Configurable Software Verification:

Concretizing the Convergence of Model Checking and Program Analysis *

Dirk Beyer[1], Thomas A. Henzinger[2], and Grégory Théoduloz[2]

[1] Simon Fraser University, B.C., Canada
[2] EPFL, Switzerland

Abstract. In automatic software verification, we have observed a theoretical convergence of model checking and program analysis. In practice, however, model checkers are still mostly concerned with precision, e.g., the removal of spurious counterexamples; for this purpose they build and refine reachability trees. Lattice-based program analyzers, on the other hand, are primarily concerned with efficiency. We designed an algorithm and built a tool that can be configured to perform not only a purely tree-based or a purely lattice-based analysis, but offers many intermediate settings that have not been evaluated before. The algorithm and tool take one or more abstract interpreters, such as a predicate abstraction and a shape analysis, and configure their execution and interaction using several parameters. Our experiments show that such customization may lead to dramatic improvements in the precision-efficiency spectrum.

1 Introduction

Automatic program verification requires a choice between precision and efficiency. The more precise a method, the fewer false positives it will produce, but also the more expensive it is, and thus applicable to fewer programs. Historically, this trade-off was reflected in two major approaches to static verification: program analysis and model checking. While in principle, each of the two approaches can be (and has been) viewed as a subcase of the other [7, 18, 19], such theoretical relationships have had little impact on the practice of verification. Program analyzers, by and large, still target the efficient computation of few simple facts about large programs; model checkers, by contrast, focus still on the removal of false alarms through ever more refined analyses of relatively small programs. Emphasizing efficiency, static program analyzers are usually path-insensitive, because the most efficient abstract domains lose precision at the join points of program paths. Emphasizing precision, software model checkers, on the other hand, usually never join abstract domain elements (such as predicates), but explore an abstract reachability tree that keeps different program paths separate.

In order to experiment with the trade-offs, and in order to be able to set the dial between the two extreme points, we have extended the software model checker BLAST [11] to permit customized program analyses. Traditionally, customization has meant to choose a particular abstract interpreter (abstract domain and transfer functions, perhaps a widening operator) [8, 13, 14, 20], or a combination of abstract interpreters [4, 6, 10, 12]. Here, we go a step further in that we also configure the execution

* This research was supported in part by the grant SFU/PRG 06-3, and by the Swiss National Science Foundation.

engine of the chosen abstract interpreters. At one extreme (typical for program analyzers), the execution engine propagates abstract domain elements along the edges of the control-flow graph of a program until a fixpoint is reached [5]. At the other extreme (typical for model checkers), the execution engine unrolls the control-flow graph into a reachability tree and decorates the tree nodes with abstract domain elements, until each node is 'covered' by some other node that has already been explored [11]. In order to customize the execution of a program analysis, we define and implement a meta engine that needs to be configured by providing, in addition to one or more abstract interpreters, a *merge* operator and a *termination* check.

The merge operator indicates when two nodes of a reachability tree are merged, and when they are explored separately: in classical program analysis, two nodes are merged if they refer to the same control location of the program; in classical model checking, no nodes are merged. The termination check indicates when the exploration of a path in the reachability tree is stopped at a node: in classical program analysis, when the corresponding abstract state does not represent new (unexplored) concrete states (i.e., a fixpoint has been reached); in classical model checking, when the corresponding abstract state represents a subset of the concrete states represented by another node. Our motivation is practical, not theoretical: while it is theoretically possible to redefine the abstract interpreter to capture different merge operators and termination checks within a single execution engine, we wish to reuse abstract interpreters as building blocks, while still experimenting with different merge operators and termination checks. This is particularly useful when several abstract interpreters are combined. In this case, our meta engine can be configured by defining a *composite merge operator* from the component merge operators; a *composite termination check* from the component termination checks; but also a *composite transfer function* from the component transfer functions.

Combining the advantages of different execution engines for different abstract interpreters can yield dramatic results, as was shown by *predicated lattices* [9]. That work combined predicate abstraction with a data-flow domain: the data-flow analysis becomes more precise by distinguishing different paths through predicates; at the same time, the efficiency of a lattice-based analysis is preserved for facts that are difficult to track by predicates. However, the configuration of predicated lattices is just one possibility, combining abstract reachability trees for the predicate domain with a join-based analysis for the data-flow domain. Another example is *lazy shape analysis* [2], where we combined predicate abstraction and shape analysis. Again, we 'hard-wired' one particular such combination: no merging of nodes; termination by checking coverage between individual nodes; cartesian product of transfer functions. Our new, configurable implementation permits the systematic experimentation with many variations, and the results are presented in this paper. We show that different configurations can lead to large, example-dependent differences in precision and performance. In particular, it is often useful to use non-cartesian transfer functions, where information flows between multiple abstract interpreters, e.g., from the predicate state to the shape state (or lattice state), and vice versa. By choosing suitable abstract interpreters and configuring the meta engine, we can also compare the effectiveness and efficiency of symbolic versus explicit representations of values, and the use of different pointer alias analyses in software model checking.

In recent years we have observed a convergence of historically distinct program verification techniques. It is indeed difficult to say whether our configurable verifier is a model checker (as it is based on BLAST) or a program analyzer (as it is configured by choosing a set of abstract interpreters and some parameters for executing and combining them). We believe that the distinction is no longer practically meaningful (it has not been theoretically meaningful for some time), and that this signals a new phase in automatic software verification tools.

