UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Simmo Saan

Witness Generation for Data-flow Analysis

Master's Thesis (30 ECTS)

Supervisor: Vesal Vojdani, PhD

Tartu 2020

Witness Generation for Data-flow Analysis

Abstract:
A program analyzer, which determines whether a given program satisfies or violates the specification, may itself contain bugs and thus be untrustworthy. Hence, the analyzer should back its claims with witnesses, which can be understood by the programmer and automatically checked by independent tools. Interprocedural data-flow analysis is well-suited for certain problems but its abstractions do not directly correspond to required witnesses. We show that witnesses can be generated with data-flow analysis by designing the necessary methods to handle interprocedurality and adapting a technique from model checking to increase precision of the generated witnesses. The ideas are implemented and experimentally evaluated in the data-flow analyzer Goblint. This allows improving trustworthiness and usability of data-flow analyzers and enables their comparison with other verifiers.

Keywords:
static analysis, software verification, witnesses, data-flow analysis, reachability, interprocedural analysis, Goblint

CERCS:
P170 Computer science, numerical analysis, systems, control

Andmevooanalüüsi tõestusobjektide loomine

Lühikokkuvõte:
Programmianalüsaator, mis määrab, kas antud programm rahuldab või rikub spetsifikatsiooni, võib ise sisaldada vigu ja seega olla ebausaldusväärne. Seepärast peaks analüsaator tõendama oma väiteid tõestusobjektidega (ingl. *witness*), mis on arusaadavad programmeerijale ja automaatselt kontrollitavad sõltumatute tööriistadega. Protseduuridevaheline andmevooanalüüs sobib hästi teatud probleemide lahendamiseks, aga selle abstraktsioonid otseselt ei vasta nõutud tõestusobjektidele. Töös näidatakse, et andmevooanalüüsiga saab tõestusobjekte luua, disainides vajalikud meetodid, millega käsitleda protseduuridevahelisust, ja kohandatakse võte mudelkontrollist, et suurendada loodud tõestusobjektide täpsust. Ideed implementeeritakse ja eksperimentaalselt testitakse andmevooanalüsaatoris Goblint. See võimaldab suurendada andmevooanalüsaatorite usaldusväärtust ja kasutatavust ning lubab neid võrrelda teiste verifitseerijatega.

Võtmesõnad:
staatiline analüüs, tarkvara verifitseerimine, tõestusobjektid, andmevooanalüüs, saavutatavus, protseduuridevaheline analüüs, Goblint

CERCS:
P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

On a very high level, program analysis can be described as follows: given a program and a specification, an analyzer should output whether the program satisfies the specification or not. Usually we are interested in specifications which state that certain problematic behavior cannot occur during the execution of the program. Common examples of specifications include [1]:

1. Certain error function is never called.
2. Assertions in the program always hold.
3. Dynamically allocated memory is not accessed after it is freed.
4. There are no memory leaks, i.e., all dynamically allocated memory is eventually freed.
5. No numerical computation overflows.
6. The program always terminates, i.e., will not get stuck in an infinite loop.

If the program satisfies the specification, it is said to be *correct* and the analyzer should output "true". If the program does not satisfy the specification, it is said to *violate* it and be *incorrect*, and the analyzer should output "false". Just a true/false output is unhelpful and unsatisfactory: When the specification is violated, the programmer wants to fix the program such that it would not violate the specification anymore. For any nontrivial program, this would be difficult because the output gives no indication of where in the program and under what circumstances the violation happens. Since the analyzer found the violation, it should be able to also give those details. Furthermore, it is unclear whether the analyzer actually did any meaningful analysis or simply "flipped a coin" to produce its output. More realistically, the analyzer might itself contain bugs which affect the output. Either way, the analyzer should not be blindly trusted to always be right.

Therefore, it is desirable that the analyzer also outputs some kind of proof corresponding to its true/false output. This would give insight to the programmer, why the specification holds or does not hold, and allow the proof to be independently checked, significantly increasing trustworthiness. Writing complete and rigorous machine-checkable proofs is difficult even for humans, let alone automatic program analyzers. Thus, we consider proof objects called *witnesses* instead, which convey the key information of a proof, for example:

- To prove that a violation occurs in the program, one can give a concrete *error path*, which is a sequence of program locations from the beginning of the program to a violating location. In addition, it may specify concrete return values for non-deterministic operations (e.g., randomness, input), which lead to the violation.

- To prove that no violation occurs in the program, one can give *invariants* for some program locations, usually for loops. They are a key component of standard program correctness proofs, e.g., in Hoare logic [2].

Error paths and invariants are easily understandable by humans, to reproduce the error or to see why one does not happen, and automatically checkable by independent tools.Witnesses for violating programs are called *violation witnesses* and witnesses for correct programs are called *correctness witnesses*.

Furthermore, witnesses allow comparing the analyzers that generated them. The "Competition on Software Verification" (SV-COMP) is an annual competition for fully automatic software verifiers of C (and Java) programs [1, 3]. It has a strong focus on replicability and requires competing analyzers to produce witnesses, which are validated by independent analyzers (called witness validators), in order to score points. Analyzers compete in various categories based on their support for specifications (also called properties) listed above and other complicating factors of the programs, e.g., use of arrays, pointers and recursion.

Contribution. Goblint is a static analyzer for multi-threaded C programs [4, 5]. It is developed at the University of Tartu and the Technical University of Munich. The plan is to have Goblint compete in SV-COMP. It is based on data-flow analysis, which well-suited for certain problems but unfortunately is not directly amenable to generating the required witnesses, due to its abstractions. The goal of this thesis is to design and implement methods for that. Doing so also makes Goblint more usable and trustworthy.

We only focus on the error function unreachability property because it is directly modeled by data-flow analysis. It is also the most common property used in the competition, and verification of many other interesting properties can be recast as verification of unreachability. Therefore, this choice will not be a major limitation.

The key contribution of this thesis is the design of methods which allow generating witnesses from data-flow analysis. Thorough overview of the challenges, which arise in the process, is presented with examples. Applicability and correctness of these methods is validated by implementing them in Goblint and evaluating the generated witnesses.

Organization of the thesis. Section 2 makes the notion of witnesses formal and describes the SV-COMP witness exchange format. Section 3 is dedicated to interprocedural programs: their witnesses, their data-flow analysis and generating the former from the latter. The Goblint implementation is described. Section 4 addresses two issues of the witness generation from the previous section: imprecise violation witnesses and violation witness false positives. The Goblint implementation of a technique which mitigates both issues is described.

2 Witnesses in program analysis

In this section, the abstract notion of witnesses as proof objects is made concrete by formally defining them as automata. This formalism is quite flexible as it is not too strongly tied to any program analysis approach and admits witnesses of highly variable levels of precision. This theoretical background is necessary for understanding the witness exchange format and the witness validation of SV-COMP.

Let S be the set of possible primitive program statements, which may be limited to the following [6]:

- assignment statements,
- conditional guard statements, which are used to model ifs and loops [7],
- function call statements, which are used in interprocedural programs.

Both witnesses and data-flow analysis use the following standard representation for programs:

Definition 1 (Beyer et al. [8])**.** A **control-flow automaton** (CFA) a.k.a. **control-flow graph** (CFG) $C = (L, E, l_0)$ is a finite automaton with:

- the set L of program locations as states,
- the alphabet S,
- the transition relation $E \subseteq L \times S \times L$,
- the initial program location $l_0 \in L$ as initial state.

We write $(u, s, v) \in E$ as $u \xrightarrow[E]{s} v$ but omit arrow subscripts when they are unambiguous from the context. A sequence $l_0 \xrightarrow{s_1} l_1 \xrightarrow{s_2} \cdots \xrightarrow{s_{n-1}} l_{n-1} \xrightarrow{s_n} l_n$ is called a *program path*.

2.1 Witness automata

Violation and correctness witnesses are represented by slightly different forms of automata, which are defined in this subsection. In addition to the definitions, intuition is given as to how the witness automata are used and validated. Knowing the latter is necessary for understanding the meaning of their components and constructing witnesses that are actually useful.

Let Φ be the set of possible predicates over the program variables.

Definition 2 (Beyer et al. [8])**.** A **violation witness automaton** $W = (Q, \Sigma, \delta, q_0, F)$ for a CFA $C = (L, E, l_0)$ is a non-deterministic finite automaton with:

- the finite set Q of states,
- the alphabet $\Sigma \subseteq \mathcal{P}(E) \times \Phi$,
- the transition relation $\delta \subseteq Q \times \Sigma \times Q$,
- the initial state $q_0 \in Q$,
- the set $F \subseteq Q$ of accepting states.

We write $(u, (D, \psi), v) \in \delta$ as $u \xrightarrow[\delta]{(D,\psi)} v$. A sequence $q_0 \xrightarrow[\delta]{(D_1,\psi_1)} \cdots \xrightarrow[\delta]{(D_n,\psi_n)} q_n$ is a *simulation sequence* of a program path $l_0 \xrightarrow[E]{s_1} \cdots \xrightarrow[E]{s_n} l_n$ if every step $l_{i-1} \xrightarrow[E]{s_i} l_i$ corresponds to a step $q_{i-1} \xrightarrow[\delta]{(D_i,\psi_i)} q_i$ such that $(l_{i-1}, s_i, l_i) \in D_i$ and ψ_i is true at program location l_i. The simulation sequence is accepted if $q_n \in F$ and the program path is accepted if there exists an accepting simulation sequence for it.

Validation. A witness validator uses the violation witness automaton as an additional component of its analysis, making simulation steps as it makes steps through the program. The witness is considered confirmed when the validator reaches a specification violation (e.g., an error location) *and* the automaton is in an accepting state. The violation witness automaton can lead the validator to the violation through the following means:

Transitions By being in a state without outgoing transitions, the witness stops the validator from analyzing that program state any further because there are no corresponding simulation steps, thereby restricting state-space exploration.

Edge sets By having a transition whose edge set contains only some (usually just one) of the possible edges, the witness forces the validator to only analyze paths taking those edges, thereby restricting state-space exploration.

Predicates By having a transition whose predicate is non-trivial, the witness stops the validator from analyzing program states where the predicate does not hold, thereby restricting state-space exploration.

Through restricting state-space exploration, the violation witness makes it easier and faster for the validator to confirm it if the witness is valid. For a more detailed description of violation witness validation, see Beyer et al. [8].

Definition 3 (Beyer et al. [9])**.** A **correctness witness automaton** $W = (Q, \Sigma, \delta, q_0, I)$ for a CFA $C = (L, E, l_0)$ is a non-deterministic finite automaton with:

- the finite set Q of states,
- the alphabet $\Sigma \subseteq \mathcal{P}(E) \times \Phi$,
- the transition relation $\delta \subseteq Q \times \Sigma \times Q$,
- the initial state $q_0 \in Q$,
- the invariant function $I : Q \to \Phi$,

where the transition relation is such that for every $q \in Q$ and $e \in E$:

$$\bigvee \left\{ \psi \in \Phi \;\middle|\; \exists q' \in Q, \exists D \subseteq E, q \xrightarrow{(D,\psi)} q' \wedge e \in D \right\} \tag{1}$$

is true.

Analogously to violation witness automata, simulation sequences can be defined. Unlike violation witness automata, there is no notion of acceptance and the transition relation is not allowed to restrict the state space in any way, which is ensured by condition (1). Hence, the correctness witness automaton must be able to simulate every step taken by the program. The correctness information is carried by the invariants assigned to witness states. Note that simulation is independent of the invariants.

Validation. Analogously, a witness validator uses the correctness witness automaton as an additional component of its analysis. The witness is considered confirmed when the validator verifies that the invariants hold at their corresponding states *and* no specification violation (e.g., an error location) is reachable given the invariants. The correctness witness automaton can lead the validator to the correctness proof through the following means:

Invariants By providing strong enough invariants, the witness makes it easy for the validator to verify the correctness without having to come up with sufficient invariants itself.

