Program Analysis with Dynamic Precision Adjustment *

Dirk Beyer
Simon Fraser University, Canada

Thomas A. Henzinger
EPFL, Switzerland

Grégory Théoduloz
EPFL, Switzerland

Abstract

We present and evaluate a framework and tool for combining multiple program analyses which allows the dynamic (on-line) adjustment of the precision of each analysis depending on the accumulated results. For example, the explicit tracking of the values of a variable may be switched off in favor of a predicate abstraction when and where the number of different variable values that have been encountered has exceeded a specified threshold. The method is evaluated on verifying the SSH client/server software and shows significant gains compared with predicate abstraction-based model checking.

1. Introduction

The success of program analysis depends on finding a delicate trade-off between the precision of an analysis and its cost. If the analysis is not precise enough, the result is an overwhelming number of false alarms; if the analysis is too expensive, it will not scale to large programs. Traditional research in program analysis has focused on efficient, scalable analyses that are specified by the user before being executed by a tool [9, 17]. Research in model checking, on the other hand, has focused on expressive, expensive analyses (such as predicate abstraction) whose precision can be increased automatically, during execution of the analysis, as much as necessary [1,7]. The traditional approach has the drawbacks that when the result of an analysis is inconclusive, the user has to start a new analysis, e.g., with greater precision; and that a given analysis and precision are applied globally to the whole program. The model-checking approach touts to overcome these drawbacks [16]. However, not only is it expensive to automatically increase the precision of an analysis (e.g., predicate discovery [6,15]), but in order to support an automatic precision-refinement procedure, the analysis itself (predicate abstraction) is so expressive as to be prohibitively expensive. As a result, there remains a gap of several orders of magnitude between the size of programs that can be model checked and the size of programs that yield to traditional static analyses [5].

In this paper, we evaluate a new way of increasing the precision of an analysis during execution. What we do is best viewed as running several different analyses simultaneously and using their results on-line to adjust their respective precisions. For example, we may run an explicit analysis, which tracks the values of a set of program variables, in parallel with a predicate analysis, which tracks the values of a set of predicates. We may start by tracking all variables explicitly, and no predicates. As soon as we encounter, during the analysis, a specified threshold number of different values of a variable, we may switch from tracking the value of the variable to tracking a predicate (or set of predicates) involving the variable. In other words, we dynamically decrease the precision of one analysis (tracking fewer variables explicitly) and at the same time increase the precision of another analysis (tracking more predicates). This scheme, which can be applied to any combination of program analyses, has several advantages. Compared with the purely predicate-based precision refinement performed by many current software model checkers [1, 2], predicates are used only when and where a more efficient analysis fails. Compared with the explicit and symbolic analyses performed by execution-based model checkers [12, 18], predicates (or other forms of widening) can be introduced when and where they are needed to complete a proof. Compared with traditional program analyzers [5, 17], the precision of an analysis can be refined on-line, during the analysis, when and where necessary.

In a second set of examples, we run an explicit heap analysis in parallel with a graph-based shape analysis [17]. We start tracking heap-stored data structures with an explicit, concrete representation of the heap content. When a certain number of concrete heap cells is reached for a given data structure, we switch from tracking heap content explicitly to an appropriate symbolic representation based on shape graphs.

We present our scheme using the formalism of configurable program analysis (CPA) [4], which captures within one framework both traditional program analyses and soft-

*This research was supported in part by the Canadian NSERC grant RGPIN 341819-07, by the Swiss NSF, and by Microsoft Research through its PhD scholarship program.

978-1-4244-2188-6/08/$25.00 ©2008 IEEE

```
1 int main() {
2   int st = 0; int ok = 0; int p;
3   int *a = getarray(); int n = length(a);
4   int cmd = readcmd();
5   while (1) {
6     switch (st) {
7     case 0:
8       if (cmd == 177) { st = 1; } else { st = 2; }
9       break;
10    case 1:
11      if (cmd == 177) { st = 3; cmd = readcmd(); }
12      else { st = 0; }
13      break;
14    case 2:
15      if (cmd != 177) { cmd = 79; st = 4; p = cmd;
16          while (p > 0) { --p; *(a+p) = 0; }
17          if (p > 0) goto ERR; }
18      else { goto ERR; }
19      break;
20    case 3:
21      if (cmd == 78) { st = 4; ok = 1; }
22      else { st = 0; cmd = readcmd(); }
23      break;
24    case 4:
25      if (ok) { if (cmd != 78) goto ERR; }
26      else { if (cmd != 79) goto ERR; }
27      goto cont;
28  } }
29 cont: p = cmd;
30  while (p < n) { ++p; *(a+p) = 0;}
31  if (p < n) goto ERR;
32  return 0;
33  ERR: goto ERR; return 1;
34 }
```

Figure 1. Example program

ware model checking, and offers a flexible composition of several analyses. We compute not on abstract states, but on pairs (e, π) consisting of an abstract state e and a precision π. For example, e may be the value of a floating-point variable and π the number of digits in the mantissa of the floating-point representation; together they represent a set of possible real numbers. In our algorithm, the precision of an analysis can be changed depending on all abstract state–precision pairs computed so far, i.e., depending on global information. For example, a new predicate may be introduced depending on the fact that a certain set of explicit values of a variable have been encountered (collected) in the analysis so far. Or, one analysis may be switched on or off depending on the accumulated results produced by another analysis so far.

Example. The program shown in Fig. 1 is inspired by the code in the SSH server software. We verify that the location at label `ERR` is not reachable. For the analysis we use the already mentioned combination of an explicit analysis and a predicate analysis, where new predicates are introduced whenever the number of encountered values of a variable exceeds a given threshold. Initially, the explicit analysis tracks the values of the variables `st`, `ok`, `cmd`, and `p`. For each of the first three variables, no more than five different values are encountered. However, the loop counter `p` assumes a number of different values that cannot be bounded by a constant. The precision adjustment that takes place when the number of values for variable `p` hits, say, 10, prevents the explicit analysis from exploring infinitely many concrete values. Then, the precision adjustment injects the predicate $p < n$ into the predicate analysis and turns off the explicit analysis for variable `p`. By tracking the value of the predicate $p < n$ in the subsequent iteration, the loop analysis terminates and the falsehood of the predicate prevents the symbolic execution from entering the error state. During the analysis, we simultaneously performed a *refinement* of the predicate abstract domain, because we added a new predicate, and an *abstraction* of the explicit abstract domain, because we removed a variable. We call this a dynamic (on-line) precision adjustment of the analysis.