2 Formalism and Algorithm

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.[3] A *program* is represented by a *control-flow automaton* (CFA), which consists of a set L of control locations (models the program counter pc), an initial location pc_0 (models the program entry), and a set $G \subseteq L \times Ops \times L$ of control-flow edges (models the operation that is executed when control flows from one location to another). The *concrete state* of a program is a variable assignment c that assigns to each variable from $X \cup \{pc\}$ a value. The set of all concrete states of a program is denoted by C. Each edge $g \in G$ defines a (labeled) transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The complete transition relation $\rightarrow$ is the union over all edges: $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$. 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 $c_{i-1} \rightarrow c_i$.

2.1 Configurable Program Analysis

A *configurable program 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 termination check stop, which are explained in the following. These four components *configure* our algorithm and influence the precision and cost of a program analysis.

1. The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ is defined by a set C of concrete states, a semi-lattice $\mathcal{E}$, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of a (possibly infinite) set E of elements, a top element $\top \in E$, a bottom element $\bot \in E$, a preorder $\sqsubseteq \subseteq E \times E$, and a total function $\sqcup : E \times E \rightarrow E$ (the join operator). The lattice elements from E are the abstract states. The concretization function $[\![\cdot]\!] : E \rightarrow 2^C$ assigns to each abstract state its meaning, i.e., the set of concrete states that it represents. The abstract domain determines the objective of the analysis.

2. The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E$ assigns to each abstract state e possible new abstract states e' that are abstract successors of e, and each transfer is labeled with a control-flow edge g. We write $e \overset{g}{\rightsquigarrow} e'$ if $(e, g, e') \in \rightsquigarrow$, and $e \rightsquigarrow e'$ if there exists a g with $e \overset{g}{\rightsquigarrow} e'$. For soundness and progress of the program analysis, the abstract domain and the corresponding transfer relation have to fulfill the following requirements:

[3] In our implementation based on BLAST, we allow C programs as inputs, and transform them into the intermediate language CIL [16]. Interprocedural analysis is supported.

(a) $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$;
(b) $\forall e, e' \in E$: $[\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$
(the join operator is precise or over-approximates);
(c) $\forall e \in E$: $\exists e' \in E$: $e \rightsquigarrow e'$ (the transfer relation is total);
(d) $\forall e \in E, g \in G : \bigcup_{e \overset{g}{\rightsquigarrow} e'} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{c' \mid c \xrightarrow{g} c'\}$
(the transfer relation over-approximates operations).

3. The *merge operator* $\mathsf{merge} : E \times E \to E$ combines the information of two abstract states. To guarantee soundness of our analysis, we require that $e' \sqsubseteq \mathsf{merge}(e, e')$ (the result can only be more abstract than the second parameter). Dependent on the element e, the result of merge can be anything between e' and $\top$, i.e., the operator weakens the second parameter depending on the first parameter. Furthermore, if $\mathbb{D}$ is a composite analysis, the merge operator can join some of the components dependent on others. Note that the operator merge is not commutative, and is not the same as the join operator $\sqcup$ of the lattice, but merge can be based on $\sqcup$. Later we will use the following merge operators: $\mathsf{merge}^{sep}(e, e') = e'$ and $\mathsf{merge}^{join}(e, e') = e \sqcup e'$.

4. The *termination check* $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ checks if the abstract state that is given as first parameter is covered by the set of abstract states given as second parameter. We require for soundness of the termination check that $\mathsf{stop}(e, R) = \mathit{true}$ implies $[\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$. The termination check can, for example, go through the elements of the set R that is given as second parameter and search for a single element that subsumes ($\sqsubseteq$) the first parameter, or —if D is a powerset domain[4]— can join the elements of R to check if R subsumes the first parameter. Note that the termination check stop is not the same as the preorder $\sqsubseteq$ of the lattice, but stop can be based on $\sqsubseteq$. Later we will use the following termination checks (the second requires a powerset domain):
$\mathsf{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ and $\mathsf{stop}^{join}(e, R) = (e \sqsubseteq \bigsqcup_{e' \in R} e')$.

Note that the abstract domain on its own does not determine the precision of the analysis; each of the four configurable components (abstract domain, transfer relation, merge operator, and termination check) independently influences both precision and cost.

Among the program analyses that we use later in the experiments are the following:
Location Analysis. A configurable program analysis $\mathbb{L} = (D_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, \mathsf{merge}_{\mathbb{L}}, \mathsf{stop}_{\mathbb{L}})$ that tracks the reachability of CFA locations has the following components: the domain $D_{\mathbb{L}}$ is based on the flat lattice for the set L of CFA locations; the transfer relation $\rightsquigarrow_{\mathbb{L}}$ with $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$ if there exists an edge $g = (l, op, l') \in G$, and $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} \bot$ otherwise (the syntactical successor in the CFA without considering the semantics of the operation op); the merge operator $\mathsf{merge}_{\mathbb{L}} = \mathsf{merge}^{sep}$; and the termination check $\mathsf{stop}_{\mathbb{L}} = \mathsf{stop}^{sep}$.
Predicate Abstraction. A program analysis for cartesian predicate abstraction was defined by Ball et al. [1]. Their framework can be expressed as a configurable program analysis $\mathbb{P}$ by using their abstract domain and transfer relation, and choosing the merge operator $\mathsf{merge}_{\mathbb{P}} = \mathsf{merge}^{sep}$ and the termination check $\mathsf{stop}_{\mathbb{P}} = \mathsf{stop}^{sep}$.
Shape Analysis. Shape analysis is a static analysis that uses finite structures (shape graphs) to represent instances of heap-stored data structures. We can express the framework of Sagiv et al. [17] as a configurable program analysis $\mathbb{S}$ by using their

[4] A *powerset domain* is an abstract domain such that $[\![e_1 \sqcup e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$.

