Combining Model Checking and Data-Flow Analysis

Dirk Beyer, Sumit Gulwani, and David A. Schmidt

Abstract Until recently, model checking and data-flow analysis — two traditional approaches to software verification — were used independently and in isolation for solving similar problems. Theoretically, the two different approaches are equivalent; they are two different ways to compute the same solution to a problem. In recent years, new practical approaches have shown how to combine the approaches and how to make them benefit from each other — model-checking techniques can make data-flow analyses more precise, and data-flow-analysis techniques can make model checking more efficient. This chapter starts by discussing the relationship (differences and similarities) between type checking, data-flow analysis, and model checking. Then we define algorithms for data-flow analysis and model checking in the same formal setting, called configurable program analysis. This identifies key differences that make us call an algorithm a "model-checking" algorithm or a "data-flow-analysis" algorithm. We illustrate the effect of using different algorithms for running certain classic example analyses and point out the reason for one algorithm being "better" than the other. The chapter presents combined verification techniques in the framework of configurable program analysis, in order to emphasize techniques used in data-flow analysis and in model checking. Besides the iterative algorithm that is used to illustrate the similarities and differences between data-flow analysis and model checking, we discuss different algorithmic approaches for constructing program invariants. To show that the border between data-flow analysis and model checking is blurring and disappearing, we also discuss directions in tool implementations for combined verification approaches.

Dirk Beyer
Ludwig-Maximilians-Universität München, Munich, Germany

Sumit Gulwani
Microsoft Research, Redmond, WA, USA

David A. Schmidt
Kansas State University, Manhattan, KS, USA

Contents

Combining Model Checking and Data-Flow Analysis 1
Dirk Beyer, Sumit Gulwani, and David A. Schmidt
1 Introduction 5
2 General Considerations 5
2.1 Type Checking 5
2.2 Data-Flow Analysis 7
2.3 Model Checking 10
3 Unifying Formal Framework/Comparison of Algorithms 12
3.1 Preliminaries 12
3.2 Algorithm of Data-Flow Analysis 15
3.3 Algorithm of Model Checking 17
3.4 Unified Algorithm Using Configurable Program Analysis 17
3.5 Discussion 20
3.6 Composition of Configurable Program Analyses 21
4 Classic Examples (Component Analyses) 22
4.1 Reachable-Code Analysis 22
4.2 Constant Propagation 23
4.3 Reaching Definitions 25
4.4 Predicate Analysis 26
4.5 Explicit-Heap Analysis 27
4.6 BDD Analysis 28
4.7 Observer Automata 29
5 Combination Examples (Composite Analyses) 31
5.1 Predicate Analysis + Constant Propagation 31
5.2 Predicate Analysis + Constant Propagation + Strengthen . 32
5.3 Predicate Analysis + Explicit-Heap Analysis 34
5.4 Predicate Analysis + Observer Automata 34
6 Algorithms for Constructing Program Invariants 36
6.1 Iterative and Monotonic Fixed-Point Approaches 36
6.2 Counterexample-Guided Abstraction Refinement 37
6.3 Template- and Constraint-Based Approaches 38

6.4 Proof-Rule-Based Approaches 39
6.5 Iterative, but Non-monotonic Approaches 39
6.6 Comparison with Standard Recurrence Solving 40
6.7 Discussion 42
7 Combinations in Tool Implementations 43
8 Conclusion 43
References 45

1 Introduction

In the context of software verification, model checking is considered a semi-decidable, exhaustive, and precise analysis of an abstract model of a program, whereas data-flow analysis is considered a terminating, imprecise abstract interpretation of a concrete model of a program.

For example, to validate a safety property, abstraction-refinement-based model checking creates an abstract model of the program and precisely analyzes every reachable abstract state for the property, repeatedly refining and rechecking the model until validation is achieved, whereas a classic data-flow analysis computes abstract values of the states that arrive at the program locations of the concrete program, repeatedly computing and combining the abstract values until convergence at all program locations is achieved. Classic data-flow analyses are efficient (as required for compiler optimization) at the cost of precision. Model checkers aim at being precise (as required for proof construction) at the cost of efficiency.

Precisely defining the difference between model checking and data-flow analysis is not easy, and indeed the two approaches have been proven to be "the same" in that each can be coded in the framework of the other. This chapter illustrates why the two approaches are theoretically equivalent — they are two fashions of computing the same solution. In practice, the two approaches are extremes in a spectrum of many possible algorithms, and the spectrum can be defined by a few parameters that describe the different implementation techniques. As soon as we set the parameters differently from the extremes that define the two approaches, we see how new combinations are possible. While in most of the chapter we assume that the popular, iteration-based algorithm is used, we later also provide a comparative overview of other algorithmic approaches for constructing program invariants.

In this chapter, we restrict ourselves to verifying safety properties of software.

2 General Considerations

For background, we compare and contrast three techniques that are widely used for static (pre-execution) program validation: type checking, data-flow analysis, and model checking. These techniques come with different representations of *program*, *property*, and *analysis algorithm*. We describe here the commonly used versions, but with small extensions it is possible for each technique to express the other two [108].

2.1 Type Checking

Type checking is an analysis of a program's syntax tree that attaches properties ("types") to the phrases that are embedded in the tree. Types might be primitive (*int*, *float*, *string*, *void*) or compound ($int[]$, $string \times float$, $\{\text{"}\texttt{name}\text{"} : string, \text{"}\texttt{age}\text{"} : int\}$)

or phrase-type (*command*(*int*), *declaration*(*ident*,*float*)). For example, the C command `float x = y + 1.5` might be parsed and type checked like this:

$$(\texttt{float x} = (\texttt{y}^{int} + \texttt{1.5}^{float})^{float})^{declaration(\texttt{x},\, float)}$$

provided that `y`'s declaration was typed by $declaration(\texttt{y}, int)$.

Type checking can validate safety properties ("a well-typed program cannot go wrong at execution") and can help a compiler generate target code.

Program Representation. A program's syntax tree (parse tree) is used for type checking. The tree is often accompanied by a symbol table that holds typings of free (global) variables.

Property Representation. There is no firm designation as to what types are, but a type should have semantic significance. Types are typically defined inductively. The earlier example used types derived from this grammar:

$$\begin{array}{ll} p : \mathit{PhraseType} & a : \mathit{ExpressionType} \\ \quad p ::= \mathit{command}(a) \mid \mathit{declaration}(\mathit{ident}, a) & \\ \quad a ::= \mathit{int} \mid \mathit{float} \mid a[\,] & \end{array}$$

The type language resembles a propositional logic, where primitive types (*int*, *string*) define the primitive propositions and compound types ($command(a)$, $declaration(ident, a)$) define the compound propositions. Data structures, such as arrays, tuples, structs, and function closures, are annotated with compound types.

The "type logic" need not be a mere propositional logic. Languages that support templates or parametric polymorphism, e.g., Standard ML [43], include Prolog-style logical variables in the syntax of types; the logical variables are placeholders for types that are inserted later, or they are understood as universally quantified variables. For example, $\alpha \to (\alpha \times \alpha)$ is a typing of this function definition:

$$\texttt{define f(x)} = \texttt{makePair(x,x)}.$$

The occurrences of α are placeholders that can be filled later, e.g., as in $f(\texttt{1.5})$ (α is replaced by *float*) or $f(\texttt{"hello"})$ (α is replaced by *string*). Indeed, the type can be read as the predicate-logic formula $\forall\alpha(\alpha \to (\alpha \wedge \alpha))$ [93].

At the other extreme, the type language can be an ad hoc collection of labels, provided there is some significance as to how the labels annotate the syntax tree. An example is *value numbering*, where each expression node is annotated by the set of expression nodes in the tree whose run-time values will equal the present node's [101].

Analysis Algorithm. A finite (usually, left-to-right, one- or two-pass) tree-traversal algorithm attaches types to the nodes of the syntax tree. In the language of Knuthian attribute grammars [82], properties that are *inherited* are carried from parent nodes to child nodes for further computation, and properties that are *synthesized* are com-

municated from child nodes to parent nodes. In the first example in this section, variable y's type is inherited information that is passed to the phrase $(\mathtt{y} + 1.5)$, which synthesizes the type *float*. The algorithm for attaching properties can be written with attribute-grammar equations [86] or inference rules [97].

The equations (or rules) are meant to be deterministic, but some program phrases might be annotated with multiple acceptable choices (e.g., $2 : \mathit{int}$ and also $2 : \mathit{float}$). In this case, an ordering, $\leq$, as in $\mathit{int} \leq \mathit{float}$, lets one deduce a most precise property for a phrase. This concept, called *subtyping* [1], is central to type checking for object-oriented languages. Using logical variables, ML's Algorithm W [43] deduces a most general typing for an ML program that can be correctly typed in multiple ways.

Extensions. If the typing language is complex enough, it can express any or all semantic properties of a program, e.g., a phrase's "type" might be its compiled code or it might be the input-output function that the phrase denotes! (The former is called a *syntax-directed translation* [2] and the latter is the program's *denotational semantics* [106].)

When the typing language is a logic, a type checker reads a syntax tree as a "proof" of a "proposition," namely, the program's type — the type checker does proof checking [95]. [1]

The algorithm that attaches properties to the program tree might repeatedly traverse the tree and compute the types attached to the program locations until a convergence is achieved. (The iteration is a least-fixed-point computation, requires that the property language is partially ordered, and uses a join (union) operation to refine types.) This algorithm leads to the next analysis form, because it is a *structured data-flow analysis* [2].

Further, if the type language is a temporal logic, the iterative traversal computes the temporal properties that are valid at each node of the tree — from here, it is a small step to branching-time model checking on Kripke structures. Further examples of fixed-point computation on parse trees are discussed in the literature [40].

2.2 Data-Flow Analysis

Data-flow analysis predicts the "flow" of information through the locations of a program. The flow can be computed either forward or backward and is often set-like, e.g., predicting the set of arithmetic expressions that have been previously evaluated at a program location, or the set of variables that will be needed for future computation, or the set of variables that definitely have constant values, or the set of aliased pointers. Because the analysis over-approximates a program's possible execution sequences, its results are imprecise.

The information gathered by a data-flow analysis can be used to validate safety properties or to help a compiler generate efficient target code. For example, if a

[1] The connection between model checking and theorem proving is discussed in Chap. 20.

constant-propagation analysis calculates that a variable has a known constant value at a program location, a constant-load instruction can replace the storage-lookup instruction.

Program Representation. A program is portrayed as a *directed graph*, whose nodes represent program locations. An edge connects two nodes if execution can transfer control from one location to the next; the initial node represents the program entry; final nodes represent the program exits. The edges are labeled by the primitive action (assignment, test expression) that transfers control — the directed graph is a *control-flow automaton* (CFA)[2], which displays the program's operations and its semantics of control. Figure 1 shows an example C function (a) together with its CFA (b). The CFA might be further condensed [9] or unrolled ("expanded") [115] so that more precise properties can be computed for its nodes.

Property Representation. Data-flow analysis annotates the CFA's locations with properties, which are usually sets that have semantic significance. A program location's property set might indicate the variables or expressions whose values were transferred along a control path to the program location [78]. For example, an available-expressions analysis predicts which arithmetic expressions (that appear in the program's text) will definitely be evaluated and be ready for use at subsequent program locations. The *property language* for available-expressions analysis is the collection of all subsets of expressions that appear in the program.

Another example is a constant-propagation analysis, which predicts which variables possess specific, constant values at the program locations. The property language consists of sets of the form $\{(\mathtt{x}_0, c_0), \cdots, (\mathtt{x}_n, c_n)\}$, where each $\mathtt{x}_i$ is a variable name in the program, all $\mathtt{x}_i$ are unique, and c_i is a fixed, constant value (e.g., 1.5 or 2) *or* the "unspecific value" $\top$, which indicates that $\mathtt{x}_i$ is defined but cannot be validated as having a constant value.

Analysis Algorithm. An iterative algorithm computes the properties that annotate the program locations. Starting from an initial configuration, where some program locations are annotated with input information, the algorithm propagates the properties along the CFA's edges. Recall that each edge from program location l to program location l' is labeled by an action. The semantics of the action is defined by a *transfer function* $f_{l \to l'}$, which defines how properties are updated when they traverse the edge [38]. There are two important versions of the analysis algorithm [2, 77, 78]:

1. *Maximal Fixed Point (MFP)*: Properties for each program location are computed directly on the CFA. The information computed for program location l' is defined by an equation of the following form [78]:

[2] Although in principle equivalent to the classical control-flow graphs [2], assigning the program operations to the edges is more compatible with the model-checking view (cf. the more detailed discussion by Steffen [113, 114]). The notion of control-flow automata is meanwhile established as a standard representation for programs (cf. the implementation in BLAST [13] and other verifiers).

$$propertyAtNode(l') = \bigcup_{l \in pred(l')} f_{l \to l'}(propertyAtNode(l))$$

where $pred(l')$ is the set of all predecessor locations for location l' in the CFA, and $f_{l \to l'}$ is the transfer function. The equations for the CFA's program locations are initialized to a minimal value, e.g., $\{\}$, and are iterated until they stabilize. To ensure finite convergence, the property language can be made finite and partially ordered (say, by subset, $\subseteq$). More generally, the property language can be a lattice of finite height [38].

2. *Meet Over all Paths (MOP)*: The iterative analysis algorithm enumerates the *paths within the CFA*. Properties are computed for all program locations on each individual path, and the results of all the analyses on all paths are joined (unioned). Since a program might have an infinite number of paths, path generation must be made convergent, say by bounding the expansion of loops[3] and procedure calls.

This example shows the distinction: For the program

```
1 if (...) {
2   x = 2; y = 3;
3 } else {
4   x = 3; y = 2;
5 }
6 z = x + y;
7 ;
```

where each program location l_i corresponds to the line i before the line is executed, an MOP analysis enumerates this path set: $\{l_1\, l_2\, l_3\, l_6\, l_7,\ l_1\, l_4\, l_5\, l_6\, l_7\}$. If the properties that are computed are the constant values (constant propagation), the MOP analysis generates these properties for the first path:

$$\begin{array}{ll} L_{1a} = \{\} & L_{3a} = \{(\mathtt{x},2),(\mathtt{y},3)\} \\ L_{2a} = \{\} & L_{6a} = \{(\mathtt{x},2),(\mathtt{y},3)\} \\ & L_{7a} = \{(\mathtt{x},2),(\mathtt{y},3),(\mathtt{z},5)\} \end{array}$$

and these for the second path:

$$\begin{array}{ll} L_{1b} = \{\} & L_{5b} = \{(\mathtt{x},3),(\mathtt{y},2)\} \\ L_{4b} = \{\} & L_{6b} = \{(\mathtt{x},3),(\mathtt{y},2)\} \\ & L_{7b} = \{(\mathtt{x},3),(\mathtt{y},2),(\mathtt{z},5)\}. \end{array}$$

The sets are joined ($L_k = L_{ka} \sqcup L_{kb}$), giving the following annotations for the labels:

$$\begin{array}{ll} L_1 = \{\} & L_3 = \{(\mathtt{x},2),(\mathtt{y},3)\} \\ L_2 = \{\} & L_5 = \{(\mathtt{x},3),(\mathtt{y},2)\} \\ L_4 = \{\} & L_6 = \{(\mathtt{x},\top),(\mathtt{y},\top)\} \\ & L_7 = \{(\mathtt{x},\top),(\mathtt{y},\top),(\mathtt{z},5)\}. \end{array}$$

The analysis determines that `z` is constant 5 at l_7. In contrast, an MFP analysis calculates the properties directly on the program locations, like this:

[3] More details on treating loops during the construction of program invariants are given in Sect. 6.

$$\begin{array}{ll} L_1 = \{\} & L_3 = \{(\mathtt{x},2),(\mathtt{y},3)\} \\ L_2 = \{\} & L_5 = \{(\mathtt{x},3),(\mathtt{y},2)\} \\ L_4 = \{\} & L_6 = L_3 \sqcup L_5 = \{(\mathtt{x},\top),(\mathtt{y},\top)\} \\ & L_7 = \{(\mathtt{x},\top),(\mathtt{y},\top),(\mathtt{z},\top)\}. \end{array}$$

In particular, the value of z at program location l_7 is calculated from the set L_6, and the transfer function for z=x+y computes $\top + \top = \top$. The example shows that an MOP analysis can be more precise than an MFP analysis, but the two results coincide if the transfer functions distribute over $\sqcup$ [77]. In practice, a hybrid approach is often taken, where the MFP algorithm is augmented by a limited program expansion and MOP computation, e.g., "property-oriented expansion" [22, 28, 115].