Implementation. We have implemented the configurable program analysis with dynamic precision adjustment (CPA+) as an extension of the BLAST toolkit [2], which enables us to reuse many components such as the parser-frontend CIL, the interface to external theorem provers for the predicate analysis, the pointer analysis, and the interface to the TVLA package for the heap analysis. We were careful to define CPA+ so that it allows the reuse of existing CPA components, such as explicit and predicate analyses [4]. Other component analyses include a shape analysis with pairs (e, π), where the abstract state e is a set of shape graphs and the precision π is a tracking definition that specifies the predicates used in the shape analysis [3]; and a polyhedral analysis with pairs (e, π), where the precision π is a tuple of variables for which we track linear relations and the abstract state e is a matrix that represents relations between the variable values.

Experiments. We show that configurable program analysis with dynamic precision adjustment significantly outperforms, in the two combinations we consider, pure predicate abstraction-based model checking and pure shape graph-based static analysis, respectively. We evaluate the method on several artificial programs to expose different scenarios for which different precisions are necessary, and on real code from the SSH client/server software.

Related Approaches. First, it is theoretically possible to model dynamic precision adjustments within a configurable program analysis [4], but to do so, one would have to encode the set of previously encountered abstract states as part of each abstract state. This would result in a contrived abstract domain and would not allow the reuse of existing analyses.

Second, there have been several previous proposals for combining explicit and symbolic analyses [13, 19]. While their emphasis is on executing program parts in actual runtime environments in order to provide inexpensive information to a symbolic search for proof, we view both explicit and symbolic execution as program analyses whose precisions can be adjusted and traded off dynamically. This gives us greater flexibility. While the previous approaches apply explicit program execution to all variables, we choose dynamically and automatically which variables to track explicitly.

Third, in predicate-based abstraction refinement algorithms, new predicates are introduced based on informa-

tion obtained from counterexamples [1, 7]. By contrast, we introduce new predicates based on information obtained from the abstract states encountered so far during the analysis. This is related to the automatic discovery of invariants from state samples [11], such as linear relationships between variables, and to widening operators used in abstract interpretation [9]. However, unlike widening, CPA+ does not relax the precision of individual abstract states but adjusts the precision of entire analyses going forward. In particular, the precision of any component analysis may be increased as well as decreased, depending on the accumulated results so far.

2. Program-Analysis Framework

We introduce the concept of configurable program analysis with dynamic precision adjustment (CPA+). A CPA+ makes it possible to dynamically adjust the precision of an analysis while exploring the abstract state space of the program. A composite CPA+ can control the precision of its component analyses during the verification process: it can make a component analysis more abstract, and more efficient (in the extreme: switch it off completely), and it can make a component analysis more precise, and more expensive (in the extreme: track the values for each variable with the best precision possible).

Preliminaries. 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 pc), an initial program location pc_0 (models the program entry), and a set $G \subseteq L \times Ops \times L$ of control-flow edges (models the operation that is executed when control flows from one program location to another). The set of program variables that occur in operations from Ops is denoted by X. A *concrete state* of a program is a variable assignment $c : X \cup \{pc\} \to \mathbb{Z}$ that assigns to each variable an integer value. The set of all concrete states of a program is denoted by C. A set $r \subseteq C$ of concrete states is called a *region*. Each edge $g \in G$ defines a (labeled) transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The complete transition relation $\to$ is the union over all control-flow edges: $\to = \bigcup_{g \in G} \xrightarrow{g}$. We write $c \xrightarrow{g} c'$ if $(c, g, c') \in \to$, and $c \to c'$ if there exists a g with $c \xrightarrow{g} c'$. A concrete state c_n is *reachable* from a region r, denoted by $c_n \in Reach(r)$, if there exists a sequence of concrete states $\langle c_0, c_1, \ldots, c_n \rangle$ such that $c_0 \in r$ and for all $1 \leq i \leq n$, we have $c_{i-1} \to c_i$.

[1] In our implementation based on BLAST, we allow C programs as inputs, and transform them into the intermediate language CIL (`http://manju.cs.berkeley.edu/cil/`). BLAST supports interprocedural program analysis.

2.1. Configurable Program Analysis with Dynamic Precision Adjustment

For presentation and formalization purposes, we use the framework of configurable program analysis [4]. This provides the advantage of making explicit the new, orthogonal capability that dynamic precision adjustment gives the analysis. We add two components to the CPA: in order to pair abstract-domain elements with precisions, we introduce a set of precisions, and in order to adjust the precision of such pairs, we introduce a precision adjustment operator. Thus the new algorithm for dynamic precision adjustment will be able to maintain for each abstract state its precision.

A *configurable program analysis with dynamic precision adjustment* (CPA+) $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ consists of an abstract domain D, a set Π of precisions, a transfer relation $\rightsquigarrow$, a merge operator merge, a termination check stop, and a precision adjustment function prec, which are explained in the following.

1. The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ is defined by the set C of concrete states, the semi-lattice $\mathcal{E}$ of abstract states, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of the (possibly infinite) set E of abstract domain elements, the top element $\top \in E$, the bottom element $\bot \in E$, the partial order $\sqsubseteq \subseteq E \times E$, and the function $\sqcup : E \times E \to E$ (the join operator). The function $\sqcup$ yields the least upper bound for two lattice elements, and the symbols $\top$ and $\bot$ denote the least upper bound and greatest lower bound of the set E, respectively. The concretization function $[\![\cdot]\!] : E \to 2^C$ assigns to each abstract state e its meaning, i.e., the set of concrete states that it represents. For soundness of the program analysis, the abstract domain has to fulfill the following requirements:

(a) $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$

(b) $\forall e, e' \in E : e \sqsubseteq e' \Rightarrow [\![e]\!] \subseteq [\![e']\!]$

(c) $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$
(the join operator is precise or overapproximate)

Note that requirement (c) is implied by (b) and the fact that $\sqcup$ is the least upper bound.