Partitioning Even though the correctness witness may not restrict the state space, it may use the predicates on transitions to partition the state space arbitrarily. By partitioning the witness can supply different and more precise invariants for different cases.

For a more detailed description of correctness witness validation, see Beyer et al. [9].

2.2 Abstract reachability graphs

A priori an analyzer does not know whether the program violates the specification or not and hence cannot know whether to start constructing a violation or a correctness witness. Thus, a more general notion is required. For this reason, many analyzers use the following as their internal representation of the reachable state space of the program that is being analyzed [8, 9]:

Definition 4. An **abstract reachability graph** (ARG) $A = (Q, \Sigma, \delta, q_0)$ is a directed graph with:

- the finite set Q of states,
- the alphabet Σ,
- the transition relation $\delta \subseteq Q \times \Sigma \times Q$,
- the initial state $q_0 \in Q$

that represents possibly reachable states of a program on a suitable level of abstraction.

We write $(u, \sigma, v) \in \delta$ as $u \xrightarrow[\delta]{\sigma} v$. A CFA $C = (L, E, l_0)$ is trivially also an ARG $A = (L, S, E, l_0)$, albeit without useful reachability information. In Section 3 CFAs of functions are combined to construct an ARG for an interprocedural program, also containing reachability information from data-flow analysis.

Conversion to witness automaton. If the ARG contains an abstract state which violates the specification, then a violation witness automaton is constructed; otherwise, a correctness witness automaton is constructed. Either way, the witness states are exactly the abstract states and the initial witness state is the initial abstract state. The witness transitions are constructed directly from the abstract transitions through a suitable mapping of the alphabet, for example using a singleton set and a true predicate.[1]

In case of a violation witness, the accepting witness states are the abstract states which violate the specification. In case of a correctness witness, if the analyzer knows any invariants, then they are used as the witness invariants; otherwise, trivial true predicates are also sound.

2.3 SV-COMP

In SV-COMP, a competing analyzer is given a verification task, consisting of a program and a property to check, and it should output a verdict, whether the specification is violated or not, and a corresponding witness (automaton). The produced witness is validated using independent analyzers.

Verification task programs. SV-COMP verification task programs use some non-standard functions, which have special meaning to the analyzers [1]. The error function, the reachability of which we are interested in, is called `__VERIFIER_error`.

It is common, both in the competition and in general, to specify correctness using assertions with boolean C expressions. These are recast as error function reachability via the following function:

```
void __VERIFIER_assert(int cond)
{
    if (!cond)
        __VERIFIER_error();
}
```

For brevity, we will call these functions `error` and `assert`, respectively.

Additionally, the programs use helper functions, e.g., `__VERIFIER_nondet_int`, which return nondeterministic values of the given primitive type.

Witness validation. Witness automata have no objective and formal requirements for how precise they need to be. Violation witnesses should restrict state-space exploration in the witness validator, but the amount of restriction necessary is not mathematically

[1] To be precise, such mapping requires the ARG alphabet to be related to the CFA, which is true in practice and in the ARGs generated in Section 3.

specified. Correctness witnesses should provide invariants for the witness validator, but the amount of invariants necessary nor their usefulness is not mathematically specified.

Therefore, witness validation in SV-COMP is entirely subjective: given the witness with its information, the validator needs to analyze the program and confirm the witness within a time limit. Violation witness validation has a time limit (90 seconds), which is ten times shorter than the time limit for the competing analyzer (15 minutes). The witness must restrict the state space enough for its validating analysis to be that much faster. Correctness witness validation has the same time limit (15 minutes) as the competing analyzer because the state space is not reduced. Still, the witness must provide enough useful invariants for its validating analysis to accept the verdict in the given time.

Witness exchange format. While many analyzer-specific formats for violation or correctness witnesses exist, SV-COMP defines their own general-purpose witness format based on witness automata. This makes the witnesses of different analyzers easier to compare and automatically check.

Since automata can be viewed as graphs, the SV-COMP witness exchange format is based on GraphML [8–10], which is an XML format for describing graphs. It allows adding custom data to nodes and edges. The most important SV-COMP specific GraphML data keys are listed and described in Table 1.

Conversion from witness automaton. Given either a violation witness automaton $W = (Q, \Sigma, \delta, q_0, F)$ or a correctness witness automaton $W = (Q, \Sigma, \delta, q_0, I)$ for a CFA $C = (L, E, l_0)$, it is converted to the SV-COMP witness exchange format as follows:

Nodes The witness states Q become the nodes of the graph.

Edges The witness transitions δ become the edges of the graph.

Entry node The initial witness state q_0 is marked as the entry node.

Edge line numbers The elements of the edge sets D from the witness transition alphabet Σ correspond to source code line numbers which are added to the edges. Since each edge can specify at most one corresponding line number, if $|D| > 1$, the edge must be duplicated $|D|$ times with each one specifying a different line number.

Control-flow edges The conditional guard statement transitions in E, which correspond to true and false branches of ifs and loops, are marked as such control-flow edges.

Loop head nodes As additional preprocessing, loop heads are identified by depth-first search (DFS) traversal of C, starting from l_0, as the target nodes of back edges [11].

Table 1. SV-COMP witness exchange format data [8–10].

(a) General witness data.

For	Key	Value	Description
Graph	`witness-type`	`violation_witness` or `correctness_witness`	Type of witness.
Node	`entry`	`true` or `false`	Start state of the program. Exactly one allowed.
Edge	`control`	`condition-true` or `condition-false`	Which conditional branch the edge corresponds to. Used for ifs and loops.
	`enterLoopHead`	`true` or `false`	Edge goes to a loop head (first node of a loop, target of back jump). Used for loops.
	`enterFunction`	Function name	Edge goes into a function.
	`returnFromFunction`	Function name	Edge goes out of a function. Must correspond to `enterFunction`.
	`startline`	Line number	Which source code line's statement the edge corresponds to.

(b) Violation witness data.

For	Key	Value	Description
Node	`violation`	`true` or `false`	Violation state.
	`sink`	`true` or `false`	State which cannot lead to a violation.
Edge	`assumption`	C expression	Condition for the edge. Used for reducing state space by specifying concrete values.

(c) Correctness witness data.

For	Key	Value	Description
Node	`invariant`	C expression	Invariant for the state.

In case of a violation witness, the following data is added:

Violation nodes The accepting witness states F are marked as the violation nodes and their outgoing edges are omitted since program execution after a violation is irrelevant.

Sink nodes As additional preprocessing, sink nodes are identified by backwards graph traversal, starting from all violation nodes, as the nodes which are unreachable by that traversal. They are marked as sink nodes and their outgoing edges are omitted since program execution from there cannot lead to a violation.

Edge assumptions The predicates ψ from the witness transition alphabet Σ are added as assumptions for the edges.

In case of a correctness witness, the following data is added:

Node invariants For each witness state the invariant function I is called to get the invariant predicate.

Note that even though correctness witness automata by Definition 3 can use predicates on transitions to partition the state space, the SV-COMP witness exchange format does not allow that, except for the partitioning induced by conditional guard statements of control-flow edges.

3 Witnesses from interprocedural data-flow analysis

In this section, interprocedural programs and their ARGs are introduced. It is then shown how to use data-flow analysis to construct such ARGs, which in turn allow generating witnesses.

An interprocedural program consists of multiple functions (procedures), which can call each other. Data-flow analysis uses the following standard representation for such programs. Let F be the set of functions of the program. For each function $f \in F$, let $C_f = (L_f, E_f, \mathrm{entry}_f)$ be its CFA. Without loss of generality, assume there is exactly one return location $\mathrm{return}_f \in L_f$ for the function. Let the function $\mathrm{main} \in F$ be the starting point of the program. Let $L = \bigcup_{f \in F} L_f$ and $E = \bigcup_{f \in F} E_f$.

3.1 Interprocedural ARGs

In the intraprocedural case, an ARG can be the CFA of the entire program. Since an ARG must be a single connected graph, which includes edges for entering and returning from functions, standalone CFAs of the functions need to be combined.

The basic idea is to replace each function call edge with two edges: one entering the called function from the call-site and another returning from the called function to the return-site. For example, consider the program from Figure 1a. The CFAs of its functions are shown in dotted rectangles in Figure 1b and the resulting ARG is obtained by removing the edge $1 \xrightarrow{\texttt{foo(1)}} 2$ and adding the edges $1 \to a$ and $b \to 2$.

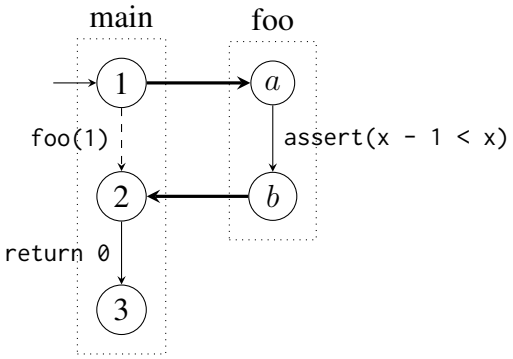
This corresponds to inlining the CFA of the called function inside the CFA of the calling function. The resulting graph is referred to as the *interprocedural supergraph* [7] of the program. Unfortunately, this simple inlining is too inaccurate for an ARG, as the following examples will explain.

Context-sensitive inlining. Consider the program from Figure 2a, which calls the same function multiple times with different arguments. Basic inlining of its function calls results in the ARG in Figure 3a. Even though the function is called with different arguments, which can lead to different behavior, the corresponding invariants inside the function are merged. For example, an interval analysis can at best establish that $\texttt{x} \in [1, 2]$ at node a, which is insufficient for interval arithmetic to establish that `x - 1 < x` is always true.

Therefore, it is desired to obtain the ARG in Figure 3b, where different contexts (arguments) of the same function are separated. This allows $\texttt{x} \in [1, 1]$ at node a and $\texttt{x} \in [2, 2]$ at node a', both of which are sufficient for interval arithmetic to establish that the corresponding assertion is always true.

```c
void foo(int x)
{
    assert(x - 1 < x);
}

int main()
{
    foo(1);
    return 0;
}
```

(a) C source code.



(b) Function CFAs and resulting ARG.

Figure 1. Basic inlining of function calls for ARG.

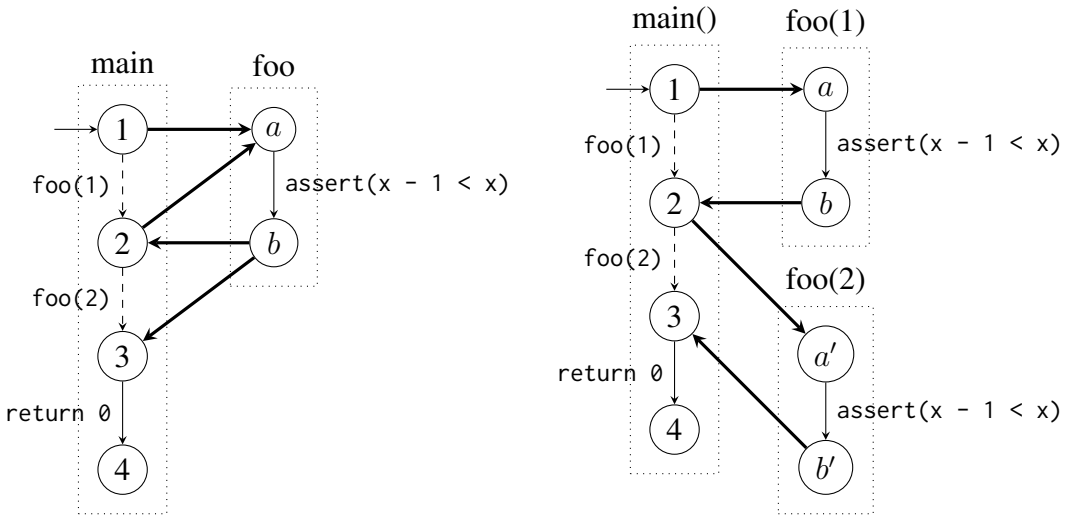
```c
void foo(int x)
{
    assert(x - 1 < x);
}

int main()
{
    foo(1);
    foo(2); // cf. Figure 1a
    return 0;
}
```

(a) C source code of a program which requires context-sensitive inlining.

```c
void foo(int x)
{
    assert(x - 1 < x);
}

int main()
{
    foo(1);
    foo(1); // cf. Figure 2a
    return 0;
}
```

(b) C source code of a program which requires call-site–sensitive inlining.