Extensions. The previous presentation used forward analysis, where input properties generate output properties, which are combined with union as the join operation. Iteration-until-convergence is a least-fixed-point calculation. It is possible to compute properties in a backward analysis (e.g., definitely-live-variables analysis), where intersection is the join operation; this is usually a greatest-fixed-point calculation (cf. Sect. 6.1).

Some analyses, e.g., partial redundancy elimination (PRE) [89] use both forward and backward analysis. PRE can be simplified into a two-step fixed-point computation, where the results of a backward analysis are complemented and used as input to a forward analysis [114]. This reformulation reveales several crucial insights:

1. A program's control-flow automaton can be defined as a Kripke structure, and expansion (unrolling) of the automaton is analogous to exploring the program's state space.
2. The value sets computed by a data-flow analysis can be represented as temporal-logic formulas, where the meaning of a formula is a value set.
3. The MFP algorithm operates like the algorithm for branching-time model checking, and the MOP algorithm operates like the algorithm for linear-time model checking.

These insights were documented earlier [81, 114, 115], and form the foundation for the remainder of this chapter.

2.3 Model Checking

Model checking enumerates the sequences of states that arise during a program's execution and decides whether the sequences of states satisfy a safety property. Other chapters in this Handbook develop several variants of the notions *program*, *property*, and *algorithm* for model checking, so here we merely compare and contrast model checking to data-flow analysis.

Program Representation. Rather than a syntax tree or a control-flow automaton, classic model checking operates on a directed graph whose nodes are the program's

run-time states, connected by edges that define sequencing. The directed graph is typically infinite and must be represented by a recursively enumerable set of transition rules. There are three variants of the transition rules:

1. A *Kripke structure* (S,R,I) consists of a set S of states, a transition relation $R \subseteq S \times S$, and a map $I : S \to 2^{\Phi}$ that assigns to each state a subset of properties from property set Φ that hold for the state.
2. A *labeled transition system* $(S,Act,\to)$ consists of a set S of states, a set Act of actions (transfer functions), and a transition relation $\to \subseteq S \times Act \times S$.
3. A *Kripke transition system* $(S,Act,\to,I)$ combines the components of the two previous forms.

The details needed to represent even one state can be practically prohibitive, therefore states are often abstracted by forgetting details of the state's "content" or even by replacing a state by a set of propositions that hold true for the state — this is often done with the Kripke-structure representation.

At the other extreme, if the set S of states is defined as exactly the program locations, then a control-flow automaton is readily expressed [114] and program expansion is easily done [115]. The notion of *abstract reachability graphs* (ARGs) is often used in the context of software verification (cf. BLAST [13]).

Property Representation. Model checking uses a temporal logic as its property language — it is a logic because it includes conjunction, disjunction, and (usually) negation; it is temporal because it uses operators that are interpreted on the sequences of nodes in a path or graph. The properties might be

1. *path-based (linear time)*, e.g., $E\phi$ might mean "there exists a state along a path that validates proposition ϕ," or
2. *graph-based (branching time)*, e.g., $EF\psi$ might mean "there exists a path generated from the current state that includes a state that validates ψ."

More details about temporal logics are provided in Chap. 2. The two variants recall the property languages used for MOP- and MFP-based data-flow analyses. The connection stands out when one reconsiders classic definitions, like this one for MFP-based live-variable calculation [78]:

$$LiveVarsAt(l) = UsedAt(l) \cup \left(NotModifiedAt(l) \cap \left(\bigcup_{l' \in succ(l)} LiveVarsAt(l') \right) \right)$$

where l is a program location and $succ(l)$ is the set of all successor locations for location l in the control-flow automaton. The equation defines the set of possibly live variables at a program location l. Compare the definition to the following, coded in branching-time temporal logic, which holds for a program location when variable x is possibly live at that program location [107, 113]:

$$isLiveVar_{\mathtt{x}} = isUsed_{\mathtt{x}} \vee (\neg isModified_{\mathtt{x}} \wedge EF(isLiveVar_{\mathtt{x}})) .$$

We have that $\mathtt{x} \in \mathit{LiveVarsAt}(l)$ iff $l \models \mathit{isLiveVar}_{\mathtt{x}}$ for every program variable $\mathtt{x}$ [113] — *the temporal-logic formula defines the data-flow set*.

Analysis Algorithm. A model-checking algorithm answers queries, posed in temporal logic, about a program representation. The algorithm generates a graph or path set from the program representation (transition rules) and applies the interpretation function to the nodes in the graph (respectively, paths) to answer the query. There are numerous algorithms for performing this activity, but with the perspective provided in this chapter, we can say that a model-checking algorithm is an MFP (resp., MOP) calculation of the graph (resp., paths) generated from a program's transition rules for answering the branching-time (resp., linear-time) query.

The generated graph or paths might be infinite, thus answering queries is semi-decidable. A bound can be placed on the number of iterations or graph size ("bounded model checking") or a join operation ("widening" [38]) might be used to force the generated graph to be finite. (When the latter is used, the technique is sometimes called "abstract model checking.")

The remainder of this chapter develops several variations of *property* and *algorithm* that are inspired by the deep correspondence between data-flow analysis and model checking.

3 Unifying Formal Framework/Comparison of Algorithms

The previous section has outlined the differences, and similarities, between the three static-analysis techniques type checking, data-flow analysis, and model checking. The discussion was structured by the components of every static analysis: program representation, property representation, and analysis algorithm. In the following, we explain the unifying formal framework of *configurable program analysis*, which has successful implementations in software-verification tools (CPACHECKER [21], CPALIEN [91], CPATIGER [20], JAKSTAB [73]). The framework makes it possible to formalize each of the three techniques in the same formal setting.[4] In order to concretely explain the differences, we model the algorithms that were traditionally used for data-flow analysis and software model checking as instances of the framework.

3.1 Preliminaries

Control-Flow Automaton (CFA). A *program* is represented by a *control-flow automaton*. We restrict our formal presentation to simple imperative programming, where all operations are either assignments or assume operations (conditional executions),

[4] Other approaches have been proposed that address similar goals, for example, the fixed-point analysis machine [79, 80, 116].

all variables range over integers, and no function calls occur,[5] while we use C syntax to denote example program code. A CFA (L, l_0, G) consists of a set L of program locations (models the program counter *pc*), an initial program location l_0 (models the program entry), and a set $G \subseteq L \times Ops \times L$ of control-flow edges (models program operations that are executed when control flows from one program location to another). Program operations from Ops are either assignment operations or assume operations. The set of program variables that occur in program operations from Ops is denoted by X. A *concrete state* of a program is a variable assignment c that assigns a value to each variable from $X \cup \{pc\}$. 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 $\xrightarrow{g} \subseteq C \times \{g\} \times C$, which defines how concrete states of one program location (source) are transformed into concrete states of another program location (target). The complete transition relation $\rightarrow$ is the union over all control-flow edges: $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$. We write $c \xrightarrow{g} c'$ if $(c, g, c') \in \rightarrow$, and $c \rightarrow c'$ 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 $c_{i-1} \rightarrow c_i$.

Example 1. Figure 1 shows an example program (a) and the corresponding CFA (b). The CFA has seven program locations ($L = \{2,3,4,5,7,9,10\}, l_0 = 2$) and three program variables ($X = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$). The initial region r_0 of this program is the set $\{c \in C \mid c(pc) = 2\}$. The only concrete state at program location 5 (i.e., before line 5 is executed) that is reachable from the initial region is the following variable assignment: $c(pc) = 5, c(\mathtt{x}) = 0, c(\mathtt{y}) = 1, c(\mathtt{z}) = 0$. The set of concrete states at program location 9 that are reachable from the initial region can be represented by the predicate $pc = 9 \wedge ((\mathtt{x} = 1 \wedge \mathtt{y} = 1 \wedge \mathtt{z} = 0) \vee (\mathtt{x} = 0 \wedge \mathtt{y} \neq 1 \wedge \mathtt{z} = 1))$.

Semi-lattices. A *partial order* $\sqsubseteq \subseteq E \times E$ over a (possibly infinite) set E is a binary relation that is reflexive ($e \sqsubseteq e$ for all $e \in E$), transitive (if $e \sqsubseteq e'$ and $e' \sqsubseteq e''$ then $e \sqsubseteq e''$), and antisymmetric (if $e \sqsubseteq e'$ and $e' \sqsubseteq e$ then $e = e'$). The *least upper bound* for a subset $M \subseteq E$ of elements is the smallest element e such that $e' \sqsubseteq e$ for all $e' \in M$. The partial order $\sqsubseteq$ induces a *semi-lattice*[6] (defines the structure of the semi-lattice) if every subset $M \subseteq E$ has a least upper bound $e \in E$ (cf. [94] for more details). We denote a semi-lattice that is induced by a set E and a partial order $\sqsubseteq$ using the tuple $(E, \sqsubseteq, \sqcup, \top)$, in order to assign symbols to special components: the join operator $\sqcup : E \times E \rightarrow E$ yields the least upper bound for two elements (we use the set notation $\bigsqcup\{e_1, e_2, \ldots\}$ to denote $e_1 \sqcup e_2 \sqcup \ldots$) and the top element $\top$ is the least upper bound of the set E ($\top = \sqcup E$).

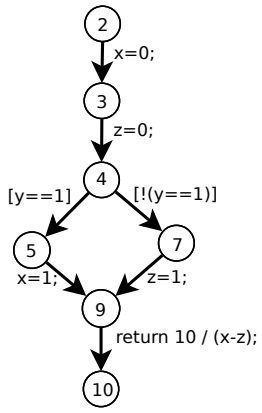
Example 2. Let us consider the semi-lattice $(V, \sqsubseteq, \sqcup, \top)$ that can be used for a constant-propagation analysis over two Boolean variables. The set V of lattice elements consists of variable assignments:

[5] Tool implementations usually support interprocedural analysis, either via function inlining, function summaries, or other techniques [100, 116]. More information on this topic is given in Chap. 17.

[6] Sometimes, complete lattices are used in formalizations of data-flow analyses, but most practical analyses require only one operator: either the least upper bound or the greatest lower bound.

```
1  int foo(int y) {
2    int x = 0;
3    int z = 0;
4    if (y == 1) {
5      x = 1;
6    } else {
7      z = 1;
8    }
9    return 10 / (x - z);
10 }
```

(a) Example C function



(b) Control-flow automaton (CFA)

Fig. 1 Example C function and corresponding CFA; the program locations in the CFA (b) correspond to the line numbers in the program text (a) before the line of code is executed

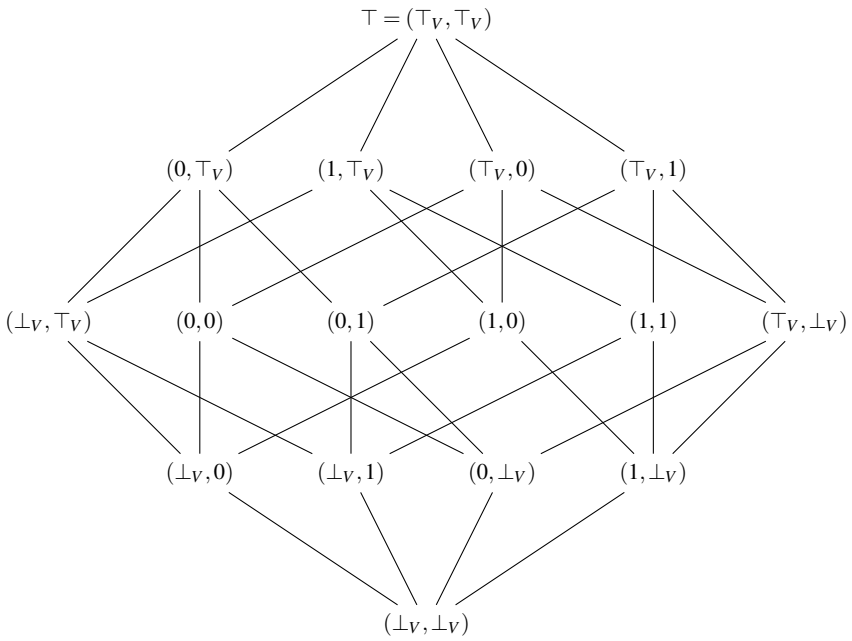


Fig. 2 Example lattice

$V = X \to \{\bot_V, 0, 1, \top_V\}$, $X = \{x_1, x_2\}$. The partial order $\sqsubseteq$ is defined as $v \sqsubseteq v'$ if $\forall x \in X : v(x) = v'(x)$ or $v(x) = \bot_V$ or $v'(x) = \top_V$. Figure 2 depicts this simple lattice as a graph. The nodes represent lattice elements, where a pair (c_1, c_2) denotes the variable assignment $\{x_1 \mapsto c_1, x_2 \mapsto c_2\}$. The edges represent the partial order (if read in the upwards direction), where reflexive and transitive edges are omitted. The top element $\top$ is the variable assignment with $\top(x) = \top_V$ for all $x \in X$.

Program Analysis. A *program analysis* for a CFA (L, l_0, G) consists of an abstract domain D and a transfer relation $\rightsquigarrow$. The *abstract domain* $D = (C, \mathscr{E}, [\![\cdot]\!])$ is defined by the set C of concrete states, a semi-lattice $\mathscr{E} = (E, \sqsubseteq, \sqcup, \top)$, and a concretization function $[\![\cdot]\!]$. The lattice elements are used as components of abstract states in the program analysis. Each abstract state represents a (possibly infinite) set of concrete states. The concretization function $[\![\cdot]\!] : E \to 2^C$ 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, i.e., the aspects of the program that are analyzed. 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 $e \overset{g}{\rightsquigarrow} e'$. A program analysis has to fulfill certain requirements for soundness, i.e., to guarantee that no violations of the property are missed by the analysis [17, 38, 94].

Example 3. Considering the example in Fig. 2 again, the concretization function $[\![\cdot]\!]$ relates the lattice elements to sets of variable assignments. For example, lattice element $(1, 0)$ maps the first variable to value 1 and the second variable to value 0. The lattice element $\top$ represents all concrete states. Given a variable x, we use the bottom element $\bot_V$ to denote the variable assignment that assigns no value to variable x. Note that in a program analysis, there might be several (strictly speaking different) lattice elements that represent the empty set of concrete states: every variable assignment that has (at least) one variable assigned to $\bot_V$ cannot represent any concrete state.[7]

3.2 Algorithm of Data-Flow Analysis

We now present an iteration algorithm for MFP data-flow analysis. According to classic definitions of data-flow analysis [94], the algorithm computes, for a given abstract domain, a function reached that assigns to each analyzed program location an abstract data state (i.e., the abstract states consist of a program location and an abstract data state, the latter represented by a lattice element).

[7] This leads to the notion of "smashed bottom," where all variable assignments with at least one variable assigned to $\bot_V$ are subsumed by one representative ($\bot$). We do not emphasize this notion in our chapter.