Algorithm 1 $CPA(\mathbb{D}, e_0)$

Input: a configurable program analysis $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$,
an initial abstract state $e_0 \in E$, let E denote the set of elements of the semi-lattice of D
Output: a set of reachable abstract states
Variables: a set reached of elements of E, a set waitlist of elements of E

```
waitlist := {e_0}
reached := {e_0}
while waitlist ≠ ∅ do
  pop e from waitlist
  for each e' with e⇝e' do
    for each e'' ∈ reached do
      // Combine with existing abstract state.
      e_new := merge(e', e'')
      if e_new ≠ e'' then
        waitlist := (waitlist ∪ {e_new}) \ {e''}
        reached := (reached ∪ {e_new}) \ {e''}
    if ¬ stop(e', reached) then
      waitlist := waitlist ∪ {e'}
      reached := reached ∪ {e'}
return reached
```

abstract (powerset) domain and transfer relation, and choosing the merge operator $\mathsf{merge}_{\mathbb{S}} = \mathsf{merge}^{join}$ and the termination check $\mathsf{stop}_{\mathbb{S}} = \mathsf{stop}^{join}$.

2.2 Execution Algorithm

The reachability algorithm CPA computes, for a given configurable program analysis and an initial abstract state, a set of reachable abstract states, i.e., an over-approximation of the set of reachable concrete states. The configurable program analysis is given by the abstract domain D, the transfer relation $\rightsquigarrow$ of the input program, the merge operator merge, and the termination check stop. The algorithm keeps updating two sets of abstract states, a set reached to store all abstract states that are found to be reachable, and a set waitlist to store all abstract states that are not yet processed (frontier). The state exploration starts with the initial abstract state e_0. For a current abstract state e, the algorithm considers each successor e', obtained from the transfer relation. Now, using the given operator merge, the abstract successor state is combined with an existing abstract state from reached. If the operator merge has added information to the new abstract state, such that the old abstract state is subsumed, then the old abstract state is replaced by the new one.[5] If after the merge step the resulting new abstract state is not covered by the set reached, then it is added to the set reached and to the set waitlist.[6]

[5] Implementation remark: The operator merge can be implemented in a way that it operates directly on the reached set. If the set reached is stored in a sorted data structure, there is no need to iterate over the full set of reachable abstract states, but only over the abstract states that need to be combined.

[6] Implementation remark: The termination check can be done additionally before the merge process. This speeds up cases where the termination check is cheaper than the merge.

Theorem 1 (Soundness). *For a given configurable program analysis $\mathbb{D}$ and an initial abstract state e_0, Algorithm CPA computes a set of abstract states that over-approximates the set of reachable concrete states:* $\bigcup_{e \in CPA(\mathbb{D}, e_0)} [\![e]\!] \supseteq Reach([\![e_0]\!])$.

We now show how model checking and data-flow analysis are instances of configurable program analysis.

Data-Flow Analysis. Data-flow analysis is the problem of assigning to each control-flow location a lattice element that over-approximates the set of possible concrete states at that program location. The least solution (smallest over-approximation) can be found by computing the least fixpoint, by iteratively applying the transfer relation to abstract states and joining the resulting abstract state with the abstract state that was assigned to the location in a previous iteration. The decision whether the fixpoint is reached is usually based on a working list of data-flow facts that were newly added. In our configurable program analysis, the data-flow setting can be realized by choosing the merge operator merge^{join} and the termination check stop^{join}.
Note that a configurable program analysis can model improvements (in precision or efficiency) for an existing data-flow analysis without redesigning the abstract domain of the existing data-flow analysis. For example, a new data-flow analysis that uses a subset of the powerset domain 2^D, instead of D itself, can be represented by a configurable program analysis reusing the domain D and its operators, but using an appropriate new merge operator that is different from merge^{join}. Moreover, static analyzers such as ASTRÉE [3] use delayed joins, or path partitioning [15], to improve the precision and efficiency of the analysis. We can model these techniques within our framework by changing only the merge operator.

Model Checking. A typical model-checking algorithm explores the abstract reachable state space of the program by unfolding the CFA, which results in an *abstract reachability tree*. For a given abstract state, the abstract successor state is computed and added as successor node to the tree. Branches in the CFA have their corresponding branches in the abstract reachability tree, but since two paths never meet, a join operator is never applied. This tree data structure supports the required path analysis in CEGAR frameworks, as well as reporting a counterexample if a bug is found. The decision whether the fixpoint is reached is usually implemented by a coverage check, i.e., the algorithm checks each time a new node is added to the tree if the abstract state of that node is already subsumed by some other node. BLAST's model-checking algorithm can be instantiated as a configurable program analysis by choosing the merge operator merge^{sep} and the termination check stop^{sep}.

Combinations of Model Checking and Program Analysis. Due to the fact that the model-checking algorithm never uses a join operator, the analysis is automatically path-sensitive. In contrast, path-sensitivity in data-flow analysis requires the use of a more precise data-flow lattice that distinguishes abstract states on different paths. On the other hand, due to the join operations, the data-flow analysis can reach the fixpoint much faster in many cases. Different abstract interpreters exhibit significant differences in precision and cost, depending on the choice for the merge operator and termination check. Therefore, we need a mechanism to combine the best choices of the operators for different abstract interpreters when composing the resulting program analyses.