2. The *set* Π *of precisions* determines the possible precisions of the abstract domain. The program analysis uses elements from Π to keep track of different precisions for different abstract states. A pair (e, π) is called *abstract state* e *with precision* π. The operators on the abstract domain are parametric in the precision.

3. The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ assigns to each abstract state e possible new abstract states e' with precision p which are abstract successors of e, and each transfer is labeled with a control-flow edge g. We write $e \overset{g}{\rightsquigarrow} (e', \pi)$

if $(e, g, e', \pi) \in \rightsquigarrow$, and $e \rightsquigarrow (e', \pi)$ if there exists a g with $e \overset{g}{\rightsquigarrow} (e', \pi)$.
The transfer relation has to fulfill the following requirement:

(d) $\forall e \in E, g \in G, \pi \in \Pi$:
$\bigcup_{e \overset{g}{\rightsquigarrow} (e', \pi)} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{c' \mid c \xrightarrow{g} c'\}$
(the transfer relation overapproximates operations for every fixed precision)

4. The *merge operator* $\mathsf{merge} : E \times E \times \Pi \to E$ weakens the second parameter using the information of the first parameter, and returns a new abstract state of the precision that is given as third parameter.
The merge operator has to fulfill the following requirement:

(e) $\forall e, e' \in E, \pi \in \Pi : e' \sqsubseteq \mathsf{merge}(e, e', \pi)$
(the result of merge can only be more abstract than the second parameter)

5. The *termination check* $\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ checks if the abstract state given as first parameter with the precision given as third parameter, is covered by the set of abstract states given as second parameter. 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[2]— can join the elements of R to check if R subsumes the first parameter.
The termination check has to fulfill the following requirement:

(f) $\forall e \in E,\ R \subseteq E,\ \pi \in \Pi$:
$\mathsf{stop}(e, R, \pi) = \mathit{true} \ \Rightarrow\ [\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$
(if an abstract state e is considered to be 'covered' by R, then every concrete state represented by e is represented by some abstract state from R)

6. The *precision adjustment function* $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ computes a new abstract state and a new precision, for a given abstract state with precision and a set of abstract states with precision. The precision adjustment function may perform widening of the abstract state, in addition to a change of precision.
The precision adjustment function has to fulfill the following requirement:

(g) $\forall e, \hat{e} \in E, p, \hat{p} \in \Pi, R \subseteq E \times \Pi$:
$(\hat{e}, \hat{p}) = \mathsf{prec}(e, p, R) \ \Rightarrow\ [\![e]\!] \subseteq [\![\hat{e}]\!]$

Note. Classical widening and strengthening operators can only decrease and increase precision, respectively. The new operator prec can be used for both increasing and decreasing the precision of abstract states. In compositions of several program analyses, the operator can even be used to increase the precision of one component analysis and decrease the precision of another component analysis, because the (composite) precision operator can access information of both.

[2] A *powerset domain* is an abstract domain s.t. $[\![e \sqcup e']\!] = [\![e]\!] \cup [\![e']\!]$.

Algorithm 2.1 CPA+$(\mathbb{D}, e_0, \pi_0)$

Input: a configurable program analysis with dynamic precision adjustment $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
an initial abstract state $e_0 \in E$ with precision $\pi_0 \in \Pi$,
where E denotes the set of elements of the semi-lattice of D
Output: a set of reachable abstract states
Variables: a set reached of elements of $E \times \Pi$,
a set waitlist of elements of $E \times \Pi$

```
waitlist := {(e_0, π_0)};  reached := {(e_0, π_0)};
while waitlist ≠ ∅ do
  pop (e, π) from waitlist;
  // Adjust the precision.
  (ê, π̂) = prec(e, π, reached);
  for each e' with ê ⇝ (e', π̂) do
    for each (e'', π'') ∈ reached do
      // Combine with existing abstract state.
      e_new := merge(e', e'', π̂);
      if e_new ≠ e'' then
        waitlist := (waitlist ∪ {(e_new, π̂)}) \ {(e'', π'')};
        reached := (reached ∪ {(e_new, π̂)}) \ {(e'', π'')};
    // Add new abstract state?
    if ¬ stop(e', {e | (e, ·) ∈ reached}, π̂) then
      waitlist := waitlist ∪ {(e', π̂)};
      reached := reached ∪ {(e', π̂)}
return {e | (e, ·) ∈ reached}
```

2.2. Reachability Algorithm for CPA+

The abstract domain of a program analysis defines a way to describe abstract program states. An analysis algorithm needs to know in addition a precision for the abstract states (for example, which variable to ignore). Usually this information is hard-wired in either the abstract-domain elements or the algorithm. The CPA+ algorithm keeps for every abstract state a precision, i.e., we use a pair (e, π) with $e \in E$ and $\pi \in \Pi$ to describe an abstract state and the precision at which it was computed. This means that the precision of the analysis depends on the *context*, i.e., abstract states the algorithm is working on. The precision used in the algorithm can change dynamically from abstract state to abstract state, by adjusting the precision using the function prec. Each of the three other configurable components (transfer relation, merge operator, and termination check) is parameterized with the precision of the resulting abstract state.

The reachability algorithm CPA+ (Algorithm 2.1) computes, for a given configurable program analysis with dynamic precision adjustment and an initial abstract state with precision, a set of reachable abstract states, i.e., an over-

approximation of the set of reachable concrete states. The configurable program analysis with dynamic precision adjustment is given by the abstract domain D, the precisions Π, the transfer relation $\rightsquigarrow$ of the input program, the merge operator merge, the termination check stop, and the precision adjustment function prec. The algorithm keeps updating two sets of abstract states with precision: a set reached to store all abstract states with precision that are found to be reachable, and a set waitlist to store all abstract states with precision that are not yet processed (the frontier). The state exploration starts from the initial abstract state e_0 with precision π_0. For a current abstract state e with precision π, the algorithm first adjusts the (local) precision of the algorithm using the precision adjustment function prec, based on the set of reached abstract states with precision. Next the algorithm considers each successor e' with the new precision $\hat{\pi}$, according to the transfer relation. Now, using the given operator merge, the abstract successor state is combined with each 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 with precision is replaced by the new abstract state with precision in the sets reached and waitlist. If after the merge step the resulting new abstract state with precision is not covered by the set reached, then it is added to the set reached and to the set waitlist.