Algorithm 1 Typical differences of data-flow analysis (Algorithm *DFA*) and software model checking (Algorithm *Reach*)

Input: set L of locations, an abstract domain E, transfer relation $\rightsquigarrow$,
initial abstract state (l_0, e_0) with $l_0 \in L, e_0 \in E$
Output: set of reachable abstract states (pairs of location and abstract data state)

(a) Algorithm $DFA(L, E, \rightsquigarrow, e_0)$

Variables: function reached : $L \rightharpoonup E$,
set waitlist $\subseteq L$

```
 1: waitlist := {l_0}
 2: reached(l_0) := e_0
 3: while waitlist ≠ {} do
 4:    choose l from waitlist
 5:    waitlist := waitlist \ {l}
 6:    for each (l', e') with (l, e) ⇝ (l', e') do
 7:       // if not already covered
 8:       if e' ⋢ reached(l') then
 9:          // join with existing abstract data state
10:          reached(l') := reached(l') ⊔ e'
11:          waitlist := waitlist ∪ {l'}
12: return reached
```

(b) Algorithm $Reach(L, E, \rightsquigarrow, e_0)$

Variables: set reached $\subseteq L \times E$,
set waitlist $\subseteq L \times E$

```
 1: waitlist := {(l_0, e_0)}
 2: reached := {(l_0, e_0)}
 3: while waitlist ≠ {} do
 4:    choose (l, e) from waitlist
 5:    waitlist := waitlist \ {(l, e)}
 6:    for each (l', e') with (l, e) ⇝ (l', e') do
 7:       // if not already covered
 8:       if ∄(l', e'') ∈ reached : e' ⊑ e'' then
 9:          // add as new abstract state
10:          reached := reached ∪ {(l', e')}
11:          waitlist := waitlist ∪ {(l', e')}
12: return reached
```

Algorithm 1(a) operates on a partial function and a set: the function reached represents the result of the data-flow analysis, i.e., the mapping from program locations to abstract data states; the set waitlist represents the program locations for which the abstract data state was changed, i.e., the fixed point is not reached as long as waitlist is not empty. Algorithm 1(a) is guaranteed to terminate if the semi-lattice has finite height; the run time depends on the height of the semi-lattice and the number of program locations. The algorithm starts by assigning the initial abstract data state to the initial program location. Then it iterates through the `while` loop until the set waitlist is empty. In every loop iteration, one program location is taken out of the waitlist and abstract successors are computed for the corresponding successor program locations. The abstract data element for the successor program location in function reached is added for the program location, or the old abstract data state is replaced by the join of the old and new abstract data states. Because we operate on a partial function reached, we extend $e' \not\sqsubseteq \mathsf{reached}(l')$ to return *false* if $\mathsf{reached}(l')$ is undefined, and we extend $\mathsf{reached}(l') \sqcup e'$ to $\bigsqcup(\{e'' \mid (l', e'') \in \mathsf{reached}\} \cup e')$.[8]

Example 4. Consider the example program from Fig. 1 and an abstract domain for constant propagation; suppose the verification task is to ensure that no division by zero occurs. The data-flow analysis computes a function reached with the following entries: $2 \mapsto \{\mathtt{x} = \top, \mathtt{y} = \top, \mathtt{z} = \top\}$, $3 \mapsto \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = \top\}$, and $4 \mapsto \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 0\}$. Following the `then` branch from program location 4, the algorithm computes the entries $5 \mapsto \{\mathtt{x} = 0, \mathtt{y} = 1, \mathtt{z} = 0\}$ and $9 \mapsto \{\mathtt{x} = 1, \mathtt{y} = 1, \mathtt{z} = 0\}$, and stores them in the function reached. For the `else`

[8] Alternative formalizations use total functions for reached and require some lower bound $\bot$ to exist in the (semi-) lattice, which is used as the initial abstract state to make reached total.

branch, the algorithm computes the entries $7 \mapsto \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 0\}$ and $9 \mapsto \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 1\}$. Since reached already has an entry for program location 9, the two abstract data states are joined, which results in the entry $9 \mapsto \{\mathtt{x} = \top, \mathtt{y} = \top, \mathtt{z} = \top\}$. The correctness of the program (in terms of division by zero) cannot be established.

3.3 Algorithm of Model Checking

We now consider an iteration algorithm for software model checking. According to the classic reachability algorithm, the algorithm computes the nodes of an abstract reachability tree [13], which contains all reachable abstract states according to the transfer relation. In difference to the data-flow analysis, the join operation is never applied.

Algorithm 1(b) operates on two sets reached and waitlist, which are initialized with a pair of the initial control-flow location and the initial abstract data state. In every iteration of the `while` loop, the algorithm takes one abstract state from the set waitlist and computes successors, as long as the fixed point is not reached. Algorithm 1(b) is not guaranteed to terminate if the semi-lattice is infinite; software model checking in general is a semi-decidable analysis. If there is no abstract state in the set reached that entails the new abstract state, then the new abstract state is added to the sets reached and waitlist. The join operation is never called, and thus, the set of reached abstract states contains all nodes that an abstract reachability tree (ART) [13] would contain (the edges of the actual tree are not necessarily stored; but many model-checking algorithms do store an ART to support certain features, such as error-path analysis [34]).

Example 5. We reconsider the example program from Fig. 1 and an abstract domain for constant propagation. The model-checking algorithm computes a set reached with the following entries: $(2, \{\mathtt{x} = \top, \mathtt{y} = \top, \mathtt{z} = \top\})$, $(3, \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = \top\})$, and $(4, \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 0\})$. Following the `then` branch from program location 4, the algorithm computes the entries $(5, \{\mathtt{x} = 0, \mathtt{y} = 1, \mathtt{z} = 0\})$ and $(9, \{\mathtt{x} = 1, \mathtt{y} = 1, \mathtt{z} = 0\})$, and stores them in the set reached. For the `else` branch, the algorithm computes the entries $(7, \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 0\})$ and $(9, \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 1\})$. Although reached already has an entry for program location 9, this second entry is stored in the set reached, and the correctness of the example program (in terms of division by zero) is established: the value of variable x is always different from the value of variable z.

3.4 Unified Algorithm Using Configurable Program Analysis

In theory, data-flow analysis and model checking have the same expressive power [108]. In this section, we explain the unifying framework of configurable pro-

gram analysis [17,18], a formalism and algorithm that makes it possible to *practically* unify the approaches. Comparing the two Algorithms 1(a) and (b) again reveals the similarity that motivates a unified algorithm, and also the differences that motivate the configurable operators merge and stop, which we will define below as part of the configurable program analysis and then use in the unified Algorithm 2.

Configurable Program Analysis (CPA). A *configurable program analysis* $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ for a CFA (L, l_0, G) 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* $D = (C, \mathscr{E}, [\![\cdot]\!])$ is defined by the set C of concrete states, a semi-lattice $\mathscr{E} = (E, \sqsubseteq, \sqcup, \top)$, and a concretization function $[\![\cdot]\!]$.

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.

3. The *merge operator* $\mathsf{merge} : E \times E \to E$ combines the information of two abstract states. The operator *weakens* the abstract state (also called *widening*) that is given as second parameter depending on the first parameter (the result of $\mathsf{merge}(e, e')$ can be anything between e' and $\top$).
Note that the operator merge is not commutative, and is not necessarily 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 whether the abstract state e that is given as first parameter is covered by the set R of abstract states given as second parameter, i.e., every concrete state that e represents is represented by some abstract state from R. 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 power-set domain[9] — can join the elements of R to check whether $\bigsqcup R$ subsumes the first parameter.
Note that the termination check stop is not the same as the partial order $\sqsubseteq$ of the lattice, but stop can be based on $\sqsubseteq$. Later we will use the following termination checks (the second requires a power-set domain): $\mathsf{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ and $\mathsf{stop}^{join}(e, R) = (e \sqsubseteq \bigsqcup R)$.

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 contribute to adjusting both precision and cost of the analysis.

[9] A *power-set domain* is an abstract domain such that $[\![e_1 \sqcup e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$.

Algorithm 2 *CPA*$(\mathbb{D}, e_0)$

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

```
 1: waitlist := {e_0}
 2: reached := {e_0}
 3: while waitlist ≠ {} do
 4:    choose e from waitlist
 5:    waitlist := waitlist \ {e}
 6:    for each e' with e ⇝ e' do
 7:       for each e'' ∈ reached do
 8:          // combine with existing abstract state
 9:          e_new := merge(e', e'')
10:          if e_new ≠ e'' then
11:             waitlist := (waitlist ∪ {e_new}) \ {e''}
12:             reached := (reached ∪ {e_new}) \ {e''}
13:       if ¬ stop(e', reached) then
14:          waitlist := waitlist ∪ {e'}
15:          reached := reached ∪ {e'}
16: return reached
```

Unified Algorithm. In order to experiment with both data-flow analysis and model checking in one single algorithm, we unify the two algorithms using the operators merge and stop of the configurable program analysis.

Algorithm 2 abstracts from program locations and operates on two sets of abstract states (abstract-domain elements), i.e., the program location is represented in the abstract domain and is not specially treated anymore (classic data-flow analyses rely on an explicit representation of the program location). The sets reached and waitlist are initialized with the initial abstract state for the given configurable program analysis. As in the previous algorithms, every iteration of the loop processes one element from the set waitlist, and computes all abstract successors for that abstract state. The set waitlist is empty if the fixed point of the iteration is reached.

Now, for every abstract successor state, the algorithm merges the new abstract state with every existing abstract state in the set reached. It depends solely on the merge operator how often abstract states from reached are combined and how abstractly they are combined. In the case that the merge operator does not produce a new combined state, it simply returns the existing abstract state that was given as second parameter. Otherwise, it returns a new abstract state that entails the existing abstract state. In the latter case, the existing abstract state is removed from the sets reached and waitlist and the new abstract state is added to the sets reached and waitlist. (Obviously, an efficient implementation of the algorithm applies optimization to the `for each` loop from line 7 to line 12, e.g., using partitions or projections for the set reached.)

After the current abstract successor state has been merged with all existing abstract states, the stop operator determines whether the algorithm needs to store the current abstract state in the sets reached and waitlist. For example, if all concrete states that

are represented by the current abstract state are covered (i.e., also represented) by existing abstract states in the set reached, then the current state may be ignored.

The set reached, at the fixed point of the iteration, represents the program invariant. Such fixed-point iterations and several other algorithms for constructing program invariants are discussed in Sect. 6.

3.5 Discussion

Effectiveness. The *effectiveness* of an analysis refers to the degree of precision with which the analysis determines whether a program satisfies or violates a given specification (number of false positives and false negatives). Model checking has a high degree of precision, due to the fact that all reachable abstract states are stored separately in the set of reachable states, i.e., model checking is automatically path-sensitive due to never applying join operations (if the set of reachable abstract states is seen as the reachability tree that represents execution paths). Data-flow analysis is often imprecise, when join operations are applied in order to reduce two abstract states to one. In comparison to standard data-flow analysis, power-set constructions for increasing the precision (e.g., for making an analysis path-sensitive) are not necessary in a configurable program analysis: the effect can easily be achieved by setting the merge operator to not join.

This is the strength of defining program analyses as CPA: the components abstract domain, transfer relation, merge operator, and stop operator separate concerns and provide a flexible way of tuning these components or exchanging them with others. For example, the merge operator encodes whether the algorithm works like MFP, or MOP, or uses a hybrid approach (cf. the merge operator used in adjustable-block encoding [22]). Each of the components has an important impact on the precision and performance of the program analysis.

Efficiency. The *efficiency* (also called performance) of an analysis measures the resource consumption of an algorithm (in time or space). The resources required for an analysis often decide whether the analysis should be applied to the problem or not. For example, the run time of a data-flow analysis is determined by the height of the abstract domain's lattice, the size of the control-flow automaton, and the number of variables in the program. Most of the classic data-flow analyses are efficient (low polynomial run time) and can be used in compilers for optimization. Model checking sometimes requires resources exponential in the program size (if terminating at all). Due to the high precision of typical model-checking domains, such as predicate abstraction, the sub-problem of computing an abstract successor state is often NP-hard already.

Iteration Order. The *iteration order* defines the sequence in which abstract states from the set waitlist are processed by the exploration algorithm. We did not discuss

this parameter because it is orthogonal to the difference between data-flow analysis and model checking, i.e., most iteration orders can be used for both techniques. In Algorithm 2, the iteration order is implemented in the operator choose. The most simple iteration orders are breadth-first search (BFS) and depth-first search (DFS). The iteration order DFS is often not advisable for data-flow analysis, because after each join operation, the algorithm has to re-explore all successors of abstract states that represent more concrete states after the join. For model checking, both orders are applicable, while some existing implementations of model-checking tools prefer the DFS order (e.g., BLAST [13]). The best iteration order is often a combination of both, for example by using a topological (reverse post-order) algorithm in which DFS is performed until a meet point, while further exploration has to wait for the control flow to arrive via all other branches [22]. Also chaotic iteration orders [29] were investigated and found to be useful. More details can be found in Sect. 6.1.

3.6 Composition of Configurable Program Analyses

Different CPAs have different strengths and weaknesses, and therefore, we need to construct combinations of component analyses to pick the advantages of several components, in order to achieve more effective program analyses.

Composite. A configurable program analysis can be composed of several configurable program analyses [17]. A *composite program analysis* $\mathscr{C} = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$[10] consists of two configurable program analyses $\mathbb{D}_1$ and $\mathbb{D}_2$ sharing the same set of concrete states with E_1 and E_2 being their respective sets of abstract states, a composite transfer relation $\rightsquigarrow_\times \subseteq (E_1 \times E_2) \times G \times (E_1 \times E_2)$, a composite merge operator $\mathsf{merge}_\times : (E_1 \times E_2) \times (E_1 \times E_2) \to (E_1 \times E_2)$, and a composite termination check $\mathsf{stop}_\times : (E_1 \times E_2) \times 2^{E_1 \times E_2} \to \mathbb{B}$. The three composites $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$ are expressions over the components of $\mathbb{D}_1$ and $\mathbb{D}_2$ $(\rightsquigarrow_i, \mathsf{merge}_i, \mathsf{stop}_i, [\![\cdot]\!]_i, E_i, \sqsubseteq_i, \sqcup_i, \top_i)$, 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 $\preceq$.

Strengthen. 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 direct product relation. For example, if we combine predicate analysis and constant propagation, the strengthening operator $\downarrow_{\mathbb{CO},\mathbb{P}}$ can "sharpen" the explicit-value assignment of the

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

constant propagation (cf. Sect. 4.2) by considering the predicates in the predicate analysis (cf. Sect. 4.4).

Compare. 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.

Composition. For a given composite program analysis $\mathscr{C} = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, we can construct the 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: $D_\times = D_1 \times D_2 = (C, \mathscr{E}_\times, [\![\cdot]\!]_\times)$. The product lattice is $\mathscr{E}_\times = \mathscr{E}_1 \times \mathscr{E}_2 = (E_1 \times E_2, \sqsubseteq_\times, \sqcup_\times, (\top_1, \top_2))$ 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 the following holds $(e_1, e_2) \sqcup_\times (e_1', e_2') = (e_1 \sqcup_1 e_1', e_2 \sqcup_2 e_2')$). 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 [36,39]. Even improvements over the direct product (e.g., the reduced product [28, 39] or the logical product [61]) 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 definitions of the three composite operators determine 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. For certain numerical abstract domains, the composite transfer relation can be automatically constructed: if the abstract domains of two given CPAs fulfill certain requirements (convex, stably infinite, disjoint) then the most precise abstract transfer relation can be computed [61].

4 Classic Examples (Component Analyses)

We now define and explain some well-known classic example analyses, in order to demonstrate the formalism of configurable program analysis. We use the notations that were introduced in Sects. 3.1, 3.4, and 3.6.

4.1 Reachable-Code Analysis

The reachable-code analysis (also known as dead-code analysis) identifies all locations of the control-flow automaton that can be reached from the program entry location. This classic analysis tracks only syntactic reachability, i.e., the operations are not interpreted.

