to gain new insights about these algorithms, especially that the performance benefit of IMPACT compared to SBE-based predicate abstraction is not due to the omitted abstractions, but instead due to the reduction of the number of refinements using forced covering. We can now classify all existing and new configurations as in Fig. 4. We showed that using our framework does not add overhead compared to the original versions of the algorithms. Instead it is beneficial for flexibly experimenting with new configurations, such as combining the IMPACT-based algorithm and adjustable-block encoding.

These experiments confirm that the common property of the most successful configurations is to reduce the number of refinements. The new insights from this experimental study are useful for directing future research on software model checking. Specifically, we have provided an experimental infrastructure to study the impact of the various parameters that distinguish the algorithms.

We plan to extend our framework by incorporating further model-checking algorithms. A comprehensive framework will allow us to learn more about other algorithms and to experiment with new —perhaps even more powerful— strategies for software model checking that were not possible before.

REFERENCES

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. From under-approximations to over-approximations and back. In *Proc. TACAS*, LNCS 7214, pages 157–172. Springer, 2012.

[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. WHALE: An interpolation-based algorithm for inter-procedural verification. In *Proc. VMCAI*, LNCS 7148, pages 39–55. Springer, 2012.

[3] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *Proc. CAV*, LNCS 7358, pages 672–678. Springer, 2012.

[4] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The Static Driver Verifier research platform. In *Proc. CAV*, LNCS 6174, pages 119–122. Springer, 2010.

[5] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *Proc. TACAS*, LNCS 2031, pages 268–283. Springer, 2001.

[6] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.

[7] D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS*, LNCS 7214, pages 504–524. Springer, 2012.

[8] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proc. FMCAD*, pages 25–32. IEEE, 2009.

[9] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.

[10] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.

[11] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[12] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.

[13] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. *Fundam. Inform.*, 89(4):369–392, 2008.

[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[15] E. Ermis, J. Hoenicke, and A. Podelski. Splitting via interpolants. In *VMCAI*, LNCS 7148, pages 186–201. Springer, 2012.

[16] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.

[17] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *Proc. POPL*, pages 471–482. ACM, 2010.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

[20] K. L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV*, LNCS 4144, pages 123–136. Springer, 2006.

[21] K. L. McMillan. Lazy annotation for program testing and verification. In *Proc. CAV*, LNCS 6174, pages 104–118. Springer, 2010.

[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, LNCS 2304, pages 213–228. Springer, 2002.

[23] P. Shved, M. Mandrykin, and V. Mutilin. Predicate analysis with BLAST 2.7. In *Proc. TACAS*, pages 525–527. Springer, 2012.

[24] G. Weissenbacher, D. Kröning, and S. Malik. WOLVERINE: Battling bugs with interpolants. In *Proc. TACAS*, pages 556–558. Springer, 2012.