Theorem 2.1 (Soundness) *For a given configurable program analysis with dynamic precision adjustment $\mathbb{D}$ and an initial abstract state e_0 with precision π_0, Algorithm* CPA+ *computes a set of abstract states that overapproximates the set of reachable concrete states:*

$$\bigcup_{e \in CPA+(\mathbb{D},e_0,\pi_0)} [\![e]\!] \supseteq Reach([\![e_0]\!]).$$

Proof. The proof constructs an invariant over the main loop of the algorithm which is strong enough to show that in each iteration, the algorithm (1) does not replace an abstract state with precision from the set reached by one that represents only a subset of concrete states, even if the precision is refined, and (2) adds abstract states that cover all concrete successors of an abstract state when it is removed from the set waitlist.

2.3. Composition for CPA+

A configurable program analysis with dynamic precision adjustment can be composed of several configurable program analyses with dynamic precision adjustment. A *composite* CPA+ $\mathcal{C} = (\mathbb{D}_1, \mathbb{D}_2, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$[3] consists of two configurable program analyses with dynamic precision adjustment $\mathbb{D}_1$ and $\mathbb{D}_2$ sharing the same set of concrete states, E_1 and E_2 being their respective sets of abstract states, a composite set of precisions $\Pi_\times$, a composite transfer relation $\rightsquigarrow_\times \subseteq (E_1 \times E_2) \times G \times (E_1 \times E_2) \times \Pi_\times$, a composite merge operator $\mathsf{merge}_\times : (E_1 \times E_2) \times (E_1 \times E_2) \times \Pi_\times \to (E_1 \times E_2)$, a composite termination check $\mathsf{stop}_\times : (E_1 \times E_2) \times 2^{E_1 \times E_2} \times \Pi_\times \to \mathbb{B}$, and a composite precision adjustment function $\mathsf{prec}_\times : (E_1 \times E_2) \times \Pi_\times \times 2^{(E_1 \times E_2) \times \Pi_\times} \to (E_1 \times E_2) \times \Pi_\times$. The three composites $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$ are expressions over the components of $\mathbb{D}_1$, $\mathbb{D}_2$, and $\Pi_\times$ $(\rightsquigarrow_i, \mathsf{merge}_i, \mathsf{stop}_i, \mathsf{prec}_i, [\![\cdot]\!]_i, E_i, \top_i, \bot_i, \sqsubseteq_i, \sqcup_i)$, as well as the operators $\downarrow$ and $\preceq$. The *strengthening* operator $\downarrow$ and the *compare* relation $\preceq$ can be used to increase the precision of the composite operators in a way similar to their use in CPA [4].

To guarantee that a configurable program analysis with dynamic precision adjustment can be built from the composite program analysis with dynamic precision, we impose the requirements for CPA+ operators (d, e, f, g) from Section 2.1 on the four composite operators.

For a given composite program analysis with dynamic precision $\mathcal{C} = (\mathbb{D}_1, \mathbb{D}_2, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$, we can construct the program analysis with dynamic precision adjustment $\mathbb{D}_\times = (D_\times, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$, 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 $(e_1, e_2) \sqsubseteq_\times (e_1', e_2')$ if $e_1 \sqsubseteq_1 e_1'$ and $e_2 \sqsubseteq_2 e_2'$ (and for the join operation, we have $(e_1, e_2) \sqcup_\times (e_1', e_2') = (e_1 \sqcup_1 e_1', e_2 \sqcup_2 e_2')$). The product concretization function $[\![\cdot]\!]_\times$ is such that $[\![(d_1, d_2)]\!]_\times = [\![d_1]\!]_1 \cap [\![d_2]\!]_2$.

It is necessary to sharpen the analysis over the otherwise imprecise cartesian product of the component abstract domains [8, 10, 14]. Previous approaches have tried to achieve this by defining a particular, specialized product. Basing our formalism on top of configurable program analysis, we can specify the desired degree of 'sharpness' in the composite operators $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$. In previous approaches, a redefinition of basic operations was necessary, but using CPA [4], we can reuse the existing abstract interpreters. In addition to that, the CPA+ allows the analysis algorithm to specify a local precision for each abstract state and to adjust this precision dynamically during the analysis.

Theorem 2.2 (Composition) *Given a composite program analysis with dynamic precision adjustment $\mathcal{C}$, the result $\mathbb{D}_\times$ of the composition is a configurable program analysis with dynamic precision adjustment.*

Proof. The requirements (a), (b), and (c) for abstract domains of CPA+ follow from the construction of $D_\times$ as

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

product domain (abstract domains are closed under product). The operators of the composite CPA+ fulfill the requirements (d), (e), (f), and (g) of CPA+ by definition.

3. Application: Combining Explicit and Symbolic Program Analysis

The motivation of this work is the need for a formalism that allows us to compose a program analysis from known parts and that makes it possible to dynamically adjust the precision of the algorithm, locally, and across different abstract domains. Our goal is to track variables explicitly in the analysis, and at the point where it becomes too expensive to track all possible values, we abstract the values that we have seen so far by predicates and add these predicates to a predicate analysis. We first introduce three CPA+ and then the composite CPA+.

3.1. CPA+ for Explicit Analysis

The CPA+ *for explicit analysis* $\mathbb{C} = (D_\mathbb{C}, \Pi_\mathbb{C}, \rightsquigarrow_\mathbb{C}, \mathsf{merge}_\mathbb{C}, \mathsf{stop}_\mathbb{C}, \mathsf{prec}_\mathbb{C})$, which can track explicit values for program variables, consists of the following components:

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

2. The set of precisions $\Pi_\mathbb{C} = \{\pi \mid \pi : X \to (\mathbb{N} \cup \{\infty\})\}$ is a set of functions. A precision $\pi \in \Pi_\mathbb{C}$ specifies for each variable a threshold value on the maximal number of different, explicitly stored values, where $\pi(x) = 0$ means the value of variable x is not tracked explicitly, and $\pi(x) = \infty$ means the value of variable x is tracked explicitly no matter how many different values exist.