2.3 Composite Program Analyses

A configurable program analysis can be composed of several configurable program analyses. A *composite program analysis* $\mathcal{C} = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$[7] consists of two configurable program analyses $\mathbb{D}_1$ and $\mathbb{D}_2$, a composite transfer relation $\rightsquigarrow_\times$, a composite merge operator $\mathsf{merge}_\times$, and a composite termination check $\mathsf{stop}_\times$. The three composites $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$ are expressions over the components of $\mathbb{D}_1$ and $\mathbb{D}_2$ ($\rightsquigarrow_i, \mathsf{merge}_i, \mathsf{stop}_i, [\![\cdot]\!]_i, E_i, \top_i, \bot_i, \sqsubseteq_i, \sqcup_i$), as well as the operators $\downarrow$ and $\preceq$ (defined below). The composite operators can manipulate lattice elements only through those components, never directly (e.g., if $\mathbb{D}_1$ is already a result of a composition, then we cannot access the tuple elements of abstract states from E_1, nor redefine merge_1). The only way of using additional information is through the operators $\downarrow$ and $\preceq$. The *strengthening* operator $\downarrow : E_1 \times E_2 \to E_1$ computes a stronger element from the lattice set E_1 by using the information of a lattice element from E_2; it has to meet the requirement $\downarrow(e, e') \sqsubseteq e$. The strengthening operator can be used to define a composite transfer relation $\rightsquigarrow_\times$ that is stronger than a pure product relation. For example, if we combine predicate abstraction and shape analysis, the strengthening operator $\downarrow_{\mathbb{S},\mathbb{P}}$ can 'sharpen' the field predicates of the shape graphs by considering the predicate region. Furthermore, we allow the definitions of composite operators to use the *compare* relation $\preceq \subseteq E_1 \times E_2$, to compare elements of different lattices.

For a given composite program analysis $\mathcal{C} = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, we can construct a configurable program analysis $\mathbb{D}_\times = (D_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, where the product domain $D_\times$ is defined as the direct product of D_1 and D_2: $D_\times = D_1 \times D_2 = (C, \mathcal{E}_\times, [\![\cdot]\!]_\times)$. The product lattice is $\mathcal{E}_\times = \mathcal{E}_1 \times \mathcal{E}_2 = (E_1 \times E_2, (\top_1, \top_2), (\bot_1, \bot_2), \sqsubseteq_\times, \sqcup_\times)$ with $(e_1, e_2) \sqsubseteq_\times (e_1', e_2')$ iff $e_1 \sqsubseteq_1 e_1'$ and $e_2 \sqsubseteq_2 e_2'$, and $(e_1, e_2) \sqcup_\times (e_1', e_2') = (e_1 \sqcup_1 e_1', e_2 \sqcup_2 e_2')$. The product concretization function $[\![\cdot]\!]_\times$ is such that $[\![(d_1, d_2)]\!]_\times = [\![d_1]\!]_1 \cap [\![d_2]\!]_2$. The literature agrees that this direct product itself is often not sharp enough [4, 6]. Even improvements over the direct product (e.g., the reduced product [6] or the logical product [10]) do not solve the problem completely. However, in a configurable program analysis, we can specify the desired degree of 'sharpness' in the composite operators $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$. For a given product domain, the definition of the three composite operators determines the precision of the resulting configurable program analysis. In previous approaches, a redefinition of basic operations was necessary, but using configurable program analysis, we can reuse the existing abstract interpreters.

Example: BLAST's Domain. The program analysis that is implemented in the tool BLAST can be expressed as a configurable program analysis $\mathbb{D}$ that derives from the composite program analysis $\mathcal{C} = (\mathbb{L}, \mathbb{P}, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, where the components are the configurable program analysis $\mathbb{L}$ for locations and the configurable program analysis $\mathbb{P}$ for predicate abstraction. We construct the composite transfer relation $\rightsquigarrow_\times$ such that $(l, r) \overset{g}{\rightsquigarrow}_\times (l', r')$ iff $l \overset{g}{\rightsquigarrow}_\mathbb{L} l'$ and $r \overset{g}{\rightsquigarrow}_\mathbb{P} r'$. We choose the composite merge operator $\mathsf{merge}_\times = \mathsf{merge}^{sep}$ and the composite termination check $\mathsf{stop}_\times = \mathsf{stop}^{sep}$.

Example: BLAST's Domain + Shape Analysis. The combination of predicate abstraction and shape analysis [2] can now be expressed as the composite program analysis

[7] We extend this notation to any finite number of $\mathbb{D}_i$.

$\mathcal{C} = (\mathbb{L}, \mathbb{P}, \mathbb{S}, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$ with the three components: location analysis $\mathbb{L}$, predicate abstraction $\mathbb{P}$, and shape analysis $\mathbb{S}$. In our previous work [2] we used a configuration that corresponds to the composite merge operator $\mathsf{merge}_\times = \mathsf{merge}^{sep}$ and the composite termination check $\mathsf{stop}_\times = \mathsf{stop}^{sep}$. Our new tool allows us now to define the three composite operators $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$ in many different ways, and we report the results of our experiments in Sect. 3.