The *location analysis* is 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 program locations and consists of the following components:

1. The abstract domain $D_{\mathbb{L}}$ is based on the semi-lattice for the set L of program locations: $D_{\mathbb{L}} = (C, \mathscr{L}, [\![\cdot]\!])$, with $\mathscr{L} = (L \cup \{\top\}, \sqsubseteq, \sqcup, \top)$ (also called a "flat semi-lattice"), $l \sqsubseteq \top$, and $l \neq l' \Rightarrow l \not\sqsubseteq l'$ for all elements $l, l' \in L$ (this implies $\top \sqcup l = \top, l \sqcup l' = \top$ for all elements $l, l' \in L, l \neq l'$), and $[\![\top]\!] = C$, and for all $l \in L$: $[\![l]\!] = \{c \in C \mid c(pc) = l\}$.
   The element $\top$ represents the fact that the program location is not known.
2. The transfer relation $\rightsquigarrow_{\mathbb{L}}$ has the transfer $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$ if $g = (l, op, l')$, and has the transfer $\top \overset{g}{\rightsquigarrow}_{\mathbb{L}} \top$ for all $g \in G$.
   The transfer relation determines the syntactic successor in the CFA without considering the semantics of the operation *op*.
3. The merge operator does not combine elements when the control flow meets: $\mathsf{merge}_{\mathbb{L}} = \mathsf{merge}^{sep}$.
4. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{L}} = \mathsf{stop}^{sep}$.

This (simple) abstract domain can be used to perform a syntactic reachability analysis, for example to eliminate control-flow operations that can never be executed. More importantly, this CPA can be used to track the program location when combined with other CPAs, in order to separate the concern of location tracking from other analyses. In practice, a semantic reachable-code analysis would be preferred to search for dead code, for example using a predicate analysis, as was done in the context of model-checking-based test-case generation [7]. More details about the connection between model checking and testing are provided in Chap. 19.

4.2 Constant Propagation

The constant-propagation analysis identifies variables that store constant values at certain program locations, i.e., at a given program location, the value is always the same. This classic domain of data-flow analysis can be used to reduce the number of variables in a program by substituting constants for variables.

The *constant-propagation analysis* is a configurable program analysis $\mathbb{CO} = (D_{\mathbb{CO}}, \rightsquigarrow_{\mathbb{CO}}, \mathsf{merge}_{\mathbb{CO}}, \mathsf{stop}_{\mathbb{CO}})$ that tries to determine, for each program location, the value of each variable, and consists of the following components (we use the set L of program locations, the set $X \neq \{\}$ of program variables, and the set $\mathbb{Z}$ of integer values):

1. The abstract domain $D_{\mathbb{CO}} = (C, \mathscr{E}, [\![\cdot]\!])$ consists of the following three components. The set C is the set of concrete states. The semi-lattice $\mathscr{E}$ represents the abstract states, which store for a program location an abstract variable assignment. Formally, the semi-lattice $\mathscr{E} = ((L \cup \{\top_L\}) \times (X \to \mathscr{Z}), \sqsubseteq, \sqcup, (\top_L, v_\top))$, with $\mathscr{Z} = \mathbb{Z} \cup \{\top_{\mathscr{Z}}\}$, is induced by the partial order $\sqsubseteq$ that is defined as

$(l,v) \sqsubseteq (l',v')$ if $(l = l' \text{ or } l' = \top_L)$ and $\forall x \in X : v(x) = v'(x)$ or $v'(x) = \top_{\mathscr{Z}}$. (The join operator $\sqcup$ yields the least upper bound, and $v_\top$ is the abstract variable assignment with $v_\top(x) = \top_{\mathscr{Z}}$ for each $x \in X$.) A concrete state c matches a program location l if $c(pc) = l$ or $l = \top_L$. Similarly, a concrete state c is compatible with an abstract variable assignment v if for all $x \in X$, $c(x) = v(x)$ or $v(x) = \top_{\mathscr{Z}}$. The concretization function $[\![\cdot]\!]$ assigns to an abstract state (l,v) all concrete states that match the program location l and are compatible with the abstract variable assignment v.

2. The transfer relation $\rightsquigarrow_{\mathbb{CO}}$ has the transfer $(l,v) \overset{g}{\rightsquigarrow} (l',v')$ if
(1) $g = (l, \texttt{assume}(p), l')$ and $\phi(p,v)$ is satisfiable and for all $x \in X$:
$$v'(x) = \begin{cases} c & \text{if } c \text{ is the only satisfying assignment of } \phi(p,v) \text{ for variable } x \\ v(x) & \text{otherwise} \end{cases}$$
where, given a predicate p over variables in X and an abstract variable assignment v, we define $\phi(p,v) := p \wedge \bigwedge_{x \in X, v(x) \neq \top_{\mathscr{Z}}} x = v(x)$
or
(2) $g = (l, \texttt{w} := e, l')$ and for all $x \in X$:
$$v'(x) = \begin{cases} eval(e,v) & \text{if } x = \texttt{w} \\ v(x) & \text{otherwise} \end{cases}$$
where, given an expression e over variables in X and an abstract variable assignment v,
$$eval(e,v) := \begin{cases} \top_{\mathscr{Z}} & \text{if } v(x) = \top_{\mathscr{Z}} \text{ for some } x \in X \text{ that occurs in } e \\ z & \text{otherwise, where expression } e \text{ evaluates to } z \text{ when each} \\ & \text{variable } x \text{ is replaced by } v(x) \text{ in } e \end{cases}$$
or
(3) $l = l' = \top_L$ and $v' = v_\top$.

3. The merge operator is defined by
$$\mathsf{merge}_{\mathbb{CO}}((l,v),(l',v')) = \begin{cases} (l,v) \sqcup (l',v') & \text{if } l = l' \\ (l',v') & \text{otherwise} \end{cases}$$
(The two abstract variable assignments are combined where the control flow meets.)

4. The termination check is defined by $\mathsf{stop}_{\mathbb{CO}} = \mathsf{stop}^{sep}$.

Example 6. Consider the C function in Fig. 1(a) again, and construct a CPA for constant propagation. The following lattice element is an example of an abstract state that is reachable in the program code from Fig. 1(a): $(4, \{x \mapsto 0, y \mapsto \top_{\mathscr{Z}}, z \mapsto 0\})$.

Note that CPA $\mathbb{CO}$ performs an MFP computation, which is not precise enough for proving the correctness of the function in Fig. 1(a). If we change the merge operator $\mathsf{merge}_{\mathbb{CO}}$ to merge^{sep}, then we move from MFP to what corresponds to abstract reachability trees (never join). This changed analysis is similar to explicit-value analysis [24]. Explicit-state model checking is discussed in Chap. 5.

Example 7. Considering the example from Fig. 1 again, but using merge^{sep} as merge operator, we obtain the following two different abstract states for program location 9: $(9, \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 1, \texttt{z} \mapsto 0\})$ and $(9, \{\texttt{x} \mapsto 0, \texttt{y} \mapsto \top_{\mathscr{Z}}, \texttt{z} \mapsto 1\})$. This proves that a division by zero is not possible.

4.3 Reaching Definitions

The reaching-definitions analysis computes for every program location and for every variable a set of assignment operations that may have defined the value of the variable (i.e., definitions that "reach" the location). This classic domain of data-flow analysis (very similar to use-def analysis) is used in compiler optimization to infer dependencies between operations [2], and in code-structure analysis to quantitatively measure the data-flow [12]. Furthermore, in selective test-case generation [102], an efficient use-def analysis is necessary to determine which program locations need to be covered by a test case (for a given changed definition, we need to compute all uses of that definition).

An assigning CFA edge e is a *reaching definition* for program location l and variable x if there exists a path in the CFA through edge e to program location l without any (re-)definition of x.

The *reaching-definitions analysis* is a configurable program analysis $\mathbb{RD} = (D_{\mathbb{RD}}, \rightsquigarrow_{\mathbb{RD}}, \mathsf{merge}_{\mathbb{RD}}, \mathsf{stop}_{\mathbb{RD}})$, which computes the set of reaching definitions for each program location, and consists of the following components (X is the set of program variables):

1. The abstract domain $D_{\mathbb{RD}} = (C, \mathscr{E}, [\![\cdot]\!])$ consists of the set C of concrete states, the semi-lattice $\mathscr{E}$, and the concretization function $[\![\cdot]\!]$. The semi-lattice is given by $\mathscr{E} = \big((L \cup \{\top_L\}) \times 2^E, \sqsubseteq_{\mathbb{RD}}, \sqcup_{\mathbb{RD}}, (\top_L, E)\big)$, where $E \subseteq X \times (L \times L)$ is the set of definitions (variables paired with their defining edge) in the program, and we define $(l,S) \sqsubseteq_{\mathbb{RD}} (l',S')$ if $(l = l' \text{ or } l' = \top_L)$ and $S \subseteq S'$, which implies the join operator:
$(l,S) \sqcup_{\mathbb{RD}} (l',S') = \begin{cases} (l, S \cup S') & \text{if } l = l' \\ (\top_L, S \cup S') & \text{otherwise.} \end{cases}$

2. The transfer relation $\rightsquigarrow_{\mathbb{RD}}$ has the transfer $(l,S) \rightsquigarrow (l',S')$ if
(1) there exists a CFA edge $g = (l, op, l') \in G$ and
$S' = \begin{cases} (S \setminus \{(\mathtt{x}, k, k') \mid k, k' \in L\}) \cup \{(\mathtt{x}, l, l')\} & \text{if } op \text{ has the form } \mathtt{x} := \texttt{<expr>}; \\ S & \text{otherwise} \end{cases}$
or
(2) $l = l' = \top_L$ and $S' = E$.

3. The merge operator is defined as
$\mathsf{merge}_{\mathbb{RD}}((l,S),(l',S')) = \begin{cases} (l', S \cup S') & \text{if } l = l' \\ (l', S') & \text{otherwise.} \end{cases}$
(The two sets of reaching definitions are united where the control flow meets.)

4. The termination check is defined as $\mathsf{stop}_{\mathbb{RD}} = \mathsf{stop}^{sep}$.

Example 8. In the program of Fig. 1, variable x has the following reaching definitions at location 9: $\{(x,2,3),(x,5,9)\}$.

4.4 Predicate Analysis

For a given formula ϕ and a set π of predicates, the *Cartesian predicate abstraction* $(\phi)^{\pi}_{C}$ is the strongest conjunction of predicates from π that is implied by ϕ, and the *Boolean predicate abstraction* $(\phi)^{\pi}_{B}$ is the strongest Boolean combination of predicates from π that is implied by ϕ.

Predicate analysis is a program analysis that uses predicate abstraction to construct abstract states. The precision π of the predicate analysis is a finite set of predicates that controls the coarseness of the over-approximation of the abstract states. The precision can be refined during the analysis using CEGAR [34] and interpolation [69], and there can be different values for the precision at different program locations using lazy abstraction refinement [13, 71], however, for simplicity of presentation, we assume a fixed set of predicates. This classic domain of software model checking became popular and successful in the last decade due to the recent breakthroughs in decision procedures (SMT solvers) for Boolean formulas over expressions in the theory of linear arithmetic (LA) and equality with uninterpreted functions (EUF).

The Cartesian *predicate analysis* is a configurable program analysis $\mathbb{P} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{merge}_{\mathbb{P}}, \mathsf{stop}_{\mathbb{P}})$, which uses Cartesian predicate abstraction and consists of the following components (where the precision is given by the finite set π of predicates over the set X of program variables, with $\mathit{false} \in \pi$, that are tracked by the analysis; for a set $r \subseteq \pi$ of predicates, we write φ_r to denote the conjunction of all predicates in r, in particular $\varphi_{\{\}} = \mathit{true}$):

1. The domain $D_{\mathbb{P}} = (C, \mathscr{P}, [\![\cdot]\!])$ is based on the idea that regions are represented by conjunctions over a finite set of predicates. The semi-lattice is given as $\mathscr{P} = (2^{\pi}, \sqsubseteq, \sqcup, \top)$, where the partial order $\sqsubseteq$ is defined as $r \sqsubseteq r'$ if $r \supseteq r'$ (note that if $r \sqsubseteq r'$ then φ_r implies $\varphi_{r'}$). The least upper bound $r \sqcup r'$ is given by $r \cap r'$ (note that $\varphi_{r \sqcup r'}$ is implied by $\varphi_r \vee \varphi_{r'}$). The element $\top = \{\}$ leaves the abstract state unconstrained (*true*), i.e., every concrete state is represented. We used the subsets of π as the lattice elements and their subset relationship as the partial order; alternatively, one could define a lattice for predicate abstraction using conjunctions over predicates from π as the lattice elements and their formula-implication relationship as partial order. The concretization function $[\![\cdot]\!]$ is defined by $[\![r]\!] = \{c \in C \mid c \models \varphi_r\}$.
2. The transfer relation $\rightsquigarrow_{\mathbb{P}}$ has the transfer $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} r'$ if $\mathsf{post}(\varphi_r, g)$ is satisfiable and r' is the largest set of predicates from π such that φ_r implies $\mathsf{pre}(p, g)$ for each $p \in r'$, where $\mathsf{post}(\varphi, g)$ and $\mathsf{pre}(\varphi, g)$ denote the strongest post-condition and the weakest pre-condition, respectively, for a formula φ and a control-flow edge g. The two operators post and pre are defined such that $[\![\mathsf{post}(\varphi, g)]\!] = \{c' \in C \mid \exists c \in C : c \overset{g}{\rightarrow} c' \wedge c \models \varphi\}$ and $[\![\mathsf{pre}(\varphi, g)]\!] = \{c \in C \mid \exists c' \in C : c \overset{g}{\rightarrow} c' \wedge c' \models \varphi\}$. The Cartesian abstraction of the successor state is obtained by separate entailment checks for each predicate in π, which can be implemented by $|\pi|$ calls of a theorem prover.[11]

[11] A more efficient formulation of the same problem is based on the weakest pre-condition in order to avoid existential quantification.

3. The merge operator does not combine elements when the control flow meets: $\mathsf{merge}_{\mathbb{P}} = \mathsf{merge}^{sep}$.
4. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{P}} = \mathsf{stop}^{sep}$.

Note that the CPA $\mathbb{P}$ cannot run alone: it is a component analysis that works in a composite analysis with the location analysis from Sect. 4.1 as another component CPA. The analysis could in principle be designed such that the predicates in π also constrain the program location, but this is not considered here.

The first practical implementations of a program analysis with Cartesian predicate abstraction were developed more than ten years ago (cf. SLAM [3, 4] and BLAST [13, 71]). More recent advancements in predicate analysis use Boolean abstraction [9] instead of Cartesian abstraction, and a complete temporal separation of the computation of the predicate abstraction for a formula from the computation of the strongest post-condition for a program operation [22]. An overview of Cartesian predicate abstraction is also given in Chap. 15, and of SAT solving in Chap. 9.

4.5 Explicit-Heap Analysis

In the following, we outline a simple analysis of dynamic data structures on the heap (as an extension of the simple programming language that we used so far), which is, for example, used as a basis for an accelerated abstraction in shape analysis [18, 19]; we give only a coarse overview here.

The *explicit-heap analysis* is a configurable program analysis $\mathbb{H} = (D_{\mathbb{H}}, \rightsquigarrow_{\mathbb{H}}, \mathsf{merge}_{\mathbb{H}}, \mathsf{stop}_{\mathbb{H}})$, which tracks explicit heap structures up to a certain size and consists of the following components:

1. The domain of the explicit-heap analysis stores concrete instances of data structures in its abstract states. Each abstract state represents an explicit, finite part of the memory. An *abstract state* $H = (v, h)$ of an explicit-heap analysis consists of the following two components: (1) the variable assignment $v : X \to \mathbb{Z}_{\top}$ is a total function that maps each variable identifier (integer or pointer variable) to an integer (representing an integer value or a structure address) or the special value $\top$ (representing the value "unknown"); and (2) the heap assignment $h : \mathbb{Z} \rightharpoonup (F \to \mathbb{Z}_{\top})$ is a partial function that maps every valid structure address to a field assignment, also called a *structure cell* (memory content). A field assignment is a total function that maps each field identifier $f \in F$ of the structure to an integer, or the special value $\top$. We call H an *explicit heap*. The initial explicit heap $H_0 = (v_0, \{\})$, with $v_0(x) = \top$ for every program variable x, represents all program states. Given an explicit heap H and a structure address a, the *depth* of H from a, denoted by $depth(H, a)$, is defined as the maximum length of an acyclic path whose nodes are addresses and where an edge from a_1 to a_2 exists if $h(a_1)(f) = a_2$ for some field f, starting from $v(a)$. The *depth* of H, denoted by $depth(H)$, is defined as $\max_{a \in X} depth(H, a)$.