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

$$e'(x) = \begin{cases} \top & \text{if } \pi(x) = 0 \text{ or } p_{/e} \Rightarrow (x = \top) \\ \bot & \text{if } \pi(x) \neq 0 \text{ and} \\ & \quad (p_{/e} \Rightarrow \mathit{false} \text{ or } \exists y \in X : e(y) = \bot) \\ c & \text{if } \pi(x) \neq 0 \text{ and } p_{/e} \Rightarrow (x = c) \\ e(x) & \text{otherwise} \end{cases}$$

where $p_{/e}$ denotes the predicate p with all occurrences of a variable $y \in X$ replaced by $e(y)$, or
(2) $g = (\cdot, \texttt{w} := exp, \cdot)$ and for all $x \in X$:

$$e'(x) = \begin{cases} \top & \text{if } \pi(x) = 0 \\ exp_{/e} & \text{if } \pi(x) \neq 0 \text{ and } x = \texttt{w} \\ e(x) & \text{otherwise} \end{cases}$$

where $exp_{/e}$ denotes the evaluation of an expression exp over X for an abstract value assignment e:

$$exp_{/e} = \begin{cases} \bot & \text{if } x \in X \text{ occurs in } exp \text{ with } e(x) = \bot \\ \top & \text{if } x \in X \text{ occurs in } exp \text{ with } e(x) = \top \\ c & \text{otherwise, where expression } exp \\ & \text{evaluates to } c\text{, after each occurrence} \\ & \text{of variable } x \in X \text{ is replaced by } e(x) \end{cases}$$

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

5. The termination check considers abstract states individually: $\mathsf{stop}_\mathbb{C}(e, R, \pi) = (\exists e' \in R : e \sqsubseteq e')$.

6. The precision adjustment function computes a new abstract state with precision: $\mathsf{prec}(e, \pi, R) = (e', \pi')$ if the following holds for all $x \in X$: if $|R(x)| \geq \pi(x)$ then $\pi'(x) = \pi(x) \wedge e'(x) = \top$, and otherwise $\pi'(x) = \pi(x) \wedge e'(x) = e(x)$, where $R(x)$ denotes the set $\{e(x) \mid (e, \cdot) \in R\}$ of explicit values held in R for variable x. For a set M of pairs, we use the notation $(x, \cdot) \in M$ for $\exists y : (x, y) \in M$.

3.2. CPA+ for Predicate Analysis

The CPA+ *for predicate analysis* $\mathbb{P} = (D_\mathbb{P}, \Pi_\mathbb{P}, \rightsquigarrow_\mathbb{P}, \mathsf{merge}_\mathbb{P}, \mathsf{stop}_\mathbb{P}, \mathsf{prec}_\mathbb{P})$, a program analysis for cartesian predicate abstraction that tracks the validity of predicates over program variables, consists of the following components:

1. The domain $D_\mathbb{P} = (C, \mathcal{E}, [\![\cdot]\!])$ is based on predicates that represent regions (of concrete states). The semi-lattice $\mathcal{E} = (2^\mathcal{P}, \emptyset, \mathcal{P}, \supseteq, \cap)$ models the abstract states as finite subsets $e \subseteq \mathcal{P}$ of predicates, where $\mathcal{P}$ is the (infinite) set of quantifier-free predicates over variables from X (using linear-arithmetic expressions and equality with uninterpreted function symbols). The concretization function $[\![\cdot]\!] : 2^\mathcal{P} \to 2^C$ assigns to each abstract state e its meaning, i.e., the set of concrete states that it represents: $[\![e]\!] = \{c \in C \mid \forall p \in e : c \models p\}$.

2. The set of precisions $\Pi_\mathbb{P} = 2^\mathcal{P}$ models a precision for an abstract state as a set of predicates. For a predicate p, if p is in precision π, then p is tracked by the analysis when precision π is used, and if p is in an abstract state e, then p is true in all concrete states represented by the abstract state e.

3. The transfer relation $\rightsquigarrow_{\mathbb{P}}$ has the transfer $e \overset{g}{\rightsquigarrow}_{\mathbb{P}}(e', \pi)$ if e' is the largest set of predicates from π such that $[\![e]\!] \subseteq \mathsf{pre}(p, g)$ for each $p \in e'$, where $\mathsf{pre}(p, g) = \{c \in C \mid \exists c' \in C : c \overset{g}{\rightarrow} c' \text{ and } c' \models p\}$ denotes the weakest precondition for predicate p and control-flow edge g.

4. The merge operator does not combine elements when control flow meets: $\mathsf{merge}_{\mathbb{P}}(e, e', \pi) = e'$.

5. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{P}}(e, R, \pi) = (\exists e' \in R : e \supseteq e')$.

6. The precision adjustment function does not change the abstract state with precision: $\mathsf{prec}(e, \pi, R) = (e, \pi)$.

Note. If a predicate is not in the precision set and not in the abstract state, then this predicate is not tracked. If a predicate is in the precision set but not in the abstract state, then there exists a concrete state represented by the abstract state for which the predicate is not true. If a predicate is in the abstract state, then the predicate is true for all concrete states represented by the abstract state.

In the above formalization, we assume a fixed universe of possible predicates — isolated automatic refinement of the predicate abstraction is not discussed in this paper, as our goal is to point out possibilities to refine the abstract states by using information from other component analyses.