Example: BLAST's Domain + Pointer Analysis. Fischer et al. used a particular combination (called *predicated lattices*) of predicate abstraction and a data-flow analysis for pointers [9], which we can express as the composite program analysis $\mathcal{C} = (\mathbb{L}, \mathbb{P}, \mathbb{A}, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, where $\mathbb{A}$ is a configurable pointer analysis. The transfer relation $\rightsquigarrow_\times$ is such that $(l, r, d) \overset{g}{\rightsquigarrow}_\times (l', r', d')$ iff $l \overset{g}{\rightsquigarrow}_\mathbb{L} l'$ and $r \overset{g}{\rightsquigarrow}_\mathbb{P} r'$ and $d \overset{g}{\rightsquigarrow}_\mathbb{A} d'$. We can configure the algorithm of Fischer et al. by choosing the composite termination check $\mathsf{stop}_\times = \mathsf{stop}^{sep}$ and the composite merge operator that joins the third elements if the first two agree:

$$\mathsf{merge}_\times((l, r, d), (l', r', d')) = \begin{cases} (l', r', \mathsf{merge}_\mathbb{A}(d, d')) & \text{if } l = l' \text{ and } r = r' \\ (l', r', d') & \text{otherwise} \end{cases}$$

with $\mathsf{merge}_\mathbb{A}(d, d') = d \sqcup_\mathbb{A} d'$.

Remark: Location Domain. Traditional data-flow analyses do not consider the location domain as a separate abstract domain; they assume that the locations are always explicitly analyzed. In contrast, we leave this completely up to the interpreter. We find it interesting to consider the program counter as just another program variable, and define a location domain that makes the program counter explicit when composed with other domains. This releases the other abstract domains from defining the location handling, and only the parameters for the composite program analysis need to be set. This keeps different concerns separate. Usually, only the program counter variable is modeled explicitly, and all other variables are represented symbolically (e.g., by predicates or shapes). We have the freedom to treat *any* program variable explicitly, not only the program counter; this may be useful for loop indices. Conversely, we can treat the program counter symbolically, and let other variables 'span' the abstract reachability tree.

3 Experiments

We evaluated our new approach on several combinations of abstract interpreters, under several different configurations. We implemented the configurable program analysis as an extension of the model checker BLAST, in order to be able to reuse many components that are necessary for an analysis tool but out of our focus in this work. BLAST supports recursive function calls, as well as pointers and recursive data structures on the heap. For representing the shape-analysis domain we use parts of the TVLA implementation [13]. For pointer-alias analysis, we use the implementation that comes with CIL [16]. We use the configuration of Fischer et al. [9] to compare with predicated lattices.

3.1 Configuring Model Checking + Shape Analysis

For our first set of experiments, we consider the combination of predicate abstraction and shape analysis. To demonstrate the impact of various configurations on performance

and precision, we ran our algorithm on the set of example C programs from [2], extended by some programs to explore scalability. These examples can be divided into three categories: (1) examples that require only unary and binary (shape) predicates to be proved safe (`list_`i, `simple`, and `simple_backw`), (2) examples that require in addition nullary predicates (`alternating` and `list_flag`), and (3) an example that requires that information from the nullary predicates is used to compute the new value of unary and binary predicates (`list_flag2`). The verification times are given in Table 1 for the six different configurations (A-F). When BLAST fails to prove the program safe for a given configuration, a false alarm (FP) is reported.

A: Predicated Lattice (merge-pred-join, stop-sep). In our first configuration we use the traditional model-checking approach (no join) for the predicate abstraction, and the predicated-join approach for the shape analysis. This corresponds to the following composite operators:

1. $(l, r, s) \overset{g}{\rightsquigarrow}_{\times} (l', r', s')$ iff $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$ and $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} r'$ and $s \overset{g}{\rightsquigarrow}_{\mathbb{S}} s'$

2. $\mathsf{merge}_{\times}((l, r, s), (l', r', s')) = \begin{cases} (l', r', \mathsf{merge}_{\mathbb{S}}(s, s')) & \text{if } l = l' \text{ and } r = r' \\ (l', r', s') & \text{otherwise} \end{cases}$

3. $\mathsf{stop}_{\times}((l, r, d), R) = \mathsf{stop}^{sep}((l, r, d), R)$

The transfer relation is cartesian, i.e., the successors of the different components are computed independently (cf. [2]). The merge operator joins the shape graphs of abstract regions that agree on both the location and the predicate region. The predicate regions are never joined. Termination is checked using the coverage against a single abstract state. This configuration corresponds to Fischer et al.'s *predicated lattice* [9].

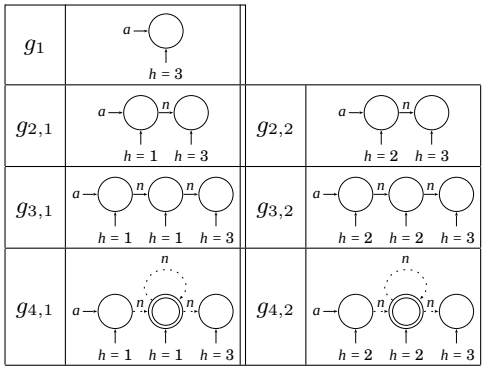
Example. To illustrate the difference between the various configurations, we use the C program in Fig. 1(a). This program constructs a lists that contains the data values 1 or 2, depending on the value of the variable `flag`, and ends with a single 3. We illustrate the example using the following abstractions. In the predicate abstraction, we keep track of the nullary predicate *flag*. In the shape analysis, we consider shape graphs for the list pointed to by program variable `a`, and field predicates for the assertions $h = 1$, $h = 2$, and $h = 3$. (These abstractions are automatically discovered by BLAST's refinement procedure [2], but this is not the subject of this paper). Figure 1(b) shows some shape graphs that are encountered during the analysis. The nodes of a shape graph correspond to memory cells. Summary nodes (double circles) represent 0, 1, or more nodes. Shape graphs are defined by the valuation of predicates over nodes in a three-valued logic. Predicates are either unary (e.g., the points-to predicate a or the field predicates $h = 1$, $h = 2$, and $h = 3$) or binary (e.g., the *next* predicate n).