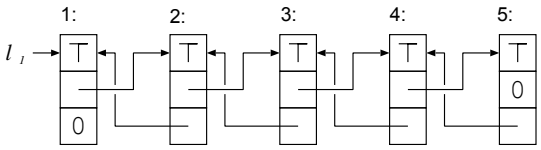


Fig. 3 Sample explicit heap for a doubly-linked list

2. The transfer relation $\rightsquigarrow_{\mathbb{H}}$ has the transfer $H \overset{g}{\rightsquigarrow}_{\mathbb{H}} H'$ if $H' = (v', h')$ is the explicit heap that results from applying the control-flow edge $g = (l, op, l')$ to the explicit heap $H = (v, h)$ according to the semantics of op. The new variable assignment v' maps every pointer variable p to $\top$ for which $depth(H, p) > c$, where c is an analysis-dependent constant maximal depth value of the heap structures.[12] (The analysis stops tracking structures that have a depth greater than the maximal depth value.)
3. The merge operator does not combine elements when the control flow meets: $\mathsf{merge}_{\mathbb{H}} = \mathsf{merge}^{sep}$.
4. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{H}} = \mathsf{stop}^{sep}$.

Besides explicit-heap analysis, which can only serve as an auxiliary analysis or for bounded bug finding, several approaches for symbolic-heap analysis were proposed in the literature [16, 19, 32, 45, 75, 103].

Example 9. Figure 3 graphically depicts an explicit heap (v, h) that can occur in a program operating on a structure `elem` $\{$`int data; elem* succ; elem* prev`$\}$, with $v = \{l_1 \mapsto 1\}$ and $h = \{$
$1 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 2, \texttt{prev} \mapsto 0\}$,
$2 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 3, \texttt{prev} \mapsto 1\}$,
$3 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 4, \texttt{prev} \mapsto 2\}$,
$4 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 5, \texttt{prev} \mapsto 3\}$,
$5 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 0, \texttt{prev} \mapsto 4\}$
$\}$.

4.6 BDD Analysis

Binary decision diagrams (BDDs) [31] are a popular data structure in model-checking algorithms. In the following, we define a configurable program analysis that uses BDDs to represent abstract states. For the details, we refer to an article on the topic [25]. An introduction to BDDs is given in Chap. 7 and to BDD-based model checking in Chap. 8. Given a first-order formula φ over the set X of program variables, we use $\mathscr{B}_\varphi$ to denote the BDD that is constructed from φ, and $[\![\varphi]\!]$ to denote all variable assignments that fulfill φ. Given a BDD $\mathscr{B}$ over X, we use $[\![\mathscr{B}]\!]$ to denote all variable assignments that $\mathscr{B}$ represents ($[\![\mathscr{B}_\varphi]\!] = [\![\varphi]\!]$).

[12] The analysis has to apply garbage collection in heap assignments of the abstract states.

The *BDD analysis* is a configurable program analysis $\mathbb{BPA} = (D_{\mathbb{BPA}}, \rightsquigarrow_{\mathbb{BPA}}, \mathsf{merge}_{\mathbb{BPA}}, \mathsf{stop}_{\mathbb{BPA}})$ that represents the data states of the program symbolically, by storing the values of variables in BDDs. The CPA consists of the following components (taken from [25]):

1. The abstract domain $D_{\mathbb{BPA}} = (C, \mathscr{E}_{\widehat{\mathscr{B}}}, [\![\cdot]\!])$ is based on the semi-lattice $\mathscr{E}_{\widehat{\mathscr{B}}}$ of BDDs, i.e., every abstract state consists of a BDD. The concretization function $[\![\cdot]\!]$ assigns to an abstract state $\mathscr{B}$ the set $[\![\mathscr{B}]\!]$ of all concrete states that are represented by the BDD. Formally, the semi-lattice $\mathscr{E}_{\widehat{\mathscr{B}}} = (\widehat{\mathscr{B}}, \sqsubseteq, \sqcup, \top)$ — where $\widehat{\mathscr{B}}$ is the set of all BDDs and $\top = \mathscr{B}_{true}$ is the BDD that represents all concrete states (1-terminal node) — is induced by the partial order $\sqsubseteq$ that is defined as: $\mathscr{B} \sqsubseteq \mathscr{B}'$ if $[\![\mathscr{B}]\!] \subseteq [\![\mathscr{B}']\!]$. The join operator $\sqcup$ for two BDDs $\mathscr{B}$ and $\mathscr{B}'$ yields the least upper bound $\mathscr{B} \vee \mathscr{B}'$.
2. The transfer relation $\rightsquigarrow_{\mathbb{BPA}}$ has the transfer $\mathscr{B} \overset{g}{\rightsquigarrow} \mathscr{B}'$ with $\mathscr{B}' = \begin{cases} \mathscr{B} \wedge \mathscr{B}_p & \text{if } g = (l, \texttt{assume}(p), l') \text{ and } [\![\mathscr{B} \wedge \mathscr{B}_p]\!] \neq \{\} \\ (\exists w : \mathscr{B}) \wedge \mathscr{B}_{w=e} & \text{if } g = (l, \texttt{w} := \texttt{e}, l'). \end{cases}$
3. The merge operator is defined by $\mathsf{merge}_{\mathbb{BPA}} = \mathsf{merge}^{join}$.
4. The termination check is defined by $\mathsf{stop}_{\mathbb{BPA}} = \mathsf{stop}^{sep}$.

A complete program analysis can be instantiated by composing the CPA $\mathbb{BPA}$ for BDD-based analysis with the CPA $\mathbb{L}$ for location analysis, in order to also track the program locations.

4.7 Observer Automata

Many software verifiers require the user to encode the safety property to be verified as a reachability problem inside the program source code. It has been shown that tools can provide more convenient and elegant specification languages for expressing safety properties separately from the program [5,8]. This approach has the advantages that the property need not be present in the program source code, and that different properties can be checked independently (and possibly simultaneously). The software model checker BLAST [8] provides a transformation technique that takes as input the original program and the specification, and produces an instrumented program. The instrumented program is then given to the standard model checker, which simply checks for reachability of an error label.

This approach can be realized even more elegantly using a composite analysis that transforms the specification into an observer automaton that runs in parallel with the other analyses of the verifier in a composition. Such a strategy was implemented, for example, in the software verifiers BLAST [120], CPACHECKER [21], and ORION [44].

A *specification* is an abstract description of a set of valid program paths for a given program. We represent such a specification as an *observer automaton* that observes whether an invalid program path is encountered. Observer automata are also called "monitors" in the literature and can be generated from temporal-logic specifications.

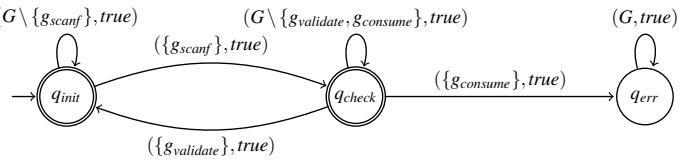


Fig. 4 Simple observer automaton

This idea is also used in test-case generation, where (temporal) coverage criteria are transformed into test-goal automata [20], and for re-playing error witnesses [11, 26].

Example 10. Consider as specification that each user input that the program reads (e.g., via `scanf`) must be validated by a call to a function `validate` before it is consumed (e.g., via function `consume`). The observer automaton in Fig. 4 starts in accepting state q_{init}, and switches to another accepting state q_{check} when `scanf` is called. From there, the automaton switches back to state q_{init} if function `validate` is called, but it switches to a non-accepting sink state if the input value is consumed without validation.

An *observer automaton* $A = (Q, \Sigma, \delta, q_{init}, F)$ for a CFA (L, l_0, G) is a non-deterministic finite automaton, with the finite set Q of control states, the alphabet $\Sigma \subseteq 2^G \times \Phi$ consisting of pairs that consist of a finite set of CFA edges and a state condition, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, the initial control state $q_{init} \in Q$, and the set F of final control states (usually, all control states except the error control state $q_{err} \in Q$ are accepting control states, i.e., $F = Q \setminus \{q_{err}\}$). Let $p \in Q$ be the current state of an automaton A. The meaning of a transition $(p, (D, \psi), q) \in \delta$ is as follows: for a given control-flow edge g of the program analysis, the successor control state is control state q if the edge g matches one of the edges in the set D of edges. In combination with another CPA, using a strengthening operator, the successor state can be required to fulfill condition ψ (a later section will describe this). The observer automaton A accepts all program paths that have not reached the error control state, and rejects all program paths that reach the error control state. The specification that the observer automaton represents is fulfilled if all program paths are accepted by the observer automaton.

The *observer analysis* for an observer automaton A is a configurable program analysis $\mathbb{O} = (D_{\mathbb{O}}, \rightsquigarrow_{\mathbb{O}}, \mathsf{merge}_{\mathbb{O}}, \mathsf{stop}_{\mathbb{O}})$, that tracks the control state of the observer automaton $A = (Q, \Sigma, \delta, q_{init}, F)$, with $\Sigma \subseteq 2^G \times \Phi$, and consists of the following components (for a given CFA (L, l_0, G)):

1. The abstract domain $D_{\mathbb{O}} = (C, \mathscr{Q}, [\![\cdot]\!])$ consists of the set C of concrete states, the semi-lattice $\mathscr{Q}$, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathscr{Q} = (Z, \sqsubseteq, \sqcup, \top_{\mathscr{Q}})$, with $Z = (Q \cup \{\top\}) \times \Phi$, consists of the set Z of abstract data states, which are pairs of a control state from Q (or special lattice element) and a condition from Φ, a partial order $\sqsubseteq$, the join operator $\sqcup$, and the top element $\top_{\mathscr{Q}}$. The partial order $\sqsubseteq$ is defined such that $(q, \psi) \sqsubseteq (q', \psi')$ if $(q' = \top$ or $q = q')$ and $\psi \Rightarrow \psi'$, the join $\sqcup$ is

the least upper bound of two abstract data states, and the top element $\top_{\mathcal{Q}} = (\top, \mathit{true})$ is the least upper bound of the set of all abstract data states. The concretization function $[\![\cdot]\!] : Z \to 2^C$ is a mapping that assigns to each abstract data state (q, ψ) the set $[\![\psi]\!]$ of concrete states.

2. The transfer relation $\rightsquigarrow_{\mathbb{O}}$ has the transfer $(q, \psi) \overset{g}{\rightsquigarrow}_{\mathbb{O}} (q', \psi')$ if the observer automaton A has a transition $(q, (D, \psi'), q') \in \delta$ such that $g \in D$. The condition ψ' of the state transition is stored in the successor in order to enable a composite strengthening operator to strengthen the successor abstract data state of another component analysis in the composite analysis using information from condition ψ'.

3. The merge operator combines elements with the same control state:

$$\mathsf{merge}_{\mathbb{O}}((q, \psi), (q', \psi')) = \begin{cases} (q', \psi \vee \psi') & \text{if } q = q' \\ (q', \psi') & \text{otherwise.} \end{cases}$$

4. The termination check considers control states and conditions of the automaton individually: $\mathsf{stop}_{\mathbb{O}} = \mathsf{stop}^{sep}$.

5 Combination Examples (Composite Analyses)

We describe several examples in which component analyses are assembled into composite analyses that are neither pure data-flow analyses nor pure model checking. We show that such combinations are relatively easy to express in the CPA formalism, and explain what is taken from which approach and why it is useful to combine them.

5.1 Predicate Analysis + Constant Propagation

"Predicated lattices" are a practical combination of the predicate-abstraction domain with a classic data-flow domain [49]. The predicate analysis behaves as in model checking: abstract predicate states are never joined. The composite analysis performs a merge of two composite abstract states as follows: if the two component abstract states of the predicate analysis are equal, then the two component abstract states of the data-flow analysis are joined and the composite analysis stores one composite abstract state, otherwise the composite analysis stores two separate composite abstract states.

Given the CPA $\mathbb{P}$ for predicate analysis and any CPA for data-flow analysis, for example, the CPA $\mathbb{CO}$ for constant propagation. The composite program analysis $\mathscr{C}_{\mathbb{PCO}} = (\mathbb{L}, \mathbb{P}, \mathbb{CO}, \rightsquigarrow_{\times}, \mathsf{merge}_{\times}, \mathsf{stop}_{\times})$ for a predicated constant propagation consists of the following components: the CPA $\mathbb{L}$ for location tracking from Sect. 4.1, the CPA $\mathbb{P}$ for predicate analysis from Sect. 4.4, the CPA $\mathbb{CO}$ for constant propagation from Sect. 4.2, the composite transfer relation $\rightsquigarrow_{\times}$, the composite

merge operator $\mathsf{merge}_\times$, and the composite termination check $\mathsf{stop}_\times$. The composite transfer relation $\rightsquigarrow_\times$ has the transfer $(e_1,e_2,e_3) \overset{g}{\rightsquigarrow}_\times (e_1',e_2',e_3')$ if $e_1 \overset{g}{\rightsquigarrow}_{\mathbb{L}} e_1'$ and $e_2 \overset{g}{\rightsquigarrow}_{\mathbb{P}} e_2'$ and $e_3 \overset{g}{\rightsquigarrow}_{\mathbb{CO}} e_3'$. The composite merge operator $\mathsf{merge}_\times$ is defined by

$$\mathsf{merge}_\times((e_1,e_2,e_3),(e_1',e_2',e_3')) = \begin{cases} (e_1,e_2,\mathsf{merge}_{\mathbb{CO}}(e_3,e_3')) & \text{if } e_1 = e_1' \text{ and } e_2 = e_2' \\ (e_1',e_2',e_3') & \text{otherwise.} \end{cases}$$

The composite termination check is defined by $\mathsf{stop}_\times = \mathsf{stop}^{sep}$.

For the combination of predicate analysis with a data-flow analysis for pointers, it has been shown that this configuration can significantly improve the verification performance [49].

5.2 Predicate Analysis + Constant Propagation + Strengthen

Now we extend the above composite program analysis by using a strengthening operator in the transfer relation. Again, a strengthening operator $\downarrow : E_1 \times E_2 \to E_1$ takes an abstract state $e_1 \in E_1$ as input and uses information stored in an abstract state $e_2 \in E_2$ from another CPA to constrain ("strengthen") the set of concrete states that the resulting abstract state $\downarrow(e_1,e_2)$ represents.

We use a strengthening operator of the concrete type $\downarrow_{\mathbb{CO},\mathbb{P}} : E_{\mathbb{CO}} \times E_{\mathbb{P}} \to E_{\mathbb{CO}}$, i.e., it strengthens a variable assignment from the constant propagation with an abstract state from the predicate analysis (set of predicates that are satisfied). The strengthening operator $\downarrow_{\mathbb{CO},\mathbb{P}}(v,r)$ is defined, if $\phi_v \wedge \phi_r$ is satisfiable, as follows, for every variable x:

$$\downarrow_{\mathbb{CO},\mathbb{P}}(v,r)(x) = \begin{cases} c & \text{if } c \text{ is the only satisfying assignment of } \phi_v \wedge \phi_r \text{ for } x \\ v(x) & \text{otherwise} \end{cases}$$

where $\phi_v := \bigwedge_{x \in X, v(x) \neq \top_{\mathscr{Z}}} x = v(x)$.