Figure 2. C source code of programs for which basic inlining does not suffice.



(a) Resulting ARG without context-sensitive inlining.

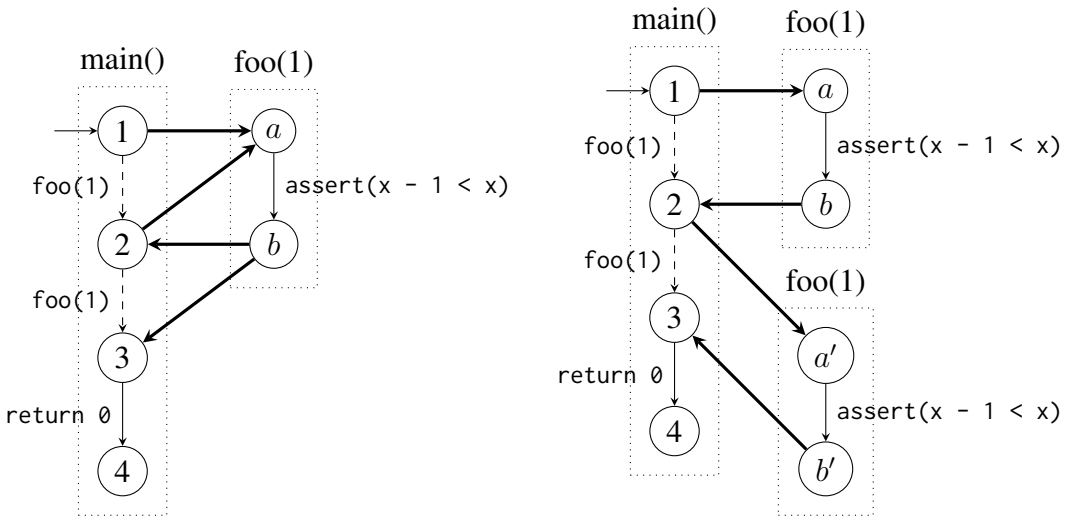
(b) Desired ARG with context-sensitive inlining.

Figure 3. Context-sensitive inlining of function calls for ARG of program from Figure 2a.

Call-site–sensitive inlining. Consider the program from Figure 2b, which calls the same function multiple times with the same argument. Context-sensitive inlining of its function calls results in the ARG in Figure 4a. The problem here is that the program appears to have a cycle $2 \to a \to b \to 2$ while it actually does not have one. Moreover, it appears to be possible that the function only gets called once on the path $1 \to a \to b \to 3$.

Therefore, it is desired to obtain the ARG in Figure 4b, where different call sites of the same function in the same context are separated. This avoids introducing infeasible paths.

Note that call-site–sensitive inlining without context-sensitive inlining produces a correct but imprecise ARG. Considering the previous program from Figure 2a, the result is the ARG in Figure 5. Due to context-insensitivity $\texttt{x} \in [1, 2]$ at both node a and a', which is insufficient for interval arithmetic to establish that `x - 1 < x` is always true.



(a) Resulting ARG without call-site–sensitive inlining.

(b) Desired ARG with call-site–sensitive inlining.

Figure 4. Call-site–sensitive inlining of function calls for ARG of program from Figure 2b.

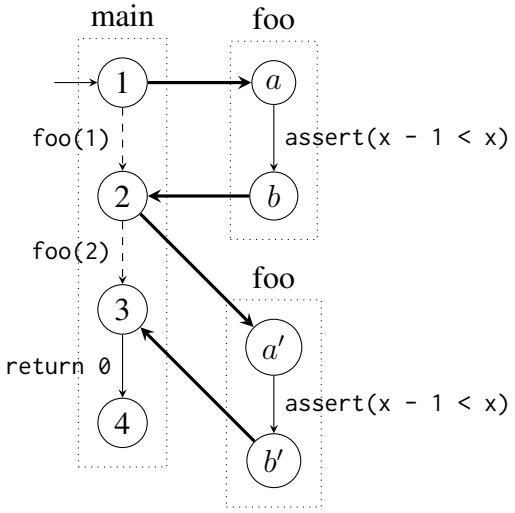


Figure 5. Resulting ARG with call-site–sensitive but without context-sensitive inlining of program from Figure 2b.

3.2 Interprocedural data-flow analysis

Data-flow analyses of programs come in various flavors, depending on whether they are intraprocedural or interprocedural. The latter may be context-insensitive, fully or partially context-sensitive. For an introduction, see Seidl et al. [7] and Apinis et al. [6]. For generality and closeness to the Goblint implementation, we skip simpler forms and immediately consider the following:

Definition 5 (Apinis et al. [6])**.** A **partially context-sensitive interprocedural data-flow analysis** is defined by:

- an abstract domain $\mathbb{D}$, which forms a complete lattice with partial order $\sqsubseteq$ and least element $\bot$,
- a set of contexts $\mathbb{C}$,
- an initial context $c_{\text{main}} \in \mathbb{C}$ and an initial abstract state $d_{\text{main}} \in \mathbb{D}$,
- for each $s \in S$, an abstract operation $[\![s]\!]^\sharp : \mathbb{D} \to \mathbb{D}$,
- an abstract operation for calling functions $\mathsf{enter}^\sharp : \mathbb{D} \to \mathbb{C} \times \mathbb{D}$ and an abstract operation for returning from functions $\mathsf{combine}^\sharp : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$.

For each $s \in S$, the operation $[\![s]\!]^\sharp$ abstractly executes the statement s on the abstract state. The operation $\mathsf{enter}^\sharp$ returns the context and initial abstract state for the function being called using the abstract state at the call site. The operation $\mathsf{combine}^\sharp$ combines the abstract state at the call site with the abstract state returned from the called function.

Partially context-sensitive data-flow analyses are performed by solving constraint systems, which come in various flavors, depending on which formalism is used and how constraints are generated. For an introduction, see Apinis et al. [6]. For reachability properties and closeness to Goblint implementation, we skip simpler forms and immediately consider the following:

Definition 6 (Apinis et al. [6])**.** A **side-effecting constraint system** over a complete lattice $\mathbb{D}$ with variables V is a set of pairs (called constraints) (x, f), where $x \in V$ and the transfer function is of the type

$$f : (V \to \mathbb{D}) \to (V \to \mathbb{D} \to \mathsf{unit}) \to \mathbb{D},$$

where $\mathsf{unit} = \{()\}$. The first argument (called get) provides access to the variable assignment and the second argument (called set) side-effectingly provides a contribution to a variable.

Instead of mathematical function syntax, we use lambda calculus syntax, notably $f\,x$ for function application instead of $f(x)$.

Definition 7 (Apinis et al. [6])**.** A variable assignment $\sigma : V \to \mathbb{D}$ is a **solution to a side-effecting constraint system** if for each (x, f) it holds that $\sigma\,x \sqsupseteq f\,\sigma\,\mathsf{set}$ where for each call $\mathsf{set}\,y\,d$ during evaluation of f it holds that $\sigma\,y \sqsupseteq d$.

When solving a constraint system, instead of any solution we aim to compute the *least* solution, which corresponds to the most precise analysis of the program.

We write $x \sqsupseteq e$ to mean $(x, \lambda\mathsf{get}.\, \lambda\mathsf{set}.\, e)$ in side-effecting constraint systems. Moreover, we use ML-like let expressions to write side-effecting constraints. For a given program the analysis forms the following side-effecting system of constraints with variables $V = L \times \mathbb{C}$ [6]:

$$[\mathrm{entry}_{\mathrm{main}}, c_{\mathrm{main}}] \sqsupseteq d_{\mathrm{main}} \tag{2a}$$

$$[v, c] \sqsupseteq [\![s]\!]^\sharp\,(\mathsf{get}\,[u, c]) \qquad \forall u \xrightarrow{s} v \tag{2b}$$

$$\begin{aligned}[v, c] \sqsupseteq\ & \textbf{let}\,(c', d') = \mathsf{enter}^\sharp\,(\mathsf{get}\,[u, c]) \\ & ()\ =\ \mathsf{set}\,[\mathrm{entry}_f, c']\,d' \\ & \textbf{in}\ \mathsf{combine}^\sharp\,(\mathsf{get}\,[u, c])\,(\mathsf{get}\,[\mathrm{return}_f, c'])\end{aligned} \qquad \forall u \xrightarrow{f()} v \tag{2c}$$

universally quantified over $c \in \mathbb{C}$. The transitions in (2b) and (2c) are from E. By convention, (2c) is quantified over statements which are calls to some function $f \in F$, while (2b) is quantified over statements which are not function calls. Without loss of generality, only function call statements with no parameters and no return value are considered because parameter and return value passing can be modeled using auxiliary (global) variables [6, 7].

Reachability. In a constraint system solution, reachable variables, which are program locations in contexts, have non-$\bot$ values and unreachable variables have $\bot$ values. Intuitively, solving starts with all variables being unreachable, except the beginning of the program is made reachable through (2a). Then through (2b) and (2c) other reachable variables are eventually made reachable. This is known as a *forward analysis* [7]. Therefore, the constraint system and its least solution directly contain reachability information, which will be exploited below to construct an ARG.

We require the right-hand sides and thus the abstract operations to be such that they do not spuriously introduce a reachable state from an unreachable one. Formally, this means that they are *strict*, i.e., they preserve $\bot$.

Local solving. In many analyses, the set of contexts $\mathbb{C}$ is infinite. This means that the constraint system is also infinite and cannot be completely solved in practice. However, the set of contexts which are actually reached in the analysis of a given program is finite and the solution only contains finitely many non-$\bot$ variables. Therefore, such constraint system can be solved using *local solvers* which obtain *partial solutions* [6] that are not only finite but also describe reachability of contexts.

3.3 Witness analysis

Unlike model-checking–based analyses, data-flow analysis does not directly use and generate ARGs [12]. This is why getting ARGs from data-flow analysis is non-trivial.

We first focus on context-sensitive inlining (see Section 3.1), which is achieved by extending the constraint system (2) to record the context-sensitive predecessor relationship between program locations (and contexts) in the abstract domain without altering the original analysis. Conceptually, this is similar to the ARG CPA used in configurable program analysis [13].

Let $S' = S \cup \{\mathrm{enterFunction}_f \mid f \in F\} \cup \{\mathrm{returnFromFunction}_f \mid f \in F\}$, where the added labels will be used for inlining:

- $\mathrm{enterFunction}_f$ marks the entry to function f,
- $\mathrm{returnFromFunction}_f$ marks the return from function f.

Define a new abstract domain $\overline{\mathbb{D}} = \mathbb{D} \times \mathcal{P}(V \times S')$. Let $\langle\cdot\rangle_1$ and $\langle\cdot\rangle_2$ denote the left and right projection operators, respectively. Consider the following system of constraints over $\overline{\mathbb{D}}$:

$$[\mathrm{entry}_{\mathrm{main}}, c_{\mathrm{main}}] \sqsupseteq (d_{\mathrm{main}}, \varnothing) \tag{3a}$$

$$[v, c] \sqsupseteq (\llbracket s \rrbracket^\sharp \langle \mathsf{get}\, [u, c] \rangle_1, \{([u, c], s)\}) \qquad \forall u \xrightarrow{s} v \tag{3b}$$

$$\begin{aligned} [v, c] \sqsupseteq\ & \textbf{let}\ (c', d') = \mathsf{enter}^\sharp \langle \mathsf{get}\, [u, c] \rangle_1 \\ & \qquad () = \mathsf{set}\, [\mathrm{entry}_f, c']\, (d', \{([u, c], \mathrm{enterFunction}_f)\}) \\ & \textbf{in}\ (\mathsf{combine}^\sharp \langle \mathsf{get}\, [u, c] \rangle_1 \langle \mathsf{get}\, [\mathrm{return}_f, c'] \rangle_1, \\ & \qquad \{([\mathrm{return}_f, c'], \mathrm{returnFromFunction}_f)\}) \end{aligned} \qquad \forall u \xrightarrow{f()} v \tag{3c}$$

The first components of this system mirror the original system exactly, as proven by the following theorem. Hence, by solving the extended system, the original analysis is performed.