3.3. CPA+ for Location Analysis

The CPA+ *for location analysis* $\mathbb{L} = (D_{\mathbb{L}}, \Pi_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, \mathsf{merge}_{\mathbb{L}}, \mathsf{stop}_{\mathbb{L}}, \mathsf{prec}_{\mathbb{L}})$, which tracks the syntactical reachability of program locations, consists of the following components:

1. The domain $D_{\mathbb{L}}$ is based on the flat lattice for the set L of program locations:
$D_{\mathbb{L}} = (C, \mathcal{E}, [\![\cdot]\!])$, with $\mathcal{E} = (L \cup \{\top, \bot\}, \top, \bot, \sqsubseteq, \sqcup)$,
$\bot \sqsubseteq l \sqsubseteq \top$ and $l \neq l' \Rightarrow l \not\sqsubseteq l'$ for all elements $l, l' \in L$
(this implies $\bot \sqcup l = l$, $\top \sqcup l = \top$, $l \sqcup l' = \top$ for all elements $l, l' \in L, l \neq l'$), and $[\![\top]\!] = C$, $[\![\bot]\!] = \emptyset$, and for all $l \in L$: $[\![l]\!] = \{c \in C \mid c(pc) = l\}$.

2. There is only one precision, which is the set of all locations: $\Pi_{\mathbb{L}} = \{L\}$.

3. The transfer relation $\rightsquigarrow_{\mathbb{L}}$ has the transfer $l \overset{g}{\rightsquigarrow}_{\mathbb{L}}(l', \pi)$ if $g = (l, \cdot, l')$, and the transfer $\top \overset{g}{\rightsquigarrow}_{\mathbb{L}}(\top, \pi)$ for all $g \in G$ (the syntactical successor in the CFA without considering the semantics of the operation op).

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

5. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{L}}(e, R, \pi) = (e \in R)$.

6. The precision is never adjusted: $\mathsf{prec}_{\mathbb{L}}(e, \pi, R) = (e, \pi)$.

3.4. Composition of Explicit, Predicate, and Location Analysis

We construct a composite CPA+ that dynamically changes the precision across different analyses, based on the analyses defined in the previous subsections. In particular, we define a composite CPA+ that on-the-fly abstracts the explicit CPA+ (by switching it off for certain variables) and refines the predicate CPA+ (by adding predicates), using information from the reachable set of abstract states with precision. The predicates added to the predicate CPA+ characterize the sample set of values provided by the explicit CPA+.

The composite program analysis with dynamic precision $\mathcal{C} = (\mathbb{L}, \mathbb{P}, \mathbb{C}, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$, is based on the CPA+ for location analysis $\mathbb{L}$, the CPA+ for predicate analysis $\mathbb{P}$, the CPA+ for explicit analysis $\mathbb{C}$, and the following components:

1. The composite precision is defined by $\Pi_\times = \Pi_{\mathbb{L}} \times \Pi_{\mathbb{P}} \times \Pi_{\mathbb{C}}$.

2. The transfer relation $\rightsquigarrow_\times$ has the transfer $(l, P, v) \overset{g}{\rightsquigarrow}_\times ((l', P', v'), (\pi_l, \pi_P, \pi_v))$ if $l \rightsquigarrow_{\mathbb{L}} (l', \pi_l)$ and $P \rightsquigarrow_{\mathbb{P}} (P', \pi_P)$ and $v \rightsquigarrow_{\mathbb{C}} (v', \pi_v)$.

3. The composite merge operator is defined by $\mathsf{merge}_\times((l, P, v), (l', P', v'), \pi) = (l', P', v')$.

4. The composite termination check is defined by $\mathsf{stop}_\times(e, R, \pi) = (\exists e' \in R : e \sqsubseteq_\times e')$.

5. The composite precision adjustment funct. is defined by $\mathsf{prec}_\times((l, P, v), (\pi_l, \pi_P, \pi_v), R) = ((l', P', v'), (\pi'_l, \pi'_P, \pi'_v))$ if $(l', \pi'_l) = (l, \pi_l)$ and
$(v', \pi'_v) = \mathsf{prec}_{\mathbb{C}}(v, \pi_v, \{(v'', \pi''_v) \mid ((l, p, v''), (\cdot, \cdot, \pi''_v)) \in R\})$ and
$\pi'_P = \pi_P \cup \bigcup_{x \in X : v(x) \neq \top \wedge v'(x) = \top} \mathsf{abstract}(x, \{v'' \mid ((l, P, v''), \cdot) \in R\})$ and
$P' = P \cup \{p \in (\pi'_P \setminus \pi_P) \mid v \models p\}$.

This composite precision adjustment function stops the explicit analysis for a variable x if the number of values for x exceeds the specified threshold. If this situation occurs, then the precision of the predicate analysis is increased by adding all predicates computed by abstraction function $\mathsf{abstract} : (X \times 2^{X \to \mathcal{Z}}) \to 2^{\mathcal{P}}$. Our implementation of abstract applies a simple heuristic for discovering relationships between variables and values, based on predicates that syntactically occur in the program; user-defined predicates are also possible. More interesting implementations, which mine predicates that best generalize the sample set of explicit value assignments, are left for future work. We plan to investigate methods based on linear templates to infer predicates from the sample set, and we will draw from dynamic approaches like DAIKON [11], which automatically detect partial invariants from program executions.

3.5. Analysis of Heap Structures

For lack of space we do not provide a full description of the following combination of configurable program analyses with dynamic precision adjustment. In order to analyze data structures on the heap, we use a composition of the CPA+ for explicit heap analysis and the CPA+ for shape analysis. The explicit heap analysis keeps track of instances of data structures (lists, trees) in its abstract states. Each abstract state represents an explicit, finite part of heap memory that is modeled as a table. Pointer addresses are modeled as symbolic values and memory content can be looked up in the table (i.e., pointer arithmetic is currently not supported). We use an additional mapping from program variables to a scalar value, a symbolic pointer, or top. The size of the explicit memory state is measured using the number of entries in the heap-content table.

The CPA+ for shape analysis is derived from the CPA for shape analysis that we used earlier [3, 4]. The precision adjustment function tries to find a trade-off between the large space requirements of the explicit heap analysis and the expensive computations of the shape analysis. More precisely, the precision adjustment function measures the size of the explicit data structures that are produced already. Once a threshold is hit, the function abstract generalizes from a set of explicit data structures to shapes, i.e., the function analyzes the explicit sample structures by querying for well-known data-structure invariants, and derives interesting core and instrumentation predicates for the shape analysis [17]. From here, the overall analysis proceeds with tracking more precise shapes and not storing explicit information anymore for that particular structure.