We now define the transfer relation for the new composite program analysis: The composite transfer relation $\rightsquigarrow_\times$ has the transfer $(e_1,e_2,e_3) \overset{g}{\rightsquigarrow}_\times (e_1',e_2',e_3')$ if $e_1 \overset{g}{\rightsquigarrow}_{\mathbb{L}} e_1'$, and $e_2 \overset{g}{\rightsquigarrow}_{\mathbb{P}} e_2'$, and $e_3 \overset{g}{\rightsquigarrow}_{\mathbb{CO}} e_3''$, and $\downarrow(e_3'',e_2')$ is defined, and $e_3' = \downarrow(e_3'',e_2')$.

This combined analysis is more precise than the component analyses alone, which will be illustrated in the following example. A more flexible extension of this combination was presented using the concept of *dynamic precision adjustment* [18]. Experiments with this extension have shown that combinations with strengthening operators can be more effective and more efficient than Cartesian products of analyses. While the effects of such "reduced products" [39] have been known for decades, the framework of configurable program analysis enables us to express such combinations in a simple and elegant implementation.

Example 11. Consider the example program in Fig. 5, which extends the previous example with a non-linear expression. The safety property to be checked is that no division by zero is executed. Suppose we use a predicate analysis for the theory of linear arithmetics (LA) and equalities with uninterpreted functions (EUF), with the precision (i.e., set of predicates to track) $\{\mathtt{x} = 1, \mathtt{z} = 1, \mathtt{z} = 5, \mathtt{y} \geq 1, \mathtt{y} \leq 1\}$ and a constant-propagation analysis.

```
1  int foo(int y) {
2    int x = 1;
3    int z = 1;
4    if (y < 1) {
5      return 0;
6    }
7    if (y > 1) {
8      x = 5;
9    } else {
10     z = 5 * x * y;
11   }
12   return 10 / (x - z);
13 }
```

Fig. 5 Example C function, used to illustrate predicated lattices with strengthening. Neither predicate analysis, nor constant propagation, nor a predicated lattice without strengthening are precise enough to prove the correctness of this function, but the combination with strengthening is

The predicate analysis with location tracking does not succeed in proving this example safe: At program location 7, we have the predicate abstract data state $\mathtt{x} = 1 \ \wedge \ \mathtt{z} = 1 \ \wedge \ \mathtt{y} \geq 1$. At program location 10, we have the abstract data state $\mathtt{x} = 1 \ \wedge \ \mathtt{z} = 1 \ \wedge \ \mathtt{y} \geq 1 \ \wedge \ \mathtt{y} \leq 1$. At program location 11, however, we have no information about z, due to the fact that the non-linear operation "*" is modeled as an uninterpreted function, resulting in the following abstract data state: $\mathtt{x} = 1 \ \wedge \ \mathtt{y} \geq 1 \ \wedge \ \mathtt{y} \leq 1$. Thus, the analysis conservatively assumes that the program can fail with a division by zero. This cannot be remedied by adding other predicates.

The constant propagation stores the value $\top$ for program variable y after the assume operations from the `if` statements in lines 4 and 7, and thus, also cannot determine the value of z before the division is computed, and conservatively reports that the division might fail.

Also the "predicated lattice" (without strengthening) from Sect. 5.1 is not precise enough to prove that a division by zero cannot occur. Although the analysis is now path-sensitive with respect to the predicates, the constant propagation (which can precisely interpret the multiplication) cannot determine the value of program variable y, and the predicate analysis cannot determine the result of the multiplication, regardless of the predicates used.

The composite analysis with strengthening can transfer information from the predicate abstract data states to the variable assignments of the constant-propagation analysis. At program location 10, we have the abstract data state $\mathtt{x} = 1 \ \wedge \ \mathtt{z} = 1 \ \wedge \ \mathtt{y} \geq 1 \ \wedge \ \mathtt{y} \leq 1$ for the predicate analysis and $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 1\}$ for the constant-propagation analysis. Now, the composite analysis does not store the direct product of these two abstract data states as a composite abstract state, but first strengthens the variable assignment, in particular, of variable y: the only value for y that satisfies the predicate abstract data state is 1, and therefore, the new variable assignment after strengthening is $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 1\}$. The constant-propagation analysis can now compute a value not equal to $\top$ for

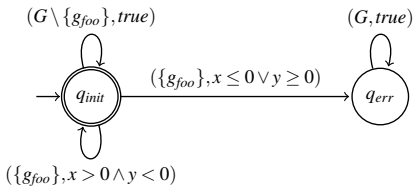


Fig. 6 Example observer automaton with conditions

program variable z for the assignment from location 10 to 11, and thus, is able to prove the program correct, i.e., that there is no division by zero.

5.3 Predicate Analysis + Explicit-Heap Analysis

Similarly to the analysis that incorporates the results of an inexpensive constant-propagation analysis into a predicate analysis, we can enhance the analysis by using the result of the explicit-heap analysis. Of course, in order to keep the analysis practically relevant, the threshold for the heap analysis should be small. This way, information about the heap that is normally not tracked by the predicate analysis can be fed to the predicate analysis via a composite strengthening operator, in order to make the path decisions of the predicate analysis more precise. A combination of an abstract domain with an explicit-heap analysis was used already in a different context [19], where a symbolic abstract shape representation was extracted from the explicit-heap results. The combination was shown to be able to verify more programs than the component analyses alone. This direction of combinations of program analyses is largely unexplored in the literature.

5.4 Predicate Analysis + Observer Automata

The following example observer automaton contains conditions at the transitions, but the observer analysis from Sect. 4.7 is not able to respect the conditions. After the motivating example, we introduce a combination analysis that is able to consider the conditions during the state-space exploration.

Example 12. Let us consider a specification that requires the pre-condition $x > 0 \wedge y < 0$ to be fulfilled whenever function `foo` is called. Figure 6 shows an observer automaton for this specification, with initial state q_{init} and error state q_{err}. The observer automaton starts in control state q_{init}. As long as the exploration of the program encounters only control-flow edges different from g_{foo}, the automaton stays in control state q_{init}. Once a control-flow edge g_{foo} is taken in the exploration, our observer automaton has to consider the conditions at the transitions for g_{foo}: if the

condition $\psi = x > 0 \wedge y < 0$ is fulfilled (specification satisfied) then the automaton stays in (accepting) control state q_{init}, otherwise the observer automaton switches to (non-accepting) control state q_{err}. The error state q_{err} is a sink state and thus the explored program path will be rejected (because it violates the specification).

We construct a composite program analysis that runs both the predicate analysis and the observer analysis as components. The resulting program analysis combines the abstract data states in such a way that (1) it uses information from the predicate analysis to determine the actual transition switch of the observer automaton, and (2) it marks all paths through the program that violate the specification with the control state q_{err}:

1. The composite domain $D_\times = D_\mathbb{P} \times D_\mathbb{O}$ is the product of the component domains $D_\mathbb{P}$ for the predicate analysis and $D_\mathbb{O}$ for the observer analysis.

2. The transfer relation $\rightsquigarrow_\times$ has the transfer $(\varphi,(q,\psi)) \overset{g}{\rightsquigarrow}_\times (\varphi',(q',\psi'))$ if $\varphi \overset{g}{\rightsquigarrow}_\mathbb{P} \varphi'$, and $(q,\psi) \overset{g}{\rightsquigarrow}_\mathbb{O} (q',\psi')$, and $\downarrow_{\mathbb{P},\mathbb{O}}(\varphi',(q',\psi'))$ is defined. The strengthening operator $\downarrow_{\mathbb{P},\mathbb{O}}$ is defined only if $\varphi' \wedge \psi'$ is satisfiable, in which case it returns $\varphi' \wedge \psi'$ as the abstract data state of the predicate analysis. In other words, the strengthening operator (a) eliminates successors of the observer automaton with conditions that contradict the abstract state of the predicate analysis and (b) restricts the abstract state of the predicate analysis to those concrete states that satisfy the condition of the observer automaton. The strengthening operator is necessary because both (a) and (b) can be evaluated only after the successors of all participating CPAs are known.

3. The merge operator keeps different abstract states separate: $\mathsf{merge}_\times = \mathsf{merge}^{sep}$.

4. The termination check considers abstract states individually: $\mathsf{stop}_\times = \mathsf{stop}^{sep}$.

Note on Soundness. The strengthening operator might replace the original abstract state by a new abstract state that represents fewer concrete states. To guarantee soundness of the composition program analysis, the observer automaton must not restrict the program exploration of other analyses, i.e., for all control states $q \in Q$, the disjunction of the conditions ψ_i' of all transitions $(q,(D_i,\psi_i'),q_i)$ that leave control state q must be equivalent to *true*.[13]

There are applications for which the soundness requirement is not desirable and the automata are used to control (restrict) the program exploration of the other analyses, for example, as used in test-goal automata [20] and error-witness automata [11, 26].

[13] This soundness requirement is easy to fulfill on the syntactical level by using a SPLIT operation in the definition of transitions of the automaton. The transition syntax SPLIT($x > 0 \wedge y < 0$, q_{init}, q_{err}), for example, defines the two transitions from control state q_{init} to q_{init} and to q_{err} (cf. Fig. 6).

6 Algorithms for Constructing Program Invariants

While the previous section focuses on abstract domains and how to practically combine abstract domains from data-flow analysis with abstract domains from model checking in a unifying, configurable framework, this section discusses different algorithmic styles that are used to compute program invariants in data-flow analysis and model checking.

The general idea is to construct a witness that proves the correctness or incorrectness of the program. To show that a program violates a property, an *error path* (*counterexample* [34], exchangeable *error witness* [11]) is constructed. To show that a program satisfies a property, a *program invariant* (main ingredient of a *correctness proof*) is constructed; program invariants can be stored as *certificates* [70] or exchangeable *correctness witnesses* [10]. The program invariants look different depending on the algorithm that is used to construct it.

Both data-flow analysis and model checking construct over-approximations of the reachable concrete states (Algorithm 1). For data-flow analysis, the program invariant is a function reached, which assigns to each reachable program location an over-approximation of all concrete data states that can occur at that location. For model checking, the program invariant is a set reached, which contains a set of abstract states that contain all concrete states of the program (possibly many abstract states for the same program location, depending on how path-sensitive the analysis is).

If the program invariant is computed via a fixed-point iteration, then the program invariant is called the *solution for the fixed-point problem*. In the following, we describe different algorithmic approaches to compute program invariants (fixed points).

6.1 Iterative and Monotonic Fixed-Point Approaches

The most commonly known and used algorithm for computing a program invariant consists of an iteration that initializes the unknown program invariants to a lower bound (or upper bound) of the abstract values and then updates them monotonically to compute a least (or greatest) fixed point over the underlying abstract domain or invariant language (cf. Sects. 2.2 and 3.2). This technique is used in various standard approaches to data-flow analysis and model checking. One notable dimension for analyses in this category is whether the analysis is forward or backward.

Forward analyses start from pre-conditions and propagate them forward (iteratively across loops until a fixed point is reached) to compute invariants at various program locations. Forward analyses have the advantage of not requiring the code to be annotated with post-conditions and hence can generate invariants not only for program verification but also for applications such as compiler optimization. The key challenge in designing a forward analysis is to design abstract transfer relations and merge operators (including *join* and *widen*) that can compute over-approximations of strongest post-conditions (cf. Sect. 3.5). Such transfer relations are known for a variety of abstract domains, including linear arithmetic [41, 92],

uninterpreted functions [57], combination of linear arithmetic and uninterpreted functions [61], heap-shape domains [16, 45, 103], combination of arithmetic and heap-shape domains [55], and quantified array properties [56].

Backward analyses typically require post-conditions to start with, but have the advantage of being goal-directed. The key challenge in backward analysis is to have an *abduct* procedure for computing under-approximations of weakest pre-conditions (as opposed to forward abstract transfer relations, which perform over-approximations of strongest post-conditions). Such procedures are known for some abstract domains including linear arithmetic [62], uninterpreted functions [62], combinations of linear arithmetic and uninterpreted functions [60], and heap shapes [32]. Backward analyses have received less attention in terms of research projects compared to forward analyses, but become more important in the context of combination of forward and backward reachability analysis [119].

Another notable dimension for analyses in this category is the order of iteration (cf. Sect. 3.5). In Algorithm 2, the iteration order is encoded in the operator choose, which selects the next abstract state to explore from the set waitlist of abstract states that are still to be processed. Besides the two simple graph-traversal orders depth-first search (DFS) and breadth-first search (BFS), there are many possible implementations of the choose operator, such as random order, chaotic order [29], post-order, reverse post-order [22], topological order, and many more (e.g., [88]).

6.2 Counterexample-Guided Abstraction Refinement

One of the challenges in data-flow analysis and model checking is to automatically construct an abstract model of a program, or more precisely, the level of abstraction for a given abstract domain. Many classic analyses hard-wire the abstraction level into the abstract domain, but the *precision* of the analysis can also be treated as a separate concern [18]. For example, if the abstract domain is taken from constant propagation, then the precision can be a set of variables and determine which variables are tracked; if the abstract domain is predicate abstraction, the precision is the set of predicates that are tracked.

The problem of computing an appropriate precision can be solved by counterexample-guided abstraction refinement (CEGAR) [34]. This technique works orthogonally to the above-mentioned iteration techniques. The analysis approach (e.g., iterative) starts with a coarse precision (very abstract model), and successively refines the abstract model by adding information to the precision. If the analysis finds a violation of the property to be verified, then the abstract error path is analyzed. If the abstract error path represents a concrete error path (executable violation), then the analysis can stop and report the violation. If the abstract error path does not represent a concrete error path (infeasible path) then that path was found due to a too-coarse (too imprecise) abstract model, and the abstract error path can be used to find out what information is necessary to track in the abstract model in order to eliminate this

abstract error path from further explorations. The extracted information is added to the precision, the set of reached abstract states is updated, and the analysis continues.

While CEGAR is most popular for predicate analysis, the technique has also been explored for value analysis [24, 96] and symbolic execution [23]. There are several techniques to extract information from counterexamples, for example, extraction from syntax and weakest pre-conditions using a set of heuristics [4, 13, 21, 35], using Craig interpolation [13, 21, 42, 69], and using invariant synthesis [15]. More details are provided in Chap. 14 on interpolation and in Chap. 13 on CEGAR.

6.3 Template- and Constraint-Based Approaches

Constraint-based approaches construct the program invariant by guessing a second-order template for each necessary loop invariant such that the only unknowns in the second-order template are first-order quantities. Then, the approach generates constraints over those first-order unknowns (after substituting the guessed form into the program invariant). The generated constraints are existentially quantified in the first-order unknowns, but universally quantified in the program variables. The challenge of solving these constraints is to have a procedure to eliminate the universally quantified variables from the constraints, and then solve the constraints for the existentially quantified variables by using some off-the-shelf constraint solver.

Consider the following example program:

```
1 if (n <= 0) { return 1;}
2 x = 0; y = 1;
3 while (x != n) { x = x + 1; y = y + 2; }
4 assert(y == 2n + 1);
```

Suppose we guess that the loop invariant that is required to prove the assertion is of the form $a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0$, where a, b, c, and d are unknown integer constants. Substituting this loop-invariant template into the program invariant for the above program yields the following constraints for the loop head, where all program variables `x`, `y`, and `n` are universally quantified:

$$\begin{aligned}
\mathtt{n} > 0 \ \wedge\ \mathtt{x} = 0 \ \wedge\ \mathtt{y} = 1 \ &\Rightarrow\ a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0 \\
a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0 \ \wedge\ \mathtt{x} = \mathtt{n} \ &\Rightarrow\ \mathtt{y} = 2\mathtt{n} + 1 \\
a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0 \ \wedge\ \mathtt{x} \neq \mathtt{n} \ &\Rightarrow\ (a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0)_{[\mathtt{x} \mapsto (\mathtt{x}+1), \mathtt{y} \mapsto (\mathtt{y}+2)]}.
\end{aligned}$$

Farkas' lemma can be used to eliminate universally quantified variables from the above constraint, thereby obtaining the following constraint:

$$c = 0 \ \wedge\ b + d = 0 \ \wedge\ 2b + a + c = 0 \ \wedge\ b \neq 0.$$

An off-the-shelf first-order constraint solver may now generate the solution $a = -2, b = 1, d = -1$, thereby yielding the invariant $y = 2x + 1$.