Theorem 1 (Soundness)**.** *Let $\overline{\sigma} : V \to \overline{\mathbb{D}}$ be a solution of the extended system* (3) *with the side-effecting function $\overline{\mathsf{set}} : V \to \overline{\mathbb{D}} \to \mathsf{unit}$.*

Then $\sigma = \langle\cdot\rangle_1 \circ \overline{\sigma}$ (of the type $\sigma : V \to \mathbb{D}$) is the solution of the original system (2) *with the side-effecting function $\mathsf{set} : V \to \mathbb{D} \to \mathsf{unit}$.*

Proof.

1. We have
$$\overline{\sigma}\, [\mathrm{entry}_{\mathrm{main}}, c_{\mathrm{main}}] \sqsupseteq (d_{\mathrm{main}}, \varnothing).$$
Thus
$$\sigma\, [\mathrm{entry}_{\mathrm{main}}, c_{\mathrm{main}}] = \langle \overline{\sigma}\, [\mathrm{entry}_{\mathrm{main}}, c_{\mathrm{main}}] \rangle_1 \sqsupseteq d_{\mathrm{main}}.$$

2. For all $u \xrightarrow{s} v$, we have

$$\overline{\sigma}\,[v,c] \sqsupseteq (\llbracket s \rrbracket^\sharp\,\langle\overline{\sigma}\,[u,c]\rangle_1, \{([u,c], s)\}).$$

Thus

$$\sigma\,[v,c] = \langle\overline{\sigma}\,[v,c]\rangle_1 \sqsupseteq \llbracket s \rrbracket^\sharp\,\langle\overline{\sigma}\,[u,c]\rangle_1 = \llbracket s \rrbracket^\sharp\,(\sigma\,[u,c]).$$

3. For all $u \xrightarrow{f()} v$, we have

$$\begin{aligned}\overline{\sigma}\,[v,c] \sqsupseteq\ & \textbf{let}\ (c', d') = \mathsf{enter}^\sharp\,\langle\overline{\sigma}\,[u,c]\rangle_1 \\ & \quad () = \overline{\mathsf{set}}\,[\mathrm{entry}_f, c']\,(d', \{([u,c], \mathrm{enterFunction}_f)\}) \\ & \textbf{in}\ (\mathsf{combine}^\sharp\,\langle\overline{\sigma}\,[u,c]\rangle_1\,\langle\overline{\sigma}\,[\mathrm{return}_f, c']\rangle_1, \\ & \quad \{([\mathrm{return}_f, c'], \mathrm{returnFromFunction}_f)\})\end{aligned}$$

with the side effect

$$\overline{\sigma}\,[\mathrm{entry}_f, c'] \sqsupseteq (d', \{([u,c], \mathrm{enterFunction}_f)\}).$$

Thus

$$\begin{aligned}\sigma\,[v,c] = \langle\overline{\sigma}\,[v,c]\rangle_1 \sqsupseteq\ & \textbf{let}\ (c', d') = \mathsf{enter}^\sharp\,\langle\overline{\sigma}\,[u,c]\rangle_1 \\ & \quad () = \overline{\mathsf{set}}\,[\mathrm{entry}_f, c']\,(d', \{([u,c], \mathrm{enterFunction}_f)\}) \\ & \textbf{in}\ \mathsf{combine}^\sharp\,\langle\overline{\sigma}\,[u,c]\rangle_1\,\langle\overline{\sigma}\,[\mathrm{return}_f, c']\rangle_1 \\ =\ & \textbf{let}\ (c', d') = \mathsf{enter}^\sharp\,(\sigma\,[u,c]) \\ & \quad () = \mathsf{set}\,[\mathrm{entry}_f, c']\,d' \\ & \textbf{in}\ \mathsf{combine}^\sharp\,(\sigma\,[u,c])\,(\sigma\,[\mathrm{return}_f, c'])\end{aligned}$$

with the side effect

$$\sigma\,[\mathrm{entry}_f, c'] = \langle\overline{\sigma}\,[\mathrm{entry}_f, c']\rangle_1 \sqsupseteq d'.$$ ■

Reachability and local solving. The modified analysis relies on a local solver finding the least partial solution, which mirrors reachability, to only record reachable predecessor relationships, which are used to construct an ARG below.

In order for finite partial solutions to exist and local solving to work, the right-hand sides must be strict. The right-hand sides of (3) record unreachable predecessors, making them non-strict. This is easily fixed by wrapping the right-hand sides with the following function $\mathrm{strict}_1 : \overline{\mathbb{D}} \to \overline{\mathbb{D}}$ which ensures strictness:

$$\mathrm{strict}_1\,(a,b) = \begin{cases}(\bot,\bot), & \text{if } a = \bot, \\ (a,b), & \text{otherwise.}\end{cases}$$

Conversion to context-sensitively inlined ARG. The second components of the extended system define the desired ARG, as proven by the following theorem. Hence, by solving the extended system, we get the ARG corresponding to the original analysis.

Theorem 2 (Context-sensitivity)**.** *Let $\overline{\sigma} : V \to \overline{\mathbb{D}}$ a solution of the extended system* (3) *with the side-effecting function $\overline{\mathsf{set}} : V \to \overline{\mathbb{D}} \to \mathsf{unit}$.*

Let $\sigma' = \langle\cdot\rangle_2 \circ \overline{\sigma}$ (of the type $\sigma' : V \to \mathcal{P}(V \times S')$). Define δ by

$$u \xrightarrow[\delta]{s} v \quad \Longleftrightarrow \quad (u, s) \in \sigma'\, v,$$

Then $A_{\text{context}} = (V, S', \delta, [\text{entry}_{\text{main}}, c_{\text{main}}])$ is the context-sensitively inlined ARG.

Proof.

1. For all $u \xrightarrow[E]{s} v$,
$$\sigma'\,[v,c] = \langle\overline{\sigma}\,[v,c]\rangle_2 \supseteq \{([u,c], s)\},$$
which gives the transition $[u,c] \xrightarrow[\delta]{s} [v,c]$.

2. For all $u \xrightarrow[E]{f()} v$,

$$\begin{aligned}\sigma'\,[v,c] = \langle\overline{\sigma}\,[v,c]\rangle_2 \supseteq \textbf{let}\,(c', d') &= \mathsf{enter}^\sharp\,\langle\overline{\sigma}\,[u,c]\rangle_1\\ () &= \overline{\mathsf{set}}\,[\text{entry}_f, c']\,(d', \{([u,c], \text{enterFunction}_f)\})\\ \textbf{in}\,&\{([\text{return}_f, c'], \text{returnFromFunction}_f)\},\end{aligned}$$

which gives the transition $[\text{return}_f, c'] \xrightarrow[\delta]{\text{returnFromFunction}_f} [v,c]$, with the side effect

$$\sigma'\,[\text{entry}_f, c'] = \langle\overline{\sigma}\,[\text{entry}_f, c']\rangle_2 \supseteq \{([u,c], \text{enterFunction}_f)\},$$

which gives the transition $[u,c] \xrightarrow[\delta]{\text{enterFunction}_f} [\text{entry}_f, c']$.

Altogether, instead of $[u,c] \xrightarrow[E]{f()} [v,c]$, we get

$$[u,c] \xrightarrow[\delta]{\text{enterFunction}_f} [\text{entry}_f, c'] \to \ldots \to [\text{return}_f, c'] \xrightarrow[\delta]{\text{returnFromFunction}_f} [v,c].$$

■

Conditional inlining. The analysis defined by the constraint system (3) inlines every single function call, which may be undesired. More flexible inlining can be achieved by replacing the function call constraint (3c) with the following:

$$
\begin{aligned}
[v,c] \sqsupseteq\ & \textbf{let}\ (c', d') = \mathsf{enter}^\sharp\, \langle \mathsf{get}\, [u,c] \rangle_1 && \forall u \xrightarrow{f()} v \quad (4)\\
& \quad () = \mathsf{set}\, [\mathrm{entry}_f, c']\, (d', \bot)\\
& \quad d'' = \mathsf{combine}^\sharp\, \langle \mathsf{get}\, [u,c] \rangle_1\, \langle \mathsf{get}\, [\mathrm{return}_f, c'] \rangle_1\\
& \quad w'' = \textbf{if}\ \text{should inline depending on } f,\ c,\ \langle \mathsf{get}\, [\mathrm{return}_f, c'] \rangle_1, \text{ etc?}\\
& \qquad\quad \textbf{let}\ () = \mathsf{set}\, [\mathrm{entry}_f, c']\, (\bot, \{([u,c], \mathrm{enterFunction}_f)\})\\
& \qquad\quad \textbf{in}\ \{([\mathrm{return}_f, c'], \mathrm{returnFromFunction}_f)\}\\
& \qquad\ \textbf{else}\\
& \qquad\quad \{([u,c], f())\}\\
& \textbf{in}\ (d'', w'')
\end{aligned}
$$

which allows deciding what and what not to inline. Having this decision point in the analysis allows it to not only depend on the function f but also the context c or values of other variables via $\langle \cdot \rangle_1 \circ \mathsf{get}$.

Still, the decision procedure must be carefully designed to preserve the ability to generate violation witnesses from the analysis result. A combination of the following techniques is therefore useful:

- Decide to inline every function call which contains a violation by keeping track of violations within the analysis domain.
- When deciding to not inline a function call which contains a violation, mark the calling node itself as violating.

3.4 Call-string lifting of ARG

The ARG $A_{\text{context}} = (V, S', \delta, [\mathrm{entry}_{\mathrm{main}}, c_{\mathrm{main}}])$ given by Theorem 2 has context-sensitive inlining but not call-site–sensitive inlining (see Section 3.1). We now focus on the latter.

A straightforward solution is to change the contexts of the underlying analysis to already contain partitioning by call site or a sequence of nested call sites, which would make A_{context} also call-site–sensitively inlined without further ado. This is known as the *call-string approach* [7]. The downside is that it requires altering the underlying analysis: the additional partitioning might even improve precision, but that would come at the cost of additional runtime, which is undesired.

The same idea of call strings can be used in ARG states by lifting A_{context} to be call-site sensitive. This is done completely after the analysis and hence has no impact on the behavior of the analysis.

Let V^* be the set of lists with elements from V. Define δ' for all $[u_1, \ldots, u_n] \in V^*$ by:

$$[u_1, \ldots, u_n, u] \xrightarrow[\delta']{s} [u_1, \ldots, u_n, v] \iff u \xrightarrow[\delta]{s} v \tag{5a}$$

$$[u_1, \ldots, u_n, u] \xrightarrow[\delta']{\text{enterFunction}_f} [u_1, \ldots, u_n, u, v] \iff u \xrightarrow[\delta]{\text{enterFunction}_f} v \tag{5b}$$

$$[u_1, \ldots, u_n, u] \xrightarrow[\delta']{\text{returnFromFunction}_f} [u_1, \ldots, u_{n-1}, v] \iff \tag{5c}$$

$$\iff u \xrightarrow[\delta]{\text{returnFromFunction}_f} v \ \wedge \ \langle u_n \rangle_1 \xrightarrow[E]{f()} \langle v \rangle_1 \ \wedge \ \langle u_n \rangle_2 = \langle v \rangle_2$$

where $s \in S$. Then the ARG $A_{\text{callsite}} = (V^*, S', \delta', [[\text{entry}_{\text{main}}, c_{\text{main}}]])$ is call-site–sensitively inlined.