4. Experiments

We implemented the configurable program analysis with dynamic precision adjustment as an extension of the BLAST toolkit, in order to be able to reuse many components that are necessary for a verification tool. The tool takes as input a C program (with assertions), and command-line arguments to select which CPA+ to run. Then, the tool computes abstract reachable states, and checks that no assertions are violated, in a fully automatic way. All experiments were run on a GNU/Linux machine with a 2.66 GHz Intel Core 2 Duo 6700 processor and 2 GB RAM.

We cannot directly compare with the experiments of our previous work [4], because CPA can perform only fixed instantiations of the precision parameters of CPA+. We are able to compare CPA+ with two extreme instances (pure predicate abstraction and pure explicit analysis) of CPA.

Explicit and Predicate Analysis. For the first part of our evaluation, we used the composite CPA+ that results from combining the CPA+ $\mathbb{L}$, $\mathbb{P}$, and $\mathbb{C}$. Table 1 reports performance results for two example sets (a) and (b). We experimented with several configurations for the threshold that triggers the injection of a predicate into the predicate analysis and switches off the explicit analysis for a variable. We used constant threshold functions of the form $\pi(x) = k$, for all program variables x, where k has the value given in the column header. The extreme threshold $\pi(x) = 0$ switches the explicit analysis off, and $\pi(x) = \infty$ keeps the explicit analysis always on. The function abstract of the composite precision adjustment function returns predicates which are currently the result of simple heuristics. We plan improvements of this part as future work, e.g., considering syntac-

(a) Extreme examples

Program	Symbolic	Explicit + Symbolic		Explicit
	$k = 0$	$k = 1$	$k = 5$	$k = \infty$
ex1	0.46 s	0.17 s	0.21 s	—
ex2	0.43 s	0.16 s	0.21 s	1.00 s
ex3_1	0.16 s	0.13 s	0.11 s	0.08 s
ex3_2	0.40 s	0.24 s	0.14 s	0.09 s
ex3_4	1.41 s	0.58 s	0.20 s	0.12 s
ex3_8	6.53 s	2.10 s	0.31 s	0.18 s
loop1	25.20 s	26.01 s	22.78 s	0.16 s
loop2	279.84 s	277.07 s	258.79 s	0.44 s
square	—	—	—	0.08 s

(b) SSH client/server software

Program	Symbolic	Explicit + Symbolic	
	$k = 0$	$k = 1$	$k = 2$
s3_clnt.1	27.61 s	2.87 s	7.73 s
s3_clnt.2	22.14 s	2.47 s	3.58 s
s3_clnt.3	14.75 s	2.54 s	3.55 s
s3_clnt.4	10.61 s	2.52 s	3.56 s
s3_srvr.1	7.95 s	1.50 s	2.25 s
s3_srvr.2	5.00 s	1.39 s	2.13 s
s3_srvr.3	3.61 s	1.44 s	2.11 s
s3_srvr.4	4.32 s	1.41 s	2.14 s
s3_srvr.6	67.93 s	1.49 s	2.17 s
s3_srvr.7	35.59 s	1.95 s	2.56 s
s3_srvr.8	4.88 s	1.44 s	2.17 s
s3_srvr.9	33.20 s	1.94 s	2.67 s
s3_srvr.10	4.58 s	1.47 s	2.16 s
s3_srvr.11	36.64 s	1.95 s	2.60 s
s3_srvr.12	64.78 s	1.50 s	2.21 s
s3_srvr.13	125.91 s	2.04 s	2.64 s
s3_srvr.14	68.27 s	1.52 s	2.19 s
s3_srvr.15	5.01 s	1.68 s	2.12 s
s3_srvr.16	69.12 s	1.51 s	2.28 s

Table 1. Performance comparison on two sets of examples for the combination of predicate abstraction and explicit variable tracking

Program	Shapes	Explicit + Shapes		Explicit
		$h = 3$	$h = 5$	$h = \infty$
list_1	0.24 s	0.17 s	0.19 s	—
list_2	0.84 s	0.85 s	1.08 s	—
list_3	2.99 s	3.30 s	5.06 s	—
list_4	8.80 s	11.40 s	24.39 s	—
list_bnd4_1	0.35 s	0.28 s	0.07 s	0.06 s
list_bnd4_2	1.02 s	1.22 s	0.22 s	0.21 s
list_bnd4_3	2.90 s	4.40 s	1.23 s	1.15 s
list_bnd4_4	7.45 s	12.78 s	6.14 s	5.90 s
list_flags_1	0.44 s	0.36 s	0.36 s	—
list_flags_2	1.36 s	1.08 s	1.19 s	—
list_flags_3	4.95 s	3.70 s	4.15 s	—
list_flags_4	19.66 s	13.89 s	16.33 s	—
list_flags_5	79.10 s	59.09 s	74.50 s	—
list_bnd2_f1	0.42 s	0.06 s	0.05 s	0.07 s
list_bnd4_f1	0.69 s	0.51 s	0.11 s	0.08 s
list_bnd4_f2	2.38 s	1.46 s	0.21 s	0.19 s
list_bnd4_f3	8.19 s	4.86 s	0.57 s	0.52 s
list_bnd4_f4	31.65 s	18.40 s	1.94 s	2.14 s
list_bnd4_f5	130.94 s	74.72 s	9.64 s	9.48 s

Table 2. Performance comparison on examples for the combination of shape analysis and explicit heap analysis

tic predicates from the program, and predicates returned by BLAST's lazy counterexample-guided refinement.

The first set of example programs consists of some constructed, relatively small examples. The programs `ex1` and `ex2` exhibit different scenarios for which the combination of the two analyses is better than any of the component analyses on its own. Program `ex1` is the example from Fig. 1. The results show that it is best to track variables with a small number of possible values explicitly and the loop index symbolically. The purely explicit analysis ($\pi(x) = \infty$) fails for `ex1` due to the unbounded number of iterations in one of the loops. Program `ex2` is like `ex1` but the exit condition of the second loop is changed so that the number of iterations is bounded by a known constant. Therefore, the purely explicit analysis succeeds, but the symbolic analysis is still faster due to the high number of concrete values of the loop counter. Program `ex3_1` results from `ex1` by removing the loops. It consequently shows that the purely explicit analysis is always best. The three variations `ex3_2`, `ex3_4` and `ex3_8` are extended versions of `ex3_1`, with a larger number of program locations, which is twice, four times and eight times the size of `ex3_1`, respectively. This experimental setting illustrates the exponential run time for the symbolic analysis versus the linear run time for the explicit analysis.