This kind of invariant-computation technique has been developed for a variety of abstract domains including linear inequalities [104], disjunctions of linear inequalities [58], non-linear inequalities [63, 105], combination of linear inequality and uninterpreted functions [14], predicate abstraction [15, 59], and quantified invariants [111]. The key component in the algorithms for these domains is often a novel procedure to eliminate universal quantification.

6.4 Proof-Rule-Based Approaches

Approaches that are based on proof rules require the analysis designer to have a good understanding of the design patterns (for loop behaviors) that occur in practice, and then to develop proof rules for each of these design patterns. The beauty of this approach is that it usually enables the analysis of program loops by simply reasoning about their (loop-free) bodies in order to identify the appropriate design pattern and apply the corresponding rule. The reasoning about loop-free code fragments can be done using off-the-shelf SMT solvers. This approach has been applied for a variety of program analyses, including symbolic computational-complexity analysis [64], continuity analysis [33], and variable-bound analysis [54].

Example 13. If the transition system of a loop implies that $\mathtt{x}' = \mathtt{x} << 1 \ \wedge \ \mathtt{x} \neq 0$ or $\mathtt{x}' = \mathtt{x}\&(\mathtt{x}-1) \ \wedge \ \mathtt{x} \neq 0$, then $LSB(\mathtt{x})$ is a ranking function for that loop (where $\mathtt{x}$ is any loop variable, $\mathtt{x}'$ denotes the update to that loop variable, and $LSB(\mathtt{x})$ returns the least significant bit of $\mathtt{x}$, and $<<$ and & represent bitwise-left-shift and bitwise-and operators respectively) [64]. As another example, if the transition system of a loop is of the form $s_1 \vee s_2$, and r_1 and r_2 are ranking functions for s_1 and s_2, respectively, and s_1 (resp., s_2) implies that r_2 (resp., r_1) is non-increasing, then the number of iterations of the loop above is bounded by $\mathtt{Max}(0, r_1) + \mathtt{Max}(0, r_2)$ [64]. Note that these judgments about loop properties require discharging standard SMT queries that are constructed using transition systems that represent loop-free code fragments.

6.5 Iterative, but Non-monotonic Approaches

There are techniques for computing fixed points that are iterative and converging, but have non-monotonic progress towards a fixed-point solution [50]. In each of the two techniques that we explain below, the non-monotonic iteration has the unifying property that the distance between the iterated abstract states and a fixed-point solution (according to some underlying distance measure) decreases in each iteration (as in the case of Newton's method for computing the roots of an equation).

Probabilistic Inference. Inspired by techniques from machine learning, we can pose the problem of computing a program invariant as an inference in probabilistic graph

models, which allows the use of probabilistic inference techniques like belief propagation, in order to perform the fixed-point computation. This technique requires us to develop appropriate distance measures between any two abstract states of an abstract domain (which traditionally is equipped with only a partial order) to guide the progress of the probabilistic inference algorithm. This technique has been applied to discovering disjunctive quantifier-free invariants on numerical programs [53]. The algorithm iteratively selects a program location randomly and updates the current abstract state to make it more *locally consistent* with respect to the abstractions at the neighboring program locations (as per the underlying distance measure, until convergence). Interestingly, this simple algorithm was shown to converge in a few rounds for the chosen benchmark examples, yielding the desired invariants. The distance measure, for a pair of abstract states e_1 and e_2, was chosen to be proportional to the number of pairs (i, j) such that e_1^i does not imply e_2^j, where $\bigvee_{i=1}^{n} e_1^i$ is the disjunctive normal form representation of e_1 and $\bigwedge_{j=1}^{m} e_2^j$ is the conjunctive normal form representation of e_2. (Note that if e_1 implies e_2 then each e_1^i implies each e_2^j.) Observe that the local inconsistency of the abstract state at a program location is thus a monotonic measure of the set of abstract states that are not consistent with the abstract states at the neighboring program locations.

Learning. Inspired by techniques from concept learning, we can pose the problem of computing a program invariant as an instance of algorithmic learning that requires an oracle to answer simpler questions about the invariant, such as whether a given invariant is the desired one (equivalence question), or whether a given state is a model of the desired invariant (membership question). This technique has been applied to discover quantifier-free invariants [76] as well as quantified invariants [83].

6.6 Comparison with Standard Recurrence Solving

The three techniques "iterative monotonic," "template-based," and "proof-rule-based" for computing program invariants bear striking similarity to the three standard techniques that have long been known in the area of algorithms for generating closed form upper/lower approximations for recurrences [37]. A *recurrence* is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are useful to describe the run time of recursive algorithms.

Substitution Method. The substitution method guesses the template of the solution and then generates constraints (over the first-order unknowns in the guessed template) after substituting the guessed template into the recurrence relation. Any solution to the generated constraints is a valid solution to the recurrence relation. This method is powerful, but requires a template of the answer to be guessed, which takes experience and sometimes requires creativity.

Example 14. Consider the problem of computing a closed form solution for the recurrence $T(n) = T(n-1) + 6n$ with the boundary condition $T(1) = 2$. Suppose we guess that the solution is of the form $T(n) = an^3 + bn^2 + cn + d$, for some (unknown) constants a, b, c, and d. Substituting the guessed form into the recurrence relation and the boundary conditions yields the following constraints:

$$\begin{aligned} an^3 + bn^2 + cn + d &\equiv a(n-1)^3 + b(n-1)^2 + c(n-1) + d + 6n, \\ a + b + c + d &\equiv 2. \end{aligned}$$

These constraints imply $a = 0$, $c = b = 3$, and $d = -4$. This yields the solution $T(n) = 3n^2 + 3n - 4$.

Iteration Method. The idea of the iteration method is to expand (iterate) the recurrence and express it as a sum of terms that are dependent only on n and the initial conditions. Techniques for evaluating sums can then be used to provide bounds on the solution.

Example 15. Consider the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + n$. We iterate it and then express it as a summation as follows (using the observation that the sub-problem size hits $n = 1$ when i exceeds $\log_4 n$):

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T\lfloor n/64 \rfloor \\ &\leq n + 3n/4 + 9n/16 + 27n/64 + \ldots + 3^{\log_4 n}\Theta(1) \\ &\leq n\sum_{i=0}^{\infty}(3/4)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) \\ &= \mathrm{O}(n). \end{aligned}$$

Master Method. The master method relies on the following theorem, which provides a case-based method for solving recurrences of the form $T(n) = aT(n/b) + f(n)$.

Theorem 1 (Master Theorem [37]). *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence*

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then, $T(n)$ can be bounded asymptotically as follows:

1. *If $f(n) = \mathrm{O}(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \lg n)$.*

3. *If* $f(n) = \Omega(n^{\log_b a+\varepsilon})$ *for some constant* $\varepsilon > 0$*, and if* $a \cdot f(n/b) \leq c \cdot f(n)$ *for some constant* $c < 1$ *and all sufficiently large n, then* $T(n) = \Theta(f(n))$.

Example 16. (Using the Master Method [37]) To use the master method, we simply need to determine which case (if any) of the master theorem matches the given recurrence relation. For example, for the recurrence $T(n) = 9T(n/3) + n$, we have $a = 9$, $b = 3$, $f(n) = n$, and thus $n^{\log_b a} = \Theta(n^2)$. Since $f(n) = \mathrm{O}(n^{\log_3 9-\varepsilon})$, where $\varepsilon = 1$, we can apply Case 1 of the master theorem to conclude that $T(n) = \Theta(n^2)$.

Connections. The substitution method for solving recurrences is quite similar to the template-based method for program invariant generation. The iteration method for solving recurrences is similar to the iterative monotonic techniques for invariant generation in that both require iteration (or unrolling) of the underlying recursive system of equations to perform an appropriate generalization. The master method for solving recurrences is in the same category as the proof-rule-based method for invariant generation since both involve (manually) establishing non-trivial theorems to allow easy automated reasoning of most instances by simply requiring a matching engine to match the given instance against an existing small collection of general rules. It is heartening to observe that two different communities have ended up discovering similar classes of useful techniques for reasoning about recursive systems!

6.7 Discussion

We now briefly discuss the advantages and disadvantages of the different techniques.

The iterative monotonic techniques have been the most popular choice in the domains of data-flow analysis and model checking, primarily because they are the oldest and most well-understood techniques. These techniques have also been very successful because they generally allow for selecting the right trade-off between precision and scalability.

The CEGAR algorithm is popular mainly in model checking; it is not applicable to path-insensitive data-flow analysis, because if a property violation is found, then an error path needs to be constructed. The technique is orthogonal to the algorithm that constructs the program invariant — it only requires a notion of precision in order to determine and adjust the abstraction level of the analysis.

Template-based techniques are generally the least scalable because they often involve the use of sophisticated constraint solvers; they are not successful in practice if used in isolation because of the scaling problem. However, these techniques are most effective in analyzing sophisticated properties of small programs, and can be practicable if applied to smaller sub-problems in a larger verification setting, such as computing invariants for path programs [15] during the verification of large programs. The techniques also have further enabled synthesis of small programs [74, 112].

Proof-rule-based techniques are the most scalable, and have been applied to the analysis of large programs; they are limited in applicability because they require the

existence and knowledge of a small set of design patterns that occur in the programs to be analyzed. When applicable, proof-rule-based techniques might be the best choice (just as the master method is the most popular choice for analyzing the run time of standard recursive algorithms as found in textbooks).

Probabilistic inference and learning-based techniques are not yet widely used, and it remains to be seen whether they can produce new impactful results in the area of program verification. Their true strength may lie in dealing with noisy or under-specified systems, and especially in the synthesis of systems.

7 Combinations in Tool Implementations

During recent years, combining approaches from data-flow analysis and model checking became state-of-the-art in tool implementations. To witness this development, we give an overview of the techniques and features that modern software verifiers implement. As a reference collection of tools for software verification, we refer to the Competition on Software Verification. In 2014, a total of 15 verifiers participated in the competition (including demo track); detailed results are available in the competition report [6] and on the competition web site.[14]

Table 1 lists the features and technologies that are used in the verification tools. This illustrates that techniques from data-flow analysis and model checking are combined to achieve better results: Counterexample-guided abstraction refinement (CEGAR, cf. Chap. 13, [34]), predicate abstraction (cf. Chap. 15, [52]), bounded model checking (BMC, cf. Chap. 10, [27]), abstract reachability graphs (ARGs, cf. [13]), lazy abstraction (cf. [16, 71]), interpolation for predicate refinement (cf. Chap. 14, [69]), and termination checking via ranking functions (cf. Chap. 15, [98]) are typical examples of techniques contributed by the model-checking community. Value analysis (similar to constant propagation, cf. [24]), interval analysis (cf. [94]), and shape analysis (cf. [32, 45, 75, 103]) are typical examples of abstract domains from the data-flow community.

8 Conclusion

In theory, there is no difference in expressive power between data-flow analysis and model checking. This chapter describes the paradigmatic and practical differences of the two approaches, which are relevant especially for precision and performance characteristics. The unifying formal framework of configurable program analysis makes the differences explicit. This framework enables an easy combination of abstract domains, no matter whether they were invented for data-flow analysis or for model checking. Several examples demonstrate that the combination of abstract domains

[14] http://sv-comp.sosy-lab.org/

Table 1 Techniques that current verification tools implement (adapted from [6])

Verification Tool	Predicate Abstraction	Bounded Model Checking	Explicit-Value Analysis	Interval Analysis	Shape Analysis	ARG-Based Analysis	Lazy Abstraction	Ranking Functions	Interpolation	CEGAR
APROVE [51]								✓		
BLAST [13, 109]	✓					✓	✓		✓	✓
CBMC [85]		✓								
CPALIEN [91]			✓		✓					
CPACHECKER [21, 87]	✓	✓	✓	✓	✓	✓	✓		✓	✓
CSEQ [72, 117]		✓								
ESBMC [90]		✓								
FUNCTION [118]				✓				✓		
FRANKENBIT [66]		✓							✓	
LLBMC [48]		✓								
PREDATOR [46]					✓					
SYMBIOTIC [110]										
T2 [30]	✓			✓		✓	✓	✓	✓	✓
TAN [84]	✓	✓		✓			✓	✓		✓
THREADER [99]	✓					✓			✓	✓
UFO [65]	✓	✓		✓		✓	✓		✓	✓
ULTIMATE [47, 67, 68]	✓						✓	✓	✓	✓

designed for data-flow analysis with abstract domains designed for software model checking improves both effectiveness (precision) and efficiency (performance) of such analyses. The new, configurable combinations make it possible to plug together composite program analyses that are strictly more powerful than the component analyses. The chapter also provides an overview of the different flavors of algorithms for computing the same solution: program invariants. There are several different approaches, originating from different research communities, and combinations have a large potential for further improving the state of the art. Modern tools for software verification — as witnesses of our considerations — almost always combine techniques from data-flow analysis with techniques from model checking.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer (1996)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstractions for model checking C programs. In: T. Margaria, W. Yi (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 2031, pp. 268–283. Springer (2001)
4. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: J. Launchbury, J.C. Mitchell (eds.) Symp. on Principles of Programming Languages (POPL), pp. 1–3. ACM (2002)
5. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, Microsoft Research (2002)
6. Beyer, D.: Status report on software verification (competition summary SV-COMP 2014). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 373–388. Springer (2014)
7. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: A. Finkelstein, J. Estublier, D.S. Rosenblum (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 326–335. IEEE (2004)
8. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST query language for software verification. In: R. Giacobazzi (ed.) Intl. Symp. on Static Analysis (SAS), *Lecture Notes in Computer Science*, vol. 3148, pp. 2–18. Springer (2004)
9. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Formal Methods in Computer Aided Design (FMCAD), pp. 25–32. IEEE (2009)
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Intl. Symp. on Foundations of Software Engineering (FSE). ACM (2016)
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: E.D. Nitto, M. Harman, P. Heymans (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 721–733. ACM (2015)
12. Beyer, D., Fararooy, A.: A simple and effective measure for complex low-level dependencies. In: Intl. Conf. on Program Comprehension (ICPC), pp. 80–83. IEEE (2010)
13. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. Intl. Journal on Software Tools for Technology Transfer **9**(5-6), 505–525 (2007)
14. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: B. Cook, A. Podelski (eds.) Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI), *Lecture Notes in Computer Science*, vol. 4349, pp. 378–394. Springer (2007)
15. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: J. Ferrante, K.S. McKinley (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 300–309. ACM (2007)
16. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: T. Ball, R.B. Jones (eds.) Intl. Conf. on Computer-Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 4144, pp. 532–546. Springer (2006)
17. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: W. Damm, H. Hermanns (eds.) Intl. Conf. on Computer-Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 4590, pp. 504–518. Springer (2007)
18. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 29–38. IEEE (2008)