An intuitive reason for the additional constraints in equation (5c) is that u_n from the source state should somehow matter and be used, otherwise there would be no reason to track the call-strings in the first place. The following examples explain this in more detail.

Call-site–sensitive returning. The additional constraint $\langle u_n \rangle_1 \xrightarrow[E]{f()} \langle v \rangle_1$ in (5c) is motivated by the following example, where one function calls another multiple times. Consider again the program from Figure 2b with context-sensitively inlined ARG in Figure 4a. If this condition is omitted from the call-string lifting, then it results in the ARG in Figure 6, which differs from the desired ARG in Figure 4b. The function body is correctly duplicated and the $\text{enterFunction}_{\text{foo}}$ transitions are also correct, but there are excessive $\text{returnFromFunction}_{\text{foo}}$ transitions to wrong return-sites, which is exactly what this constraint forbids to get the desired ARG.

Context-sensitive returning. The additional constraint $\langle u_n \rangle_2 = \langle v \rangle_2$ in (5c) is motivated by the following example, where one function in different contexts calls another in the same context. Consider the program from Figure 7a. If this condition is omitted from the call-string lifting, then it results in the ARG in Figure 7b, which differs from the desired ARG in Figure 7c. Again, the function bodies are correctly duplicated and the $\text{enterFunction}_{\text{foo}}$, $\text{enterFunction}_{\text{bar}}$ and $\text{returnFromFunction}_{\text{foo}}$ transitions are also correct, but there are excessive $\text{returnFromFunction}_{\text{bar}}$ transitions to wrong return contexts, which is exactly what this constraint forbids to get the desired ARG.

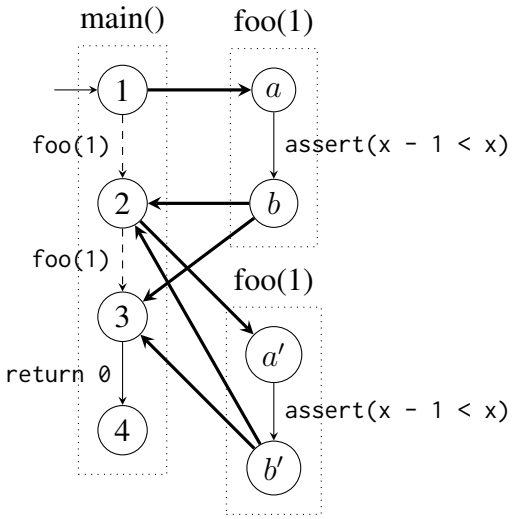
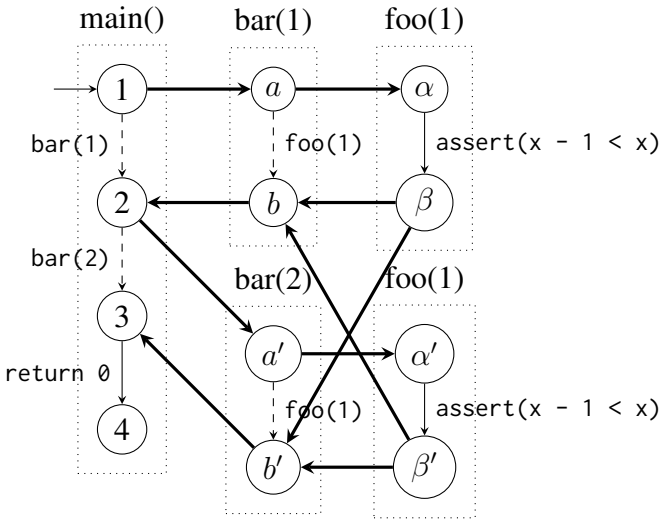


Figure 6. Resulting ARG without call-site–sensitive returning in call-site sensitive inlining of program from Figure 2b.

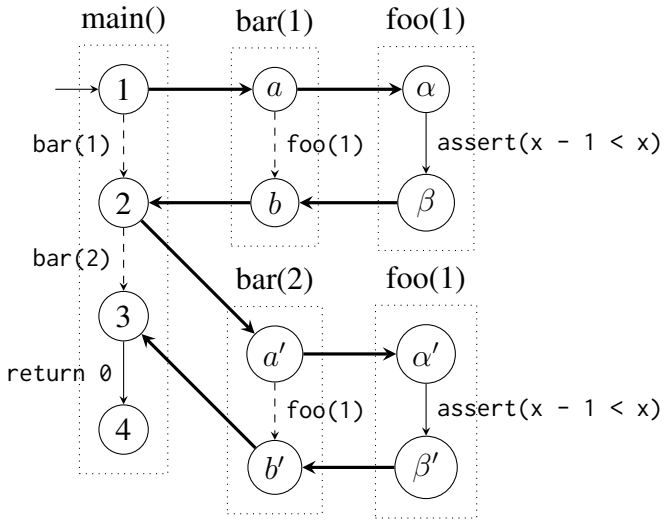
```c
void foo(int x)
{
    assert(x - 1 < x);
}

void bar(int y)
{
    foo(1);
}

int main()
{
    bar(1);
    bar(2);
    return 0;
}
```

(a) C source code.



(b) Resulting ARG without context-sensitive returning.

Figure 7. Context-sensitive returning in call-site–sensitive inlining of function calls for ARG.



(c) Desired ARG with context-sensitive returning.

Figure 7 (cont.). Context-sensitive returning in call-site–sensitive inlining of function calls for ARG.

The following theorem proves that the call-string lifting has both call-site–sensitive and context-sensitive returning, avoiding the issues described above. Therefore, as A_{context} is context-sensitively inlined, A_{callsite} is both context-sensitively and call-site–sensitively inlined.

Theorem 3. *Assume that in E every function call transition has a unique target,*[2] *i.e.,*

$$u \xrightarrow[E]{f()} v \ \wedge \ u \xrightarrow[E]{f()} v' \qquad \textit{implies} \qquad v = v'.$$

Then in A_{callsite}:

1. *Every* enterFunction_f *transition has a unique source.*
2. *Every* $\text{returnFromFunction}_f$ *transition has a unique target.*

Proof.

1. Suppose for any $f, u_1, \ldots, u_n, u, v$ there are x, x' such that

$$x \xrightarrow[\delta']{\text{enterFunction}_f} [u_1, \ldots, u_n, u, v] \xleftarrow[\delta']{\text{enterFunction}_f} x'.$$

   By inversion of (5b), we get $x = [u_1, \ldots, u_n, u] = x'$.

[2] This is true for deterministic programs, which we are considering.

2. Suppose for any $f, u_1, \ldots, u_n, u$ there are x, x' such that

$$x \xleftarrow[\delta']{\text{returnFromFunction}_f} [u_1, \ldots, u_n, u] \xrightarrow[\delta']{\text{returnFromFunction}_f} x'.$$

By inversion of (5c), we have $x = [u_1, \ldots, u_{n-1}, v]$ and $x' = [u_1, \ldots, u_{n-1}, v']$ for some v, v' such that

$$\langle u_n \rangle_1 \xrightarrow[E]{f()} \langle v \rangle_1 \ \wedge \ \langle u_n \rangle_2 = \langle v \rangle_2 \qquad \text{and} \qquad \langle u_n \rangle_1 \xrightarrow[E]{f()} \langle v' \rangle_1 \ \wedge \ \langle u_n \rangle_2 = \langle v' \rangle_2.$$

By assumption $\langle v \rangle_1 = \langle v' \rangle_1$. Moreover $\langle v \rangle_2 = \langle u_n \rangle_2 = \langle v' \rangle_2$. Hence $v = v'$ and thus $x = x'$. ■

3.5 Implementation

In Goblint, all the necessary steps for producing witnesses were implemented:

1. witness analysis (see Section 3.3),
2. call-string lifting of ARG (see Section 3.4),
3. un-CIL transformations (see below),
4. conversion of ARG to witness automaton (see Section 2.2),
5. conversion of witness automaton to SV-COMP witness exchange format (see Section 2.3).

For source code and usage, see Appendix I.

In correctness witnesses, node invariants are derived from the result of value analysis. Depending on configuration, the invariants are simple equalities, inequalities and intervals. In violation witnesses, edge assumptions cannot be directly derived from the analysis result, which is an overapproximation.

In Goblint, the abstract operations of the analysis were given direct access to the argument variables, not just their corresponding values. As a result, the extended constraint system (3) of witness analysis was implemented by defining a lifted data-flow analysis (as per Definition 5) over the abstract domain $\overline{\mathbb{D}}$. Then the lifted analysis is used with the original constraint system (2) to achieve equivalent behavior.

Un-CIL transformations. Goblint uses the CIL library [14, 15] as front-end for the C programming language. The library performs program transformations which simplify the handling of complex C semantics in Goblint itself. In particular, short-circuiting logic operators and ternary operator are rewritten using if statements and auxiliary variables. These transformations need to be reversed for constructing the witness to avoid introducing fictitious `control` edges (see Table 1a), which belong to if statements that are not present in the analyzed program but were introduced by CIL.

Functional ARG. Let $A = (Q, \Sigma, \delta, q_0)$ be an ARG. In practice, a convenient representation for the transition relation $\delta \subseteq Q \times \Sigma \times Q$ is a successor function $\vec{\delta} : Q \to \mathcal{P}(\Sigma \times Q)$ such that

$$\vec{\delta}\, u \ni (\sigma, v) \quad \Longleftrightarrow \quad u \xrightarrow{\sigma} v,$$

i.e.,

$$\vec{\delta}\, u = \{(\sigma, v) \mid u \xrightarrow{\sigma} v\}.$$

The ARG A_{context} in Theorem 2 is constructed from σ', which represents the transition relation as a predecessor function instead. This is simply because the analysis and constraint system under consideration correspond to forward analysis. Defining $\vec{\delta}$ from σ' corresponds to inverting the (labeled) relation between constraint system variables. Since both are finite, this is done purely mechanically.

Transformations of ARGs, like call-string lifting and un-CIL transformation, are even more straightforward in the successor function representation: the new ARG transition successor function $\vec{\delta'}$ is defined through $\vec{\delta}$. It has the benefit of being *lazy*: the transformations are calculated on-demand only once the final ARG is traversed as a graph for output. Actually, this is also necessary because formally A_{callsite} has an infinite state set and transition relation.

3.6 Evaluation

The implementation was evaluated manually during design and development. Furthermore, it was tested with one category of SV-COMP verification tasks in the SV-COMP environment.

Manual testing. Small C programs were crafted[3] and the implementation evaluated by manually inspecting the generated SV-COMP witnesses. These show that the methodology and implementation successfully perform context- and call-site–sensitive inlining while generating witnesses.

SV-COMP testing. The SV-COMP competition environment was set up using the official BenchExec framework [3, 16] to run Goblint on actual SV-COMP verification tasks, validate the witnesses and aggregate the results. CPAchecker [17, 18] and Ultimate Automizer [19, 20] were used as witness validators.

Since the existing analyses of Goblint have been aimed at Linux device driver verification, and support pointer aliases and function pointers [4], the "SoftwareSystems-DeviceDriversLinux64-ReachSafety" category of SV-COMP was chosen for evaluation. Official SV-COMP resource limitations were used, except the time limit of 15 minutes

[3] These can be found in `./tests/sv-comp/cfg/` subdirectory in the repository, see Appendix I.

was decreased to 30 seconds due to the sheer number of verification tasks in the category (2729).

Summarized results are given in the second column (ARG-based witnesses) of Table 2. The vast majority of correctness witnesses are successfully validated by at least one of the two validators. There is a notable number of correct programs which Goblint finds to contain a violation; that is unrelated to witness generation, but shows the amount of false positive violations. Moreover, violation witnesses for incorrect programs were not confirmed by either validator, showing that the violation witnesses are not precise enough.