To verify programs `loop1` and `loop2`, it is necessary to unroll a loop 100 and 200 times, respectively. A predicate analysis is prohibitively expensive because it requires a predicate for every value of the variable: $i = 0, i = 1, ..., i = 100$. Program `square` contains a function that computes the square using additions exclusively (strength reduction). Predicate analysis of this program fails for two reasons. First, even if unrolling the loop, the symbolic analysis could not find the predicate necessary to prove safety. Second, the main program uses an expression in an assert statement that is beyond the decision procedure used in BLAST. The explicit analysis is able to prove the program safe efficiently.

The second set of programs (`s3_clnt.`i, `s3_srvr.`i) represents the subroutine for the connection handshake protocol (state machine) of the SSH client and server, for which we verify several protocol-specified safety properties (one line in the table for one property). In all experiments, the combination analysis significantly outperforms the pure predicate analysis ($\pi(x) = 0$), sometimes by orders of magnitude. The reason is that most predicates in the pure predicate analysis are (mis-) used to track just one single explicit value, for which the explicit analysis is superior. The configuration for $k = 2$ always needs more time for the verification task, because it tries to track a second value for variables that require symbolic analysis; for these programs it is faster to switch to symbolic tracking after the analysis has found out that one value is not sufficient. A purely explicit analysis results in false alarms for all examples, because there are variables that have to be analyzed symbolically. (The programs contain branches whose outcome depends on inputs or uninitialized variables.)

Explicit Heap Analysis and Shape Analysis. For the second part of our evaluation, we used the CPA+ that results from combining the CPA+ $\mathbb{L}$ with the CPA+ for explicit heap analysis and the CPA+ for shape analysis. Table 2 reports performance results for a set of small programs that manipulate lists. We experimented with several configurations for the threshold value h, which is the number of nodes in the list after which we call the function abstract and switch off the explicit analysis for the heap. The extreme threshold $h = 0$ switches the explicit analysis off for the whole analysis, and $h = \infty$ keeps the explicit analysis always on. Function abstract takes as input the explicit representation of the data structure and creates a shape graph that summarizes the explicit structure.

The first four lines of the table let us analyze a negative effect of explicit heap analysis: the lists generated by program `list_i` represent an unbounded sequence of i different data elements in an ascending order. The shape graph of the shape analysis contains $O(i)$ nodes, whereas the size of the explicit structure not only has a size proportional to the length of the list, but explores many different versions of the list (combinatorial explosion). The next four examples are variations of the first four, but create lists of bounded length (`list_bnd4_i`). The results show an effect similar to the

unbounded case, i.e., the explicit analysis suffers from combinatorial explosion for larger values of i, and the summarization in the shape graphs overcompensates the overhead for the expensive shape operations. But the explicit heap analysis can be more efficient when the number of different lists that need to be examined is relatively small.

The third set of examples `list_flags_i` again produces lists of unbounded length. The performance numbers indicate that the explicit analysis can be helpful —if the threshold is reasonably small— for constructing the abstract shape graphs by conversion from explicit heap structures (speedup of 24 % for threshold $h = 3$). The last set of examples produces lists of bounded length, but requires larger and many shape graphs to prove the property. This shows that the overhead of expensive shape-analysis operations can be effectively avoided by explicit heap tracking when the control flow of the program limits the number of possible structures for the explicit analysis and leads to many different shape graphs in the shape analysis.

Discussion. So far, our preliminary experiments have not identified an absolut advantage of one of the configurations, but that different configurations lead to significantly different performance, and that further work is necessary to leverage the potential that the combination offers. We introduced the new concept of CPA+ in order to make it possible to compare such different analyses and combinations thereof, and showed that the results are indeed interesting and worth a further research project.

All benchmarks and an executable of the implementation are available online on our supplementary web page at `http://www.cs.sfu.ca/~dbeyer/blast_cpaplus/`.

5. Conclusion

We have provided an algorithm and tool for experimenting with the combination of several program analyses. Our method allows the on-line transfer of information between the different analyses, and this information can be used to increase or decrease the precision of any analysis on-the-fly. Using our tool, we showed that it can be beneficial to combine predicate abstraction with an explicit analysis because many predicates are used to track only specific values of variables. Our method allows us to dynamically change the partition between the variables that are analyzed symbolically using predicates and those that are analyzed explicitly. We also studied the effects of combining a symbolic, graph-based shape analysis with an explicit heap analysis. The new formalism and tool allow us to quickly set up such experiments and study the effects of different parameter settings. We have not yet addressed the problem of automatically mining meaningful predicates from a given set of sample values. This is left for future investigation.

References

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

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

[3] D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *Proc. CAV*, LNCS 4144, pages 532–546. Springer, 2006.

[4] 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.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. PLDI*, pages 196–207. ACM, 2003.

[6] S. Chaki, E. M. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *Proc. CHARME*, LNCS 2860, pages 19–34. Springer, 2003.

[7] 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.

[8] M. Codish, A. Mulkers, M. Bruynooghe, M. G. de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *Proc. PEPM*, 194–205. ACM, 1993.

[9] 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.

[10] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. POPL*, 269–282. ACM, 1979.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The DAIKON system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[12] P. Godefroid. Model checking for programming languages using VERISOFT. In *Proc. POPL*, pp. 174–186. ACM, 1997.

[13] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A new algorithm for property checking. In *Proc. FSE*, pages 117–127. ACM, 2006.

[14] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *Proc. PLDI*, pages 376–386. ACM, 2006.

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

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

[17] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[18] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272. ACM, 2005.

[19] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: Better together! In *Proc. ISSTA*, pages 145–156. ACM, 2006.