19. Beyer, D., Henzinger, T.A., Théoduloz, G., Zufferey, D.: Shape refinement through explicit heap analysis. In: D.S. Rosenblum, G. Taentzer (eds.) Intl. Conf. on Fundamental Approaches to Software Engineering (FASE), *Lecture Notes in Computer Science*, vol. 6013, pp. 263–277. Springer (2010)
20. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information reuse for multi-goal reachability analyses. In: M. Felleisen, P. Gardner (eds.) European Symp. on Programming (ESOP), *Lecture Notes in Computer Science*, vol. 7792, pp. 472–491. Springer (2013)
21. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: G. Gopalakrishnan, S. Qadeer (eds.) Intl. Conf. on Computer-Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 6806, pp. 184–190. Springer (2011)
22. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: R. Bloem, N. Sharygina (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 189–197. Formal Methods in Computer Aided Design (FMCAD) (2010)
23. Beyer, D., Lemberger, T.: Symbolic execution with CEGAR. In: Intl. Conf. on Verified Software: Theories, Tools, and Experiments (VSTTE), Lecture Notes in Computer Science. Springer (2016)
24. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: V. Cortellessa, D. Varró (eds.) Intl. Conf. on Fundamental Approaches to Software Engineering (FASE), *Lecture Notes in Computer Science*, vol. 7793, pp. 146–162. Springer (2013)
25. Beyer, D., Stahlbauer, A.: BDD-based software model checking with CPACHECKER. In: A. Kucera, T.A. Henzinger, J. Nesetril, T. Vojnar, D. Antos (eds.) Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS), *Lecture Notes in Computer Science*, vol. 7721, pp. 1–11. Springer (2013)
26. Beyer, D., Wendler, P.: Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In: E. Bartocci, C.R. Ramakrishnan (eds.) Intl. Symp. on Model Checking of Software (SPIN), *Lecture Notes in Computer Science*, vol. 7976, pp. 1–17. Springer (2013)
27. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: R. Cleaveland (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 1579, pp. 193–207. Springer (1999)
28. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 196–207. ACM (2003)
29. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: D. Bjørner, M. Broy, I.V. Pottosin (eds.) Formal Methods in Programming and Their Applications, *Lecture Notes in Computer Science*, vol. 735, pp. 128–141. Springer (1993)
30. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: N. Sharygina, H. Veith (eds.) Intl. Conf. on Computer-Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 8044, pp. 413–429. Springer (2013)
31. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (1986)
32. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Symp. on Principles of Programming Languages (POPL), pp. 289–300. ACM (2009)
33. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity analysis of programs. In: M.V. Hermenegildo, J. Palsberg (eds.) Symp. on Principles of Programming Languages (POPL), pp. 57–70. ACM (2010)
34. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
35. Clarke, E.M., Kröning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: N. Halbwachs, L.D. Zuck (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 3440, pp. 570–574. Springer (2005)

36. Codish, M., Mulkers, A., Bruynooghe, M., de la Banda, M.G., Hermenegildo, M.: Improving abstract interpretations by combining domains. In: ACM Workshop on Partial Evaluation and Program Manipulation (PEPM), pp. 194–205. ACM (1993)
37. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT (1990)
38. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Symp. on Principles of Programming Languages (POPL), pp. 238–252. ACM (1977)
39. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: A.V. Aho, S.N. Zilles, B.K. Rosen (eds.) Symp. on Principles of Programming Languages (POPL), pp. 269–282. ACM (1979)
40. Cousot, P., Cousot, R.: Formal language, grammar, and set-constraint-based program analysis by abstract interpretation. In: Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA), pp. 170–181. ACM (1995)
41. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Symp. on Principles of Programming Languages (POPL), pp. 84–96. ACM (1978)
42. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957)
43. Damas, L., Milner, R.: Principal type schemes for functional languages. In: Symp. on Principles of Programming Languages (POPL), pp. 207–212. ACM (1982)
44. Dams, D., Namjoshi, K.S.: Orion: High-precision methods for static error analysis of C and C++ programs. In: F.S. de Boer, M.M. Bonsangue, S. Graf, W.P. de Roever (eds.) Intl. Symp. on Formal Methods for Components and Objects (FMCO), *Lecture Notes in Computer Science*, vol. 4111, pp. 138–160. Springer (2005)
45. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: F. Logozzo, M. Fähndrich (eds.) Intl. Symp. on Static Analysis (SAS), *Lecture Notes in Computer Science*, vol. 7935, pp. 215–237. Springer (2013)
46. Dudka, K., Peringer, P., Vojnar, T.: Predator: A shape analyzer based on symbolic memory graphs (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 412–414. Springer (2014)
47. Ermis, E., Nutz, A., Dietsch, D., Hoenicke, J., Podelski, A.: Ultimate Kojak (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 421–423. Springer (2014)
48. Falke, S., Merz, F., Sinz, C.: LLBMC: Improved bounded model checking of C programs using LLVM (competition contribution). In: N. Piterman, S.A. Smolka (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 7795, pp. 623–626. Springer (2013)
49. Fischer, J., Jhala, R., Majumdar, R.: Joining data flow with predicates. In: Intl. Symp. on Foundations of Software Engineering (FSE), pp. 227–236. ACM (2005)
50. Geser, A., Knoop, J., Lüttgen, G., Rüthing, O., Steffen, B.: Non-monotone fixpoint iterations to resolve second-order effects. In: T. Gyimóthy (ed.) Intl. Conf. on Compiler Construction (CC), *Lecture Notes in Computer Science*, vol. 1060, pp. 106–120. Springer (1996)
51. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: U. Furbach, N. Shankar (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR), LNAI 4130, pp. 281–286. Springer (2006)
52. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: O. Grumberg (ed.) Intl. Conf. on Computer-Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 1254, pp. 72–83. Springer (1997)
53. Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: M. Hofmann, M. Felleisen (eds.) Symp. on Principles of Programming Languages (POPL), pp. 277–289. ACM (2007)
54. Gulwani, S., Juvekar, S.: Bound analysis using backward symbolic execution. Tech. Rep. MSR-TR-2009-156, Microsoft Research (2009)

55. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Z. Shao, B.C. Pierce (eds.) Symp. on Principles of Programming Languages (POPL), pp. 239–251. ACM (2009)
56. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: G.C. Necula, P. Wadler (eds.) Symp. on Principles of Programming Languages (POPL), pp. 235–246. ACM (2008)
57. Gulwani, S., Necula, G.C.: A polynomial-time algorithm for global value numbering. In: R. Giacobazzi (ed.) Intl. Symp. on Static Analysis (SAS), *Lecture Notes in Computer Science*, vol. 3148, pp. 212–227. Springer (2004)
58. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: R. Gupta, S.P. Amarasinghe (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 281–292. ACM (2008)
59. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: N.D. Jones, M. Müller-Olm (eds.) Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI), *Lecture Notes in Computer Science*, vol. 5403, pp. 120–135. Springer (2009)
60. Gulwani, S., Tiwari, A.: Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In: P. Sestoft (ed.) European Symp. on Programming (ESOP), *Lecture Notes in Computer Science*, vol. 3924, pp. 279–293. Springer (2006)
61. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: M.I. Schwartzbach, T. Ball (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 376–386. ACM (2006)
62. Gulwani, S., Tiwari, A.: Assertion checking unified. In: B. Cook, A. Podelski (eds.) Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI), *Lecture Notes in Computer Science*, vol. 4349, pp. 363–377. Springer (2007)
63. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: A. Gupta, S. Malik (eds.) Intl. Conf. on Computer-Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 5123, pp. 190–203. Springer (2008)
64. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: B.G. Zorn, A. Aiken (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 292–304. ACM (2010)
65. Gurfinkel, A., Albarghouthi, A., Chaki, S., Li, Y., Chechik, M.: UFO: Verification with interpolants and abstract interpretation (competition contribution). In: N. Piterman, S.A. Smolka (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 7795, pp. 637–640. Springer (2013)
66. Gurfinkel, A., Belov, A.: FrankenBit: Bit-precise verification with many bits (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 408–411. Springer (2014)
67. Heizmann, M., Christ, J., Dietsch, D., Hoenicke, J., Lindenmann, M., Musa, B., Schilling, C., Wissert, S., Podelski, A.: Ultimate Automizer with unsatisfiable cores (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 418–420. Springer (2014)
68. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: D.V. Hung, M. Ogawa (eds.) Intl. Symp. Automated Technology for Verification and Analysis (ATVA), *Lecture Notes in Computer Science*, vol. 8172, pp. 365–380. Springer (2013)
69. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: W. Damm, H. Hermanns (eds.) Symp. on Principles of Programming Languages (POPL), pp. 232–244. ACM (2004)
70. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: E. Brinksma, K.G. Larsen (eds.) Intl. Conf. on Computer-

Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 2404, pp. 526–538. Springer (2002)
71. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: J. Launchbury, J.C. Mitchell (eds.) Symp. on Principles of Programming Languages (POPL), pp. 58–70. ACM (2002)
72. Inverso, O., Tomasco, E., Fischer, B., Torre, S.L., Parlato, G.: Lazy-CSeq: A lazy sequentialization tool for C (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 398–401. Springer (2014)
73. J. Kinder, H.V.: JAKSTAB: A static analysis platform for binaries. In: A. Gupta, S. Malik (eds.) Intl. Conf. on Computer-Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 5123, p. 423–427. Springer (2008)
74. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: J. Kramer, J. Bishop, P.T. Devanbu, S. Uchitel (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 215–224. ACM (2010)
75. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data-flow analysis and programs with recursive data structures. In: Symp. on Principles of Programming Languages (POPL), pp. 66–74. ACM (1982)
76. Jung, Y., Kong, S., Wang, B.Y., Yi, K.: Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction. In: G. Barthe, M.V. Hermenegildo (eds.) Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI), *Lecture Notes in Computer Science*, vol. 5944, pp. 180–196. Springer (2010)
77. Kam, J., Ullman, J.: Global data-flow analysis and iterative algorithms. J. ACM **23**(1), 158–171 (1976)
78. Kennedy, K.: A survey of data-flow analysis techniques. In: N.D. Jones, S.S. Muchniek (eds.) Program Flow Analysis: Theory and Applications, pp. 5–54. Prentice Hall (1981)
79. Klein, M., Knoop, J., Koschützki, D., Steffen, B.: DFA&OPT-METAFrame: A tool kit for program analysis and optimization. In: T. Margaria, B. Steffen (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 1055, pp. 422–426. Springer (1996)
80. Knoop, J., Rüthing, O., Steffen, B.: Towards a tool kit for the automatic generation of interprocedural data-flow analyses. J. Prog. Lang. **4**(4), 211–246 (1996)
81. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: Conf. on Programming Language Design and Implementation (PLDI). ACM (1992)
82. Knuth, D.E.: Semantics of context-free languages. Math. Systems Theory **2**(2), 127–145 (1968)
83. Kong, S., Jung, Y., David, C., Wang, B.Y., Yi, K.: Automatically inferring quantified invariants via algorithmic learning from simple templates. In: K. Ueda (ed.) Asian Symp. on Programming Languages and Systems (APLAS), *Lecture Notes in Computer Science*, vol. 6461, pp. 328–343 (2010)
84. Kröning, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.: Termination analysis with compositional transition invariants. In: T. Touili, B. Cook, P. Jackson (eds.) Intl. Conf. on Computer-Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 6174, pp. 89–103. Springer (2010)
85. Kröning, D., Tautschnig, M.: CBMC: C bounded model checker (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 389–391. Springer (2014)
86. Lewis, P., Rosenkrantz, D., Stearns, R.: Compiler Design Theory. Addison Wesley (1976)
87. Löwe, S., Mandrykin, M., Wendler, P.: CPAchecker with sequential combination of explicit-value analyses and predicate analyses (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 392–394. Springer (2014)

88. Mohnen, M.: A graph-free approach to data-flow analysis. In: R.N. Horspool (ed.) Intl. Conf. on Compiler Construction (CC), *Lecture Notes in Computer Science*, vol. 2304, pp. 46–61. Springer (2002)
89. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Comm. ACM **22**(1) (1979)
90. Morse, J., Ramalho, M., Cordeiro, L., Nicole, D., Fischer, B.: ESBMC 1.22 (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 405–407. Springer (2014)
91. Muller, P., Vojnar, T.: CPAlien: Shape analyzer for CPAchecker (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 395–397. Springer (2014)
92. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: N.D. Jones, X. Leroy (eds.) Symp. on Principles of Programming Languages (POPL), pp. 330–341. ACM (2004)
93. Mycroft, A.: Polymorphic type schemes and recursive definitions. In: M. Paul, B. Robinet (eds.) European Symp. on Programming (ESOP), *Lecture Notes in Computer Science*, vol. 167, pp. 217–228. Springer (1984)
94. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
95. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory. Oxford University Press (1990)
96. Pasareanu, C.S., Dwyer, M.B., Visser, W.: Finding feasible counter-examples when model checking abstracted Java programs. In: T. Margaria, W. Yi (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 2031, pp. 284–298. Springer (2001)
97. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
98. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: B. Steffen, G. Levi (eds.) Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI), *Lecture Notes in Computer Science*, vol. 2937, pp. 239–251. Springer (2004)
99. Popeea, C., Rybalchenko, A.: Threader: A verifier for multi-threaded programs (competition contribution). In: N. Piterman, S.A. Smolka (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 7795, pp. 633–636. Springer (2013)
100. Reps, T.W., Horwitz, S., Sagiv, M.: Precise interprocedural data-flow analysis via graph reachability. In: R.K. Cytron, P. Lee (eds.) Symp. on Principles of Programming Languages (POPL), pp. 49–61. ACM (1995)
101. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Symp. on Principles of Programming Languages (POPL), pp. 12–27. ACM (1988)
102. Rothermel, G., Harrold, M.: Analyzing regression test selection techniques. IEEE Trans. Softw. Eng. **22**(8), 529–551 (1996)
103. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
104. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: R. Giacobazzi (ed.) Intl. Symp. on Static Analysis (SAS), *Lecture Notes in Computer Science*, vol. 3148, pp. 53–68. Springer (2004)
105. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: Symp. on Principles of Programming Languages (POPL), pp. 318–329. ACM (2004)
106. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. Allyn and Bacon (1986)
107. Schmidt, D.A.: Data-flow analysis is model checking of abstract interpretations. In: Symp. on Principles of Programming Languages (POPL). ACM (1998)

108. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: G. Levi (ed.) Intl. Symp. on Static Analysis (SAS), *Lecture Notes in Computer Science*, vol. 1503, pp. 351–380. Springer (1998)
109. Shved, P., Mandrykin, M., Mutilin, V.: Predicate analysis with BLAST 2.7 (competition contribution). In: C. Flanagan, B. König (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 7214, pp. 525–527. Springer (2012)
110. Slaby, J., Strejcek, J.: Symbiotic 2: More precise slicing (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 415–417. Springer (2014)
111. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 223–234. ACM (2009)
112. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: M.V. Hermenegildo, J. Palsberg (eds.) Symp. on Principles of Programming Languages (POPL), pp. 313–326. ACM (2010)
113. Steffen, B.: Data-flow analysis as model checking. In: T. Ito, A.R. Meyer (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 536, pp. 346–365. Springer (1991)
114. Steffen, B.: Generating data-flow analysis algorithms from modal specifications. Science of Computer Programming **21**(2), 115–139 (1993)
115. Steffen, B.: Property-oriented expansion. In: R. Cousot, D.A. Schmidt (eds.) Intl. Symp. on Static Analysis (SAS), LNCS 1145, pp. 22–41. Springer (1996)
116. Steffen, B., Claßen, A., Klein, M., Knoop, J., Margaria, T.: The fixpoint-analysis machine. In: I. Lee, S.A. Smolka (eds.) Intl. Conf. on Concurrency Theory (CONCUR), *Lecture Notes in Computer Science*, vol. 962, pp. 72–87. Springer (1995)
117. Tomasco, E., Inverso, O., Fischer, B., Torre, S.L., Parlato, G.: MU-CSeq: Sequentialization of C programs by shared memory unwindings (competition contribution). In: E. Ábrahám, K. Havelund (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 402–404. Springer (2014)
118. Urban, C., Miné, A.: An abstract domain to infer ordinal-valued ranking functions. In: Z. Shao (ed.) European Symp. on Programming (ESOP), *Lecture Notes in Computer Science*, vol. 8410, pp. 412–431. Springer (2014)
119. Vizel, Y., Grumberg, O., Shoham, S.: Intertwined forward-backward reachability analysis using interpolants. In: N. Piterman, S.A. Smolka (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 7795, pp. 308–323. Springer (2013)
120. O. Šerý: Enhanced property specification and verification in BLAST. In: M. Chechik, M. Wirsing (eds.) Intl. Conf. on Fundamental Approaches to Software Engineering (FASE), *Lecture Notes in Computer Science*, vol. 5503, pp. 456–469. Springer (2009)