To understand how this composite program analysis works on this example, we consider abstract states for which the location component has the value 15 (program exit point). Because of the merge operator, abstract states that agree on both the location and the predicates are joined. Consequently, shape graphs corresponding to lists with different lengths are collected in a single abstract state. At the end of the analysis, we therefore find at most one abstract state per location and predicate valuation, e.g., $(15, \mathit{flag}, \{g_1, g_{2,1}, g_{3,1}, g_{4,1}\})$ and $(15, \neg \mathit{flag}, \{g_1, g_{2,2}, g_{3,2}, g_{4,2}\})$.

Experimental Results. Precision: Shape analysis is based on a powerset domain, and therefore the join has no negative effect on the precision of the analysis. *Performance:*

```
1 typedef struct node {
2   int h;  struct node *n;
3 } *List;
4 void foo(int flag) {
5   List a = (List) malloc(...);
6   if (a == NULL) exit(1);
7   List p = a;
8   while (random()) {
9     if (flag)  p->h = 1;
10    else       p->h = 2;
11    p->n = (List) malloc(...);
12    if (p->n == NULL) exit(1);
13    p = p->n;  }
14  p->h = 3;
15 }
```

(a) Example C program



(b) Example shape graphs

Fig. 1. Example program and two list representations

The idea behind the join in data-flow analysis is to keep the number of abstract states small for efficiency and progress reasons, and in a typical data-flow analysis the join operations are efficient. However, since an abstract state contains a set of shape graphs in our analysis, the effect is the opposite: the join operations add extra work, because larger sets of shape graphs need to be manipulated. In addition, when the algorithm computes successors of a joined set, the work that may have been done already for some subset is repeated. This results in unnecessarily many, highly expensive operations.

B: As Precise as Model Checking (merge-sep, stop-sep). Now we want to avoid that the merge operator causes join overhead in the analysis when computing abstract successor states. This is easy to achieve in our composite program analysis: we replace the composite merge operator $\mathsf{merge}_\times$ by the merge operator merge^{sep}. The new composite program analysis joins neither predicate regions nor shape regions, and corresponds to *lazy shape analysis* [2].

Example. Since this composite program analysis is not joining elements, there is no reached abstract state with a set of shape graphs of size larger than 1 (unlike in the previous configuration A). Instead, we maintain distinct abstract states. In particular, at the exit location, the set of reached abstract states contains the following abstract states: $(15, \mathit{flag}, \{g_1\})$, $(15, \mathit{flag}, \{g_{2,1}\})$, $(15, \mathit{flag}, \{g_{3,1}\})$, $(15, \mathit{flag}, \{g_{4,1}\})$, $(15, \neg\mathit{flag}, \{g_1\})$, $(15, \neg\mathit{flag}, \{g_{2,1}\})$, $(15, \neg\mathit{flag}, \{g_{3,1}\})$, and $(15, \neg\mathit{flag}, \{g_{4,1}\})$. This set of abstract states represents exactly the same set of concrete states as the result of the previous analysis (configuration A).

Experimental Results. All examples in our experiments have smaller run times using this configuration, and the precision in the experiments does not change, compared to configuration A. *Precision:* Shape analysis is based on a powerset domain, and therefore, joins are precise. The precision of the predicated lattice is the same as the precision of this variant without joins. *Performance:* Although the number of explored abstract states is slightly higher, this configuration improves the performance of the analysis. The size of lattice elements (i.e., the average number of shape graphs in an abstract state) is considerably smaller than in the predicated-lattice configuration (A). Therefore, we achieve a better performance, because operations (in particular the successor computations) on small sets of shape graphs are much more efficient than on large sets.

C: More Precision by Improved Transfer Relation (merge-sep, stop-sep, transfer-new). From the first to the second configuration, we could improve the performance of the analysis. Now, we show how the precision of the analysis can be improved. We replace the cartesian transfer relation [2] by a new version that does not compute successors completely independently for the different sub-domains:

$(l, r, s) \overset{g}{\rightsquigarrow}_{\times} (l', r', s')$ iff $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$ and $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} r'$ and $s \overset{g}{\rightsquigarrow}_{\mathbb{S}} s''$ and $s' = \downarrow_{\mathbb{S},\mathbb{P}}(s'', r')$

The strengthening operator improves the precision of the transfer relation by using the predicate region to sharpen the shape information.