To measure the usefulness of generated ARG-based correctness witnesses, single-state trivial witnesses were tested with instead. This is nearly equivalent to asking the witness validators to verify the program as they would completely on their own. Summarized results are given in the third column (trivial witnesses) of Table 2. The greater number of validated witnesses means that the generated ARG-based correctness witnesses do not contain sufficiently useful invariants for the validators and instead hinder witness validation with their large number of states [9].

Conclusion. Manual testing confirmed that the witnesses are based on correctly context- and call-site–sensitively inlined ARGs. SV-COMP testing revealed that the implementation works correctly in the SV-COMP environment, but there are many false positive violations and the generated witnesses are not useful for witness validators as they are too imprecise.

Table 2. Summarized Goblint results in "SoftwareSystems-DeviceDriversLinux64-ReachSafety" category of SV-COMP, comparing ARG-based witnesses to trivial ones.

	Number of verification tasks	
Result	ARG-based witnesses	Trivial witnesses
Right verdict, validated witness	741	791
Correct program	741	791
Incorrect program	0	0
Right verdict, unconfirmed witness	83	35
Correct program	75	27
Incorrect program	8	8
Wrong verdict	54	54
Correct program	52	52
Incorrect program	2	2
Missing verdict	1851	1849
Timeout	1804	1803
Crash	47	46

4 Violation witnesses

Sound data-flow analysis overapproximates the runtime behavior of a program. This has two major downsides for witness generation:

1. Violation witnesses are not precise enough to be useful because edge assumptions cannot be derived from the analysis result.

2. There are many false positive violations (correct programs, which are found to be incorrect) because the analysis is not precise enough to rule out infeasible violations.

Both issues could be remedied by designing more precise data-flow analyses with more sophisticated abstract domains. However, such analyses would also decrease the performance.

In this section, we seek to mitigate the two issues without modifying the original analysis by adapting the approach of Junker et al. [21] from model-checking to data-flow analysis. Notably, we show how observer automata can be used to refine data-flow analysis.

Algorithm. The approach means using data-flow analysis as one part of the following refinement loop, which is also illustrated in Figure 8:

1. **Run data-flow analysis**.
2. If no violation was reached, **output correctness witness and halt**.
3. If a violation was reached, find a path from the violation to the entry node of the program.
4. **Run feasibility analysis on the path**.
5. If the path was feasible, **output violation witness and halt**.
6. If the path was infeasible, find its minimal infeasible subpath.
7. Construct an observer automaton for the minimal infeasible subpath.
8. Refine the data-flow analysis using the observer automaton.
9. Go to step 1.

The following subsections explain key steps of the algorithm in more detail.

Violation path search. After the data-flow analysis, a corresponding ARG, which contains a violation, is constructed. Any method for finding a backward path from a violation node to the entry node of the program works for our purpose. For example, breadth-first search (BFS) can be used to find a path with the minimum number of edges [11].

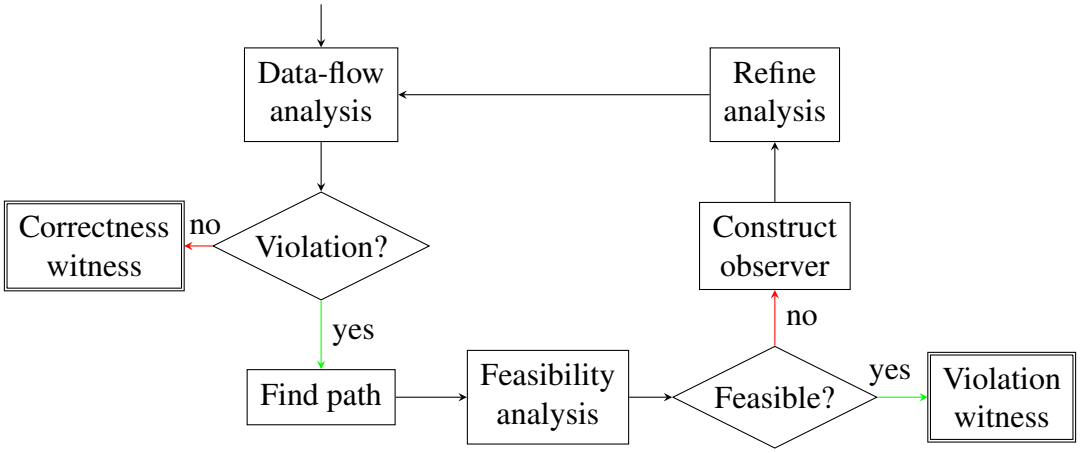


Figure 8. Refinement loop algorithm.

4.1 Feasibility analysis

The feasibility of a path can be checked by first constructing the *weakest precondition* (WP) formula for the sequence of statements of the path [2, 22]. If the path is found from the interprocedural ARG, then function calls are inlined and thus WP only needs to support other primitive statements, like assignments and conditional guards. The satisfiability of the WP formula, which corresponds to the feasibility of the path, can then be directly checked using an SMT solver.

If the path turns out to be feasible, then the SMT solver supplies a *model*, i.e., a variable assignment to free variables of the WP formula, which satisfies the formula. This model can be used for deriving edge assumptions for a violation witness.

If the path turns out to be infeasible, then its minimal infeasible subpath can be identified from the *minimal unsatisfiable core* of the unsatisfiable WP formula [21]. This requires the WP formula to be constructed in the form of labeled conjuncts, such that each statement in the path corresponds to one labeled conjunct [22]. This is needed as the minimal unsatisfiable core is returned as the set of conjunct labels.

Many SMT solvers can work incrementally, i.e., conjuncts can be asserted one by one and satisfiability checked after each assertion efficiently. Notably, this allows combining the search for violation path and its feasibility analysis in the following way. As violation paths are searched their assertions are incrementally added and checked simultaneously. If infeasibility is found early, the path search can terminate early without constructing a path all the way to the entry node of the program.

Alternative less general methods can also be used for feasibility analysis, e.g., interval equation systems [23].

4.2 Observer automata

Let $E' = L \times S' \times L$. In addition to program statement transitions, this contains $\mathrm{enterFunction}_f$ and $\mathrm{returnFromFunction}_f$ statement transitions introduced by inlining. We write $(u, s, v) \in E'$ as $u \xrightarrow[E']{s} v$. A sequence $l_0 \xrightarrow{s_1} l_1 \xrightarrow{s_2} \cdots \xrightarrow{s_{n-1}} l_{n-1} \xrightarrow{s_n} l_n$ is called an *interprocedural program path* if function calls are inlined. For a formal definition of interprocedurally valid program paths see Reps et al. [24].

Definition 8. An **observer automaton** $O = (Q, \omega, q_0, F)$ is a finite automaton with:

- the finite set Q of states,
- the alphabet E',
- the transition relation $\omega \subseteq Q \times E' \times Q$,
- the initial state $q_0 \in Q$,
- the set $F \subseteq Q$ of accepting states.

We write $(q, (u, s, v), q') \in \omega$ as $q \xrightarrow[\omega]{(u,s,v)} q'$. A sequence $q_0 \xrightarrow[\omega]{(l_0,s_1,l_1)} \cdots \xrightarrow[\omega]{(l_{n-1},s_n,l_n)} q_n$ is a *simulation sequence* of an interprocedural program path $l_0 \xrightarrow[E']{s_1} \cdots \xrightarrow[E']{s_n} l_n$ if every step $l_{i-1} \xrightarrow[E']{s_i} l_i$ corresponds to a step $q_{i-1} \xrightarrow[\omega]{(l_{i-1},s_i,l_i)} q_i$. The simulation sequence is accepted if $q_n \in F$ and the program path is accepted if there exists an accepting simulation sequence for it.

Construction from program subpath. Let $l_1 \xrightarrow{s_2} \cdots \xrightarrow{s_n} l_n$ be a program subpath. It is possible to construct an observer automaton $O = (Q, \omega, q_0, F)$, which accepts a program path if and only if it contains the given program subpath [21], such that:

- $Q = \{0, \ldots, n-1\}$,
- $q_0 = 0$,
- $F = \{n-1\}$,
- ω is constructed so that during simulation the observer automaton state q always corresponds to how many transitions of the program subpath prefix have been taken in sequence.

This automaton will be used to step-by-step observe program execution along a subpath during the analysis.

4.3 Observer analysis

Given an observer automaton $O = (Q, \omega, q_0, F)$, corresponding to an infeasible subpath to be excluded, we define an observer analysis which refines the previous data-flow analysis.

Define the relation ω_F by

$$q \xrightarrow[\omega_F]{e} q' \quad \Longleftrightarrow \quad q \xrightarrow[\omega]{e} q' \wedge q' \notin F.$$

Consider the following system of constraints with variables $V \times Q$ over $\mathbb{D}$:

$$\text{If } q_0 \notin F \tag{6a}$$
$$([\text{entry}_{\text{main}}, c_{\text{main}}], q_0) \sqsupseteq d_{\text{main}}$$
$$\forall u \xrightarrow[E]{s} v \quad \forall q \xrightarrow[\omega_F]{(u,s,v)} q' \tag{6b}$$
$$([v, c], q') \sqsupseteq [\![s]\!]^\sharp\,(\mathsf{get}\,([u, c], q))$$
$$\forall u \xrightarrow[E]{f()} v \quad \forall q \xrightarrow[\omega_F]{(u,\text{enterFunction}_f,\text{entry}_f)} q' \quad \forall q''' \in Q \tag{6c}$$
$$([v, c], q''') \sqsupseteq \textbf{let}\,(c', d') = \mathsf{enter}^\sharp\,(\mathsf{get}\,([u, c], q))$$
$$() = \mathsf{set}\,([\text{entry}_f, c'], q')\,d'$$
$$\textbf{in}\ \bot$$
$$\forall u \xrightarrow[E]{f()} v \quad \forall q \xrightarrow[\omega_F]{(u,\text{enterFunction}_f,\text{entry}_f)} q' \quad \forall q'' \xrightarrow[\omega_F]{(\text{return}_f,\text{returnFromFunction}_f,v)} q''' \tag{6d}$$
$$([v, c], q''') \sqsupseteq \textbf{let}\,(c', d') = \mathsf{enter}^\sharp\,(\mathsf{get}\,([u, c], q))$$
$$\textbf{in}\ \mathsf{combine}^\sharp\,(\mathsf{get}\,([u, c], q))\,(\mathsf{get}\,([\text{return}_f, c'], q''))$$

Compared to the constraint system (2) of the original analysis, all transitions are constrained through ω_F to be such that they don't lead into an accepting state of the observer automaton. Moreover, the single constraint for function calls (2c) is split into two constraints: one for function call entry (6c) and one for function call return (6d). This is to allow analyzing functions which the observer allows entering but prevents from reaching return_f.

The new system of constraints performs the same analysis but with increased precision thanks to the following:

1. It excludes analysis along infeasible paths forbidden by the observer.
2. It partitions the analysis at non-accepting observer states, i.e., along prefixes of the infeasible subpath of the observer. This makes the data-flow analysis partially path-sensitive at those locations.

The first property, which is proven by the following theorem, is crucial for refinement, because if the infeasible paths were not excluded, then the subsequent refinement loop iteration could find the same infeasible path again, leading to obvious non-termination.

Theorem 4 (Observer exclusion)**.** *Let $\sigma : V \times Q \to \mathbb{D}$ be the* least *solution of the refined system* (6)*. Then for all $x \in V$ and $q_f \in F$, we have $\sigma\,(x, q_f) = \bot$.*

Proof. Let $x \in V$ and $q_f \in F$. Check each constraint for a lower bound for $\sigma\,(x, q_f)$:

(6a) Does not apply since $q_0 \notin F$.
(6b) Does not apply since $q' \notin F$ through ω_F.
(6c) Applies but gives a trivial lower bound $\bot$. The side effect does not apply since $q' \notin F$ through ω_F.
(6d) Does not apply since $q''' \notin F$ through ω_F.