Example. In the example, the strengthening operator has no effect, because the nullary predicate *flag* has no relation with any predicates used in the shape graph. The strengthening operator would prove useful if, for example, the shape graphs had in addition a unary field predicate $h = x$ (indicating that the field `h` of a node has the same value as the program variable `x`), and the predicate abstraction had the nullary predicate $x = 3$. Consider the operation at line 14 (`p->h = 3`). The successor of the shape graph before applying the strengthening operator can only update the unary field predicate $h = x$ to value $1/2$, while the unary field predicate $h = 3$ can be set to value 1 for the node pointed to by p. Supposing $x = 3$ holds in the predicate region of the abstract successor, the strengthening operator updates the field predicate $h = x$ to value 1 as well.

Experimental Results. This configuration results in an improvement in precision over published results for a 'hard-wired' configuration [2], at almost no cost. *Precision:* Because of the strengthening operator, the abstract successors are more precise than using the cartesian transfer relation. Therefore, the whole analysis is more precise. *Performance:* The cost of the strengthening operator is small compared to the cost of the shape-successor computation. Therefore, the performance is not severely impacted when compared to a cartesian transfer relation.

D: As Precise as Model Checking with Improved Termination Check (merge-sep, stop-join). Now we try to achieve another improvement over configuration B: we replace the termination check with one that checks the abstract state against the join of the reached abstract states that agree on locations and predicates:

$\mathsf{stop}_{\times}((l, r, s), R) = (s \sqsubseteq_{\mathbb{S}} \bigsqcup_{\mathbb{S}} \{s' \mid (l, r, s') \in R\})$

The previous termination check was going through the set of already reached abstract states, checking against every abstract state for coverage. Alternatively, abstract states that agree on the predicate abstraction can be summarized by one single abstract state that is used for the termination check. This is sound because the shape-analysis domain is a powerset domain.

Example. To illustrate the use of the new termination check in the example, consider a set of reached abstract states that contains at some intermediate step the following abstract states: $(15, \mathit{flag}, \{g_1\})$, $(15, \mathit{flag}, \{g_{2,1}\})$, $(15, \neg\mathit{flag}, \{g_1\})$, and $(15, \neg\mathit{flag}, \{g_{2,2}\})$. If we want to apply the termination check to the abstract state $(15, \mathit{flag}, \{g_1, g_{2,1}\})$ and the given set of reached abstract states, we check whether the set $\{g_1, g_{2,1}\}$ of shape graphs is a subset of the join of all shape graphs already found for this location and valuation of predicates (that is, the set $\{g_1, g_{2,1}\}$). The check would not be positive at this point using termination check stop^{sep}.

Experimental Results. The overall performance impact is slightly negative. *Precision:* This configuration does not change the precision for our examples. *Performance:* We

expected improved performance by (1) avoiding many single coverage checks because of the summary abstract state, and (2) fewer successor computations, because we may recognize earlier that the fixpoint is reached. However, the performance impact in our examples is negligible, because a very small portion of the time is spent on termination checks, and the gain is more than negated by the overhead due to the joins.

E: Join at Meet-Points as in Data-Flow Analysis (merge-join, stop-join). To compare with a classical data-flow analysis, we choose a configuration such that the data-flow elements are joined where the control flow meets, independently of the predicate region. We use the following merge operator, which joins with *all* previously computed shape graphs for the program location of the abstract state:

$$\mathsf{merge}_\times((l,r,d),(l',r',d')) = \begin{cases} (l',r',\mathsf{merge}_\mathbb{S}(d,d')) & \text{if } l = l' \\ (l',r',d') & \text{otherwise} \end{cases}$$

Example. The composite program analysis encounters, for example, the abstract state $(15, \mathit{flag}, \{g_1, g_{2,1}, g_{2,2}, g_{3,1}, g_{3,2}, \ldots\})$, which contains shape graphs for lists that contain either 1s or 2s despite the fact that *flag* has the value true. Therefore, we note a loss of precision compared to the predicated-lattice approach (configuration A), because the less precise merge operator looses the correlation between the value of the nullary predicate *flag* and the shape graphs.

Experimental Results. The analysis is not able to prove several of the examples that were successfully verified with previous configurations. *Precision:* The shape-analysis component has lost the path-sensitivity: the resulting shape graphs are similar to what a classical fixpoint algorithm for data-flow analysis would yield. Therefore, the analysis is less precise. *Performance:* The run time is similar to configuration A.

F: Predicate Abstraction with Join (merge-join for preds). We now evaluate a composite program analysis that is similar to a classical data-flow analysis, i.e., both predicates and shapes are joined for the abstract states that agree on the program location. We consider the following merge operator:

$$\mathsf{merge}_\times((l,r,s),(l',r',s')) = \begin{cases} (l',\mathsf{merge}_\mathbb{P}(r,r'),\mathsf{merge}_\mathbb{S}(d,d')) & \text{if } l = l' \\ (l',r',d') & \text{otherwise} \end{cases}$$

where $\mathsf{merge}_\mathbb{P}(r,r') = r \sqcup_\mathbb{P} r'$ is the weakest conjunction of predicates that implies $r \vee r'$. This composite program analysis corresponds exactly to a data-flow analysis on the direct product of the two lattices: the set of reached abstract states contains only one abstract state per location, because the merge operator joins abstract states of the same location.

Example. At location 15, we have one abstract state: $(15, \mathit{true}, \{g_1, g_{2,1}, g_{2,2}, \ldots\})$.

Experimental Results. This configuration can prove the same example programs as configuration E, and the run times are also similar to configuration E. *Precision:* This composite program analysis is the least precise in our set of configurations, because the merge operator joins both the predicates and the shape graphs independently, for a given location. While join is suitable for many data-flow analyses, predicate abstraction becomes very imprecise when predicate regions are joined, because then it is not possible to express disjunctions of predicates by the means of separate abstract states for the same location. *Performance:* Compared to configuration E, the number of abstract states is smaller (only one per location), but the shape graphs have the same size. Therefore, this configuration is less precise, although not more efficient.