Thus, the constraint system only specifies $\sigma\,(x, q_f) \sqsupseteq \bot$. Since σ is the least solution, $\sigma\,(x, q_f) = \bot$. ■

On the other hand, the observer analysis should *only* exclude infeasible paths. Excluding any feasible path would make the analysis unsound. Therefore, we claim the following without further details or proof.

Theorem 5 (Soundness)**.** *Let $\sigma : V \times Q \to \mathbb{D}$ be a solution of the refined system* (6)*. Let $\sigma^* : V \to \mathbb{D}$ be the* merge over all (interprocedurally valid) paths *(MOP) solution [7] of the analysis.*

Then for each $x \in V$

$$\bigsqcup_{q \in Q} \sigma\,(x, q) \sqsupseteq \sigma^*\, x.$$

Intuitively, the partitions at x account for all interprocedurally valid feasible paths which reach x.

4.4 Examples

The following examples demonstrate how violation witnesses benefit from feasibility analysis and correctness witnesses benefit from refinement.

As opposed to assignment statements, conditional guard statements are surrounded by square brackets.

Incorrect program. Consider the program from Figure 9a, which has been adapted from Beyer et al. [8] by modeling nondeterministic values through uninitialized variables. Figure 9b shows its partial CFA, which also happens to be the ARG, constructed after initial data-flow analysis with interval analysis, as the program is intraprocedural. The violation node is marked in red and the sink nodes in gray.

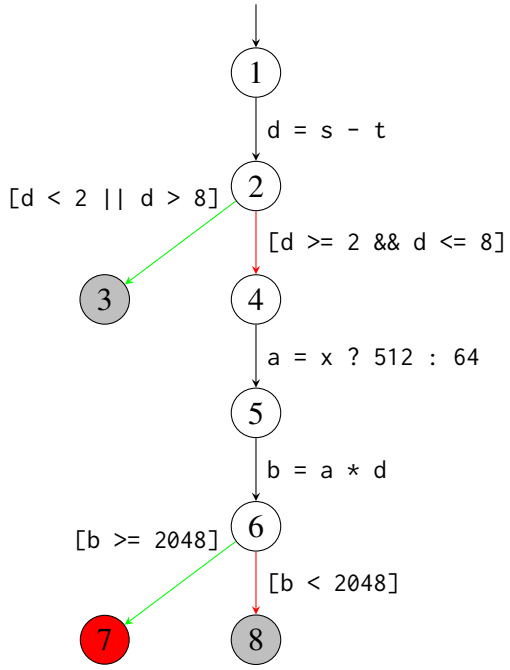
The following violation path candidate can be constructed:

$$1 \xrightarrow{\texttt{d = s - t}} 2 \xrightarrow{\texttt{[d >= 2 \&\& d <= 8]}} 4 \xrightarrow{\texttt{a = x ? 512 : 64}} 5 \xrightarrow{\texttt{b = a * d}} 6 \xrightarrow{\texttt{[b >= 2048]}} 7.$$

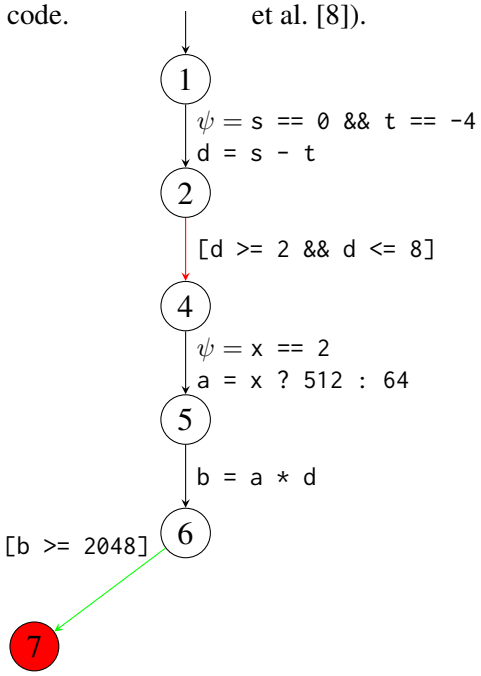
A feasibility analysis reveals this path is feasible with, for example, `s == 0`, `t == -4` and `x == 2`. Thus, a more precise ARG, shown in Figure 9c, can be constructed as a violation witness by adding edge assumptions based on the feasible model.

```c
int main()
{
    int s, t; // nondet_int
    int d = s - t;
    if (d < 2 || d > 8)
        return 0;
    int x; // nondet_int
    int a = x ? 512 : 64;
    int b = a * d;
    if (b >= 2048)
        error();
    if (b < 128)
        error();
    while (a > 0)
        a--;
    return 0;
}
```

(a) C source code.



(b) Partial CFA and ARG with sink nodes (analogous to witness 2 in Figure 3c of Beyer et al. [8]).



(c) ARG with edge assumptions (analogous to witness 3 in Figure 3d of Beyer et al. [8]).

Figure 9. Incorrect program which benefits from the feasible model, adapted from Beyer et al. [8].

Correct program. Consider the program from Figure 10a, which is intended for use-after-free memory analysis. We adapt the example to error function reachability analysis, as shown in Figure 10b, by modeling dynamic memory management through a boolean variable p, which indicates whether the memory is allocated or freed. Figure 10c shows its CFA, which also happens to be the ARG, constructed after initial data-flow analysis, as the program is intraprocedural. The violation node is marked in red.

A possible run of the refinement loop might construct the observer automata shown in Figure 11 as follows:

1. Initially, the following violation path candidate might be constructed:

   $$1 \xrightarrow{\texttt{p = 1}} 2 \xrightarrow{\texttt{x = 10}} 3 \xrightarrow{\texttt{[x > 0]}} 4 \xrightarrow{\texttt{[p != 1]}} 5.$$

   A feasibility analysis reveals this path is infeasible as `p = 1` and `[p != 1]` are contradictory. Thus, the corresponding observer, shown in Figure 11a, is constructed and used to refine the data-flow analysis, which still reaches a violation.

2. Then, the following violation path candidate might be constructed:

   $$4 \xrightarrow{\texttt{[p = 1]}} 6 \xrightarrow{\texttt{[x != 1]}} 8 \xrightarrow{\texttt{x = x - 1}} 3 \xrightarrow{\texttt{[x > 0]}} 4 \xrightarrow{\texttt{[p != 1]}} 5.$$

   A feasibility analysis reveals this path is infeasible as `[p = 1]` and `[p != 1]` are contradictory. Thus, the corresponding observer, shown in Figure 11b, is constructed and used to refine the data-flow analysis, which still reaches a violation.

3. Then, the following violation path candidate might be constructed:

   $$6 \xrightarrow{\texttt{[x = 1]}} 7 \xrightarrow{\texttt{p = 0}} 8 \xrightarrow{\texttt{x = x - 1}} 3 \xrightarrow{\texttt{[x > 0]}} 4 \xrightarrow{\texttt{[p != 1]}} 5.$$

   A feasibility analysis reveals this path is infeasible as `[x = 1]`, `x = x - 1` and `[x > 0]` in sequence are contradictory. Note that this infeasible path is not minimal. The minimal infeasible subpath excludes the last step, which is not part of the contradiction. Thus, the corresponding observer, shown in Figure 11c, is constructed and used to refine the data-flow analysis, which doesn't reach a violation.

Thus, the program is proved correct.

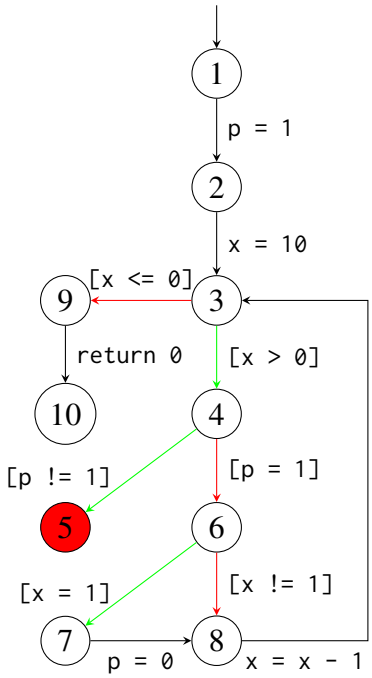
Note that in this case it is actually sufficient to only refine the data-flow analysis with the third observer, shown in Figure 11c, to prove the program correct. However, since it is not constructed from the simplest infeasible subpath, it might not be the first one to be checked for feasibility.

```c
int main()
{
    int x, *a;
    int *p = malloc(sizeof(int));
    for (x = 10; x > 0; x--)
    {
        a = p; // use
        if (x == 1)
            free(p);
    }
    return 0;
}
```

(a) Original C source code.

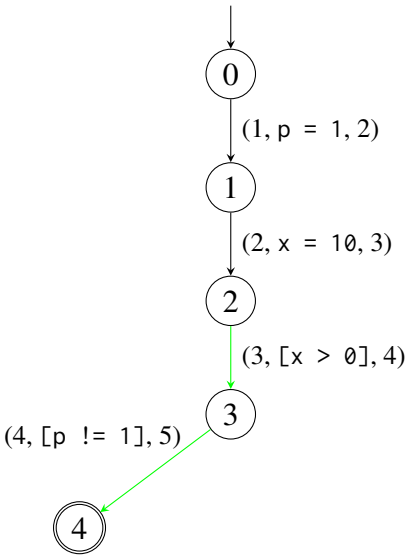
```c
int main()
{
    int x;
    int p = 1; // malloc
    for (x = 10; x > 0; x--)
    {
        if (p != 1) // use
            error();
        if (x == 1)
            p = 0; // free
    }
    return 0;
}
```

(b) Adapted C source code.

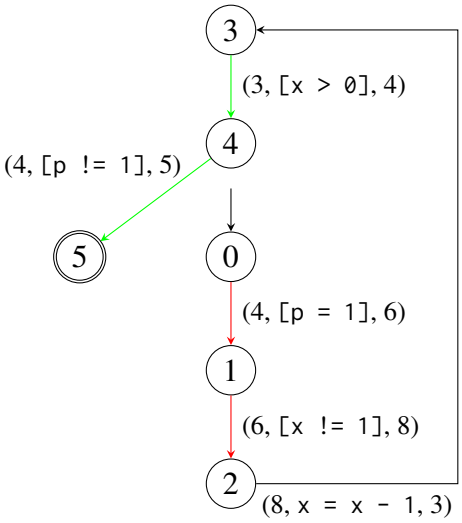


(c) CFA and initial ARG.

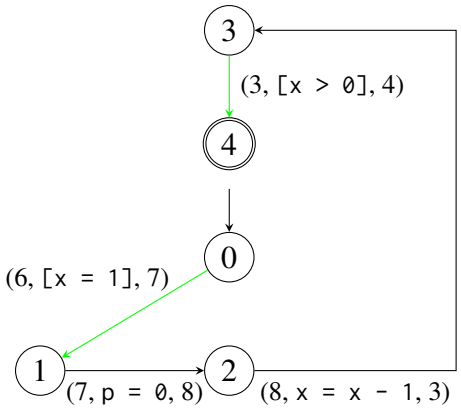
Figure 10. Correct program which requires refinement to verify, adapted from Junker et al. [21].



(a) Observer automaton corresponding to the first iteration of the loop.



(b) Observer automaton corresponding to an intermediate iteration of the loop.



(c) Observer automaton corresponding to the last iteration of the loop (analogous to infeasible subpath in Figure 4 of Junker et al. [21]).

Figure 11. Possible observer automata for infeasible violation paths of the program from Figure 10.

4.5 Implementation

In Goblint, prototypes of all the refinement loop steps were implemented to ensure that the above theory works in practice. The fully automated refinement loop was not implemented because Goblint's current codebase is not well-suited for repeatedly running the data-flow analysis. For source code and usage, see Appendix I.

Feasibility analysis. The Goblint implementation uses the Z3 SMT solver [25]. Since comprehensive WP modeling the semantics of C is beyond the scope of this thesis, only a small integer-valued subset of C is currently supported. Still, this is sufficient for examples from Section 4.4. More complete handling has been presented by Junker [22].