Table 1. Time for the different configurations[8] (false alarm: FP)

Program	A pred-join stop-sep	B merge-sep stop-sep	C merge-sep stop-sep transfer-new	D merge-sep stop-join	E merge-join stop-join	F merge-join stop-join join preds
`simple`	0.53 s	0.32 s	0.40 s	0.34 s	0.51 s	0.50 s
`simple_backw`	0.43 s	0.28 s	0.26 s	0.31 s	0.44 s	0.45 s
`list_1`	0.42 s	0.37 s	0.41 s	0.32 s	0.41 s	0.41 s
`list_2`	5.24 s	0.85 s	1.25 s	0.86 s	5.34 s	5.36 s
`list_3`	138.97 s	1.79 s	2.62 s	2.10 s	132.08 s	132.07 s
`list_4`	> 600 s	9.67 s	15.44 s	11.87 s	> 600 s	> 600 s
`alternating`	0.86 s	0.61 s	0.96 s	0.60 s	FP	FP
`list_flag`	0.69 s	0.49 s	0.79 s	0.46 s	FP	FP
`list_flag2`	FP	FP	0.81 s	FP	FP	FP

Table 2. Time for examples run with a predicated lattice

Program	CFA nodes	LOC	A: orig. (join)	B: more precision (no join)
`s3_clnt`	272	2 547	0.680 s	0.830 s
`s3_srvr`	322	2 542	0.560 s	0.590 s
`cdaudio`	968	18 225	33.50 s	> 600 s
`diskperf`	549	14 286	248.330 s	> 600 s

Summary. For our set of examples, the experiments have shown that configuration C is the best choice, and we provided justifications for the results. However, we cannot conclude that configuration C is the preferred configuration for *any* combination of abstract interpreters, and we provide evidence for this in the next subsection.

3.2 Configuring Model Checking + Pointer Analysis

In the experimental setting of this subsection we show that for a certain kind of abstract interpreter the join is not only better, but that algorithms without join show prohibitive performance, or do not terminate. We consider the combination of BLAST's predicate domain and a pointer-analysis domain, as described at the end of Sect. 2. In Table 2 we report the performance results for two different algorithms: configuration A for a "predicated lattice," as described by Fischer et al. [9], and configuration B for an algorithm without join, using the merge operator merge^{sep}. The experiments give evidence that the number of abstract states explodes and blows up the computational overhead, but the gained precision is not even necessary for proving our example programs correct.

4 Conclusion

When the goal is as difficult as automatic software verification, it is imperative to bring to bear insights and optimizations no matter if they originated in model checking, program analysis, or automated theorem proving (which is heavily used in BLAST, to compute transfer functions and to perform termination checks). We have therefore modified BLAST from a tree-based software model checker to a tool that can be configured using

[8] A: predicated join; B: no join (model checking); C: no join and more precise transfer relation; D: no join, termination check with join; E: normal join of shapes (data-flow analysis); F: join for predicate abstraction. All experiments were run on a 3 GHz Intel Xeon processor.

different lattice-based abstract interpreters, composite transfer functions, merge operators, and termination checks. Specifically configured extensions of BLAST with lattice-based analysis had been implemented before, e.g., in predicated lattices [9] and in lazy shape analysis [2]. As a side-effect, we can now express the algorithmic settings of these papers in a simple and systematic way, and moreover, we have found different configurations that perform even better.

References

1. 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.
2. D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *Proc. CAV*, LNCS 4144, pages 532–546. Springer, 2006.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566, pages 85–108. Springer, 2002.
4. M. Codish, A. Mulkers, M. Bruynooghe, M. de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *Proc. PEPM*, pages 194–205. ACM, 1993.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL*, pages 238–252. ACM, 1977.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. POPL*, pages 269–282. ACM, 1979.
7. P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Proc. CAV*, LNCS 939, pages 293–308. Springer, 1995.
8. M. B. Dwyer and L. A. Clarke. A flexible architecture for building data-flow analyzers. In *Proc. ICSE*, pages 554–564. IEEE, 1996.
9. J. Fischer, R. Jhala, and R. Majumdar. Joining data flow with predicates. In *Proc. ESEC/FSE*, pages 227–236. ACM, 2005.
10. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *Proc. PLDI*, pages 376–386. ACM, 2006.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
12. S. Lerner, D. Grove, and C. Chambers. Composing data-flow analyses and transformations. In *Proc. POPL*, pages 270–282. ACM, 2002.
13. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. SAS*, LNCS 2280, pages 280–301. Springer, 2000.
14. F. Martin. PAG: An efficient program analyzer generator. *STTT*, 2:46–67, 1998.
15. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proc. ESOP*, LNCS 3444, pages 5–20. Springer, 2005.
16. G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, LNCS 2304, pages 213–228. 2002.
17. M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24:217–298, 2002.
18. D. A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proc. POPL*, pages 38–48. ACM, 1998.
19. B. Steffen. Data-flow analysis as model checking. In *Proc. TACS*, pages 346–365. 1991.
20. S. W. K. Tjiangan and J. Hennessy. SHARLIT: A tool for building optimizers. In *Proc. PLDI*, pages 82–93. ACM, 1992.