Observer automata. The observer automaton transition relation construction from Junker et al. [21] yields a non-deterministic automaton, which is undesirable for ease of implementation and efficiency. Hence, the Goblint implementation instead uses the Knuth-Morris-Pratt string-matching algorithm [11], which gives an automaton with the following properties:

- It is deterministic, which allows using a *transition function* of the type $Q \times E \to Q$ instead of the transition relation ω.
- It has a compact representation in the form of a prefix function (array) consisting of just $|Q| = n$ elements.
- It also has just $\Theta(n)$ preprocessing time.

Observer analysis. In Goblint, the constraint system (6) is equivalently implemented as a system of constraints with variables V over the abstract domain $Q \to \mathbb{D}$, which makes the partitioning by Q explicit and allows using the original constraint system (2). Intuitively, this equivalence is due to the solution types being isomorphic through currying:

$$V \times Q \to \mathbb{D} \;\cong\; V \to (Q \to \mathbb{D}).$$

To be more specific, the partitioning by Q is implemented using Goblint's general and customizable path-sensitive analysis lifting [26], which uses a Hoare powerset domain [27].

Moving the observer state partitioning from constraint system variables to values comes with a complication. The witness analysis (see Section 3.3), which only records variable dependencies, fails to record observer state dependencies, which are necessary for the observer analysis to exclude infeasible subpaths. This requires modifying the witness analysis and the following ARG construction to make them treat each partition of a variable also as an independent ARG state. In Goblint, this required dependency-tracking extensions to the Hoare powerset domain; their interaction has not been studied in theory.

4.6 Evaluation

The implementation was evaluated manually with the examples presented in Section 4.4.[4]

For the program from Figure 9a the data-flow analysis with interval analysis alone is able to generate the witness shown in Figure 9b. The implemented feasibility analysis is able to generate the more precise witness shown in Figure 9c. Therefore, the feasibility analysis implementation successfully helps to generate more precise violation witnesses.

For the program from Figure 10b the data-flow analysis alone is unable to prove the program correct, producing a false positive violation. The implemented observer analysis using the observer from Figure 11c is able to refine the data-flow analysis and prove the program correct. Therefore, the observer analysis implementation successfully helps to eliminate false positives.

[4]These can also be found in `./tests/sv-comp/false/` and `./tests/sv-comp/observer/` subdirectories in the repository, see Appendix I.

5 Conclusion

A formal overview of automata as witnesses for specification-violating and correct programs was given. Followed by description of SV-COMP witness exchange format.

Issues of producing witnesses for interprocedural programs were discussed. These were solved for data-flow analysis without modifications to the underlying analysis itself. Context-sensitive inlining was achieved by extending the constraint system to keep track of additional dependency information to construct abstract reachability graphs. Call-site–sensitive inlining was achieved by further postprocessing of these. Properties of both were proven.

Limitations of abstract reachability graph based violation witnesses were identified. These were solved by adapting a refinement loop approach to data-flow analysis, again without having to modify the underlying analysis.

All the proposed solutions were implemented in Goblint, showing their applicability to a data-flow analyzer. Evaluation of witnesses generated by Goblint shows that interprocedural witnesses can be extracted from data-flow analysis results, despite its abstraction of function calls. Moreover, violation witness false positives can be reduced by augmenting data-flow analysis with additional feasibility analysis. The latter also allows generating more precise violation witnesses.

Future work. There are numerous theoretical and practical limitations, questions and directions which deserve further study:

1. Recursive functions have statically unbounded call stacks, which is an issue for the naïve call-string lifting and yields an infinite ARG. Additional techniques are required to handle recursion inlining.

2. Goblint still needs a fully automatic implementation of the refinement loop to use the violation-related approaches in SV-COMP. Moreover, its weakest precondition must be extended to handle most of the C programming language.

3. In general, the refinement loop may fail to terminate. For example, if contradictions do not arise in a single program loop iteration, then the refinement loop may end up unrolling the program loop iteration-by-iteration via observers. Heuristics might be necessary to avoid nontermination in such hopeless situations.

4. In the implementation of observer analysis, Goblint required dependency-tracking extensions to the Hoare powerset domain used in its path-sensitive analysis lifting. These need to be better understood through formalization.

5. The refinement loop re-runs data-flow analysis from scratch every time, although the addition of an observer might only influence a small part of the program's

analysis. We conjecture that this inefficiency can be avoided either by using constraint system side effects to modify observer-affected variables on the go or through other incremental analysis methods.

6. SV-COMP has a category for multi-threaded programs and its witness exchange format supports violation witnesses for them [1, 10, 28]. Firstly, a correctness witness format for concurrent programs is still to be designed. Secondly, as Goblint is an analyzer for multi-threaded programs, its witness generation needs to be expanded to produce concurrent witnesses.

7. As observer automata can refine data-flow analysis, the same could be done with witness automata. This might allow Goblint to also consume witnesses to increase its precision and validate them [8, 9].

References

[1] SV-COMP 2020. 9th Competition on Software Verification. 2020. https://sv-comp.sosy-lab.org/2020 (2020-05-15).

[2] Huth M. and Ryan M. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, 2004.

[3] Beyer D. Advances in Automatic Software Verification: SV-COMP 2020. *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Biere A. and Parker D. Cham: Springer International Publishing, 2020, pp. 347–367.

[4] Vojdani V., Apinis K., Rõtov V., Seidl H., Vene V., and Vogler R. Static race detection for device drivers: the Goblint approach. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. ACM, 2016, pp. 391–402. https://goblint.in.tum.de (2020-05-15).

[5] Goblint. Source code. https://github.com/goblint/analyzer (2020-05-15).

[6] Apinis K., Seidl H., and Vojdani V. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. *Programming Languages and Systems*. Ed. by Jhala R. and Igarashi A. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 157–172. https://goblint.in.tum.de/papers/side.pdf (2020-05-15).

[7] Seidl H., Wilhelm R., and Hack S. Compiler Design: Analysis and Transformation. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[8] Beyer D., Dangl M., Dietsch D., Heizmann M., and Stahlbauer A. Witness Validation and Stepwise Testification across Software Verifiers. *Proceedings of the 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE 2015, Bergamo, Italy, August 31 - September 4)*. Ed. by E. Di Nitto, M. Harman, and P. Heymans. ACM, New York, 2015, pp. 721–733.

[9] Beyer D., Dangl M., Dietsch D., and Heizmann M. Correctness Witnesses: Exchanging Verification Results Between Verifiers. *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016, Seattle, WA, USA, November 13-18)*. Ed. by T. Zimmermann, J. Cleland-Huang, and Z. Su. ACM, 2016, pp. 326–337.

[10] SoSy-Lab. An Exchange Format for Verification Witnesses. https://github.com/sosy-lab/sv-witnesses (2020-05-15).

[11] Cormen T. H., Leiserson C. E., Rivest R. L., and Stein C. Introduction to Algorithms. 3rd ed. The MIT Press, 2009.

[12] Beyer D., Gulwani S., and Schmidt D. Combining Model Checking and Data-Flow Analysis. *Handbook on Model Checking*. Ed. by E. M. Clarke, T. A. Henzinger, and H. Veith. Springer, 2018, pp. 493–540.

[13] Beyer D., Dangl M., and Wendler P. A Unifying View on SMT-Based Software Verification. *Journal of Automated Reasoning* 60.3 (2018), pp. 299–335.

[14] Necula G. C., McPeak S., Rahul S. P., and Weimer W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *Compiler Construction*. Ed. by Horspool R. N. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 213–228.

[15] C Intermediate Language (CIL). Source code. `https://github.com/cil-project/cil` (2020-05-15).

[16] SoSy-Lab. BenchExec. A Framework for Reliable Benchmarking and Resource Measurement. `https://github.com/sosy-lab/benchexec` (2020-05-15).

[17] Beyer D. and Keremoglu M. E. CPAchecker: A Tool for Configurable Software Verification. *Computer Aided Verification*. Ed. by Gopalakrishnan G. and Qadeer S. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190.

[18] SoSy-Lab. CPAchecker. The Configurable Software-Verification Platform. `https://cpachecker.sosy-lab.org` (2020-05-15).

[19] Heizmann M., Hoenicke J., and Podelski A. Software Model Checking for People Who Love Automata. *Computer Aided Verification*. Ed. by Sharygina N. and Veith H. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 36–52.

[20] Ultimate Automizer. `https://ultimate.informatik.uni-freiburg.de/automizer` (2020-05-15).

[21] Junker M., Huuck R., Fehnker A., and Knapp A. SMT-Based False Positive Elimination in Static Program Analysis. *Formal Methods and Software Engineering*. Ed. by Aoki T. and Taguchi K. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 316–331.

[22] Junker M. Using SMT Solvers for False-Positive Elimination in Static Program Analysis. Master's thesis. University of Augsburg, 2010. `https://web.archive.org/web/20170830043936/http://www4.in.tum.de/~junkerm/publications/thesis.pdf` (2020-05-15).

[23] Fehnker A., Huuck R., and Seefried S. Counterexample Guided Path Reduction for Static Program Analysis. *Concurrency, Compositionality, and Correctness: Essays in Honor of Willem-Paul de Roever*. Ed. by Dams D., Hannemann U., and Steffen M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 322–341.

[24] Reps T., Horwitz S., and Sagiv M. Precise interprocedural dataflow analysis via graph reachability. *POPL'95*. 1995, pp. 49–61.

[25] Moura L. de and Bjørner N. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Ramakrishnan C. R. and Rehof J. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[26] Apinis K. Frameworks for analyzing multi-threaded C. PhD thesis. Technische Universität München, 2014. `http://www2.in.tum.de/bib/files/apinis14diss.pdf` (2020-05-15).

[27] Miné A. Abstract Interpretation IV. Semantics and Application to Program Verification. Slides. Paris: École normale supérieure, 2016-05-27. `https://www.di.ens.fr/~rival/semverif-2017/sem-13-ai.pdf` (2020-05-15).

[28] Beyer D. and Friedberger K. A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker. *Proceedings of the 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2016, Telč, Czechia, October 21-23)*. Ed. by J. Bouda, L. Holík, J. Kofroň, J. Strejček, and A. Rambousek. EPTCS 233. ArXiV, 2016, pp. 61–71.

Appendices

I Goblint implementation

All aforementioned Goblint implementations in OCaml can be found in the following GitHub fork repository:

`https://github.com/sim642/goblint`.

The author's changes with their version control history are under the Git tag "msc-thesis", which can also be browsed online:

`https://github.com/sim642/goblint/tree/msc-thesis`.

Earlier parts of the author's contributions have already been merged to the official Goblint source code repository [5].

Usage

The following command line flags were added to control witness generation:

`--enable ana.sv-comp` Enables SV-COMP mode:

- adds support for SV-COMP functions (see Section 2.3),
- outputs the analysis verdict of error function unreachability to standard output stream,
- generates the corresponding witness in SV-COMP witness exchange format to the file `witness.graphml` in the current directory.

`--sets exp.witness_path witness.graphml` Sets the path for generated SV-COMP witness (only necessary to change the default).

`--enable exp.uncilwitness` Enables un-CIL transformations in witness generation (see Section 3.5).

`--enable exp.minwitness` Enables *experimental* support for minimizing witnesses by skipping nodes and edges without useful data.

`--enable ana.wp` Enables *experimental* weakest precondition feasibility analysis for violations.

Additional details and examples can be found in `README.SV-COMP.md` of the repository.

Observer analysis can be run by hard-coding the infeasible subpath in the module `ObserverAnalysis` of Goblint source code. Then adding the following command line flags enables it:

```
--sets ana.activated[+] 'observer' --sets ana.path_sens[+] 'observer'.
```

II Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Simmo Saan**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   "**Witness Generation for Data-flow Analysis**",

   supervised by Vesal Vojdani.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Simmo Saan
15.05.2020