TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Verifying Regular Safety Properties of C Programs Using the Static Analyzer Goblint

Ralf Vogler

TUM-I142

Acknowledgments

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement n° 269335 and from the German Science Foundation (DFG).

Abstract

The thesis starts by introducing basic concepts of program analysis and gives an overview of the program analyzer Goblint, which is developed at the chair using OCaml. Goblint is a static analyzer for multi-threaded C programs focused on data race detection.

The main part describes the development of a specification language which can be used to verify regular safety properties of C programs. The work is based on Goblint as a framework for the analyses. Verification of file handle usage serves as an example for comparing a manual implementation with the developed specification language.

Finally other possible use cases for the specification language and its limitations are examined.

Contents

List of Figures

List of Tables

Listings

1. Introduction

Testing code can be cumbersome and time-consuming. High code coverage lowers the chance that errors may go undetected, but there is no guarantee. This is especially true for multi-threaded programs. So called data races can be very hard to find and to reproduce.

Definition 1.1 (Data race) *Different threads of a program access the same shared memory location, and at least one thread writes to it.*

With the increase in size and complexity of software projects and their source code, it gets more important to automatize the testing process as much as possible. An example of how fast the source lines of code in modern operating systems have been increasing can be seen in tables 1.1 and 1.2. Although the numbers may not be precise, they give an intuition.

Year	Operating System	SLOC (Million)
1993	Windows NT 3.1	4-5
1994	Windows NT 3.5	7-8
1996	Windows NT 4.0	11-12
2000	Windows 2000	>29
2001	Windows XP	45
2003	Windows Server 2003	50

Table 1.1.: Lines of code: Windows [2]

Operating System	SLOC (Million)
Debian 2.2	55-59
Debian 5.0	324
OpenSolaris	9.7
FreeBSS	8.8
Mac OS X 10.4	86
Linux kernel 2.6.0	5.2
Linux kernel 2.6.35	13.5
Linux kernel 3.6	15.9

Table 1.2.: Lines of code: Unix [2]

One way of testing is to execute the program. This is called dynamic analysis. In order to be effective, the program has to be analyzed with enough different test inputs. The goal is to find errors by running the program on inputs that are likely to produce them. The problem with that is, that the absence of errors can not be proven and some could remain undetected. There are many examples for what insufficient testing might lead to; a prominent one is the self-destruction of the Ariane 5 rocket [5].

Goblint does sound, static analysis, which uses abstract interpretation to approximate the semantics of a program. Static means that it is able to approximate the run-time behavior of the program without having to execute it. Sound means that the absence of errors - in contrast to dynamic analysis - *can* be proven. This is especially important for the verification of properties in safety-critical applications like medical software or software for planes, launchers, reactors and so on.

The main use case for Goblint is race detection in multi-threaded C programs. Although C is more and more replaced by higher level languages like Java and C#, it is still one of the most widely used programming languages and is dominant in implementing system

software (e.g. Linux kernel) and in embedded applications. With the rise of multi-core architectures, multi-threading has also become more important.

Goblint has been tested on parts of the Linux kernel and is sufficiently efficient to be used for race-detection of multi-threaded programs up to about 25 thousand lines of code [13, 1].

In addition to XML, JSON and HTML output, there is an Eclipse plug-in that displays the results of the analysis in a view and adds warning markers for places in the code where a data race might occur. A screenshot can be seen in Figure 1.1.

Information on how to setup Goblint can be found in Chapter A or on its homepage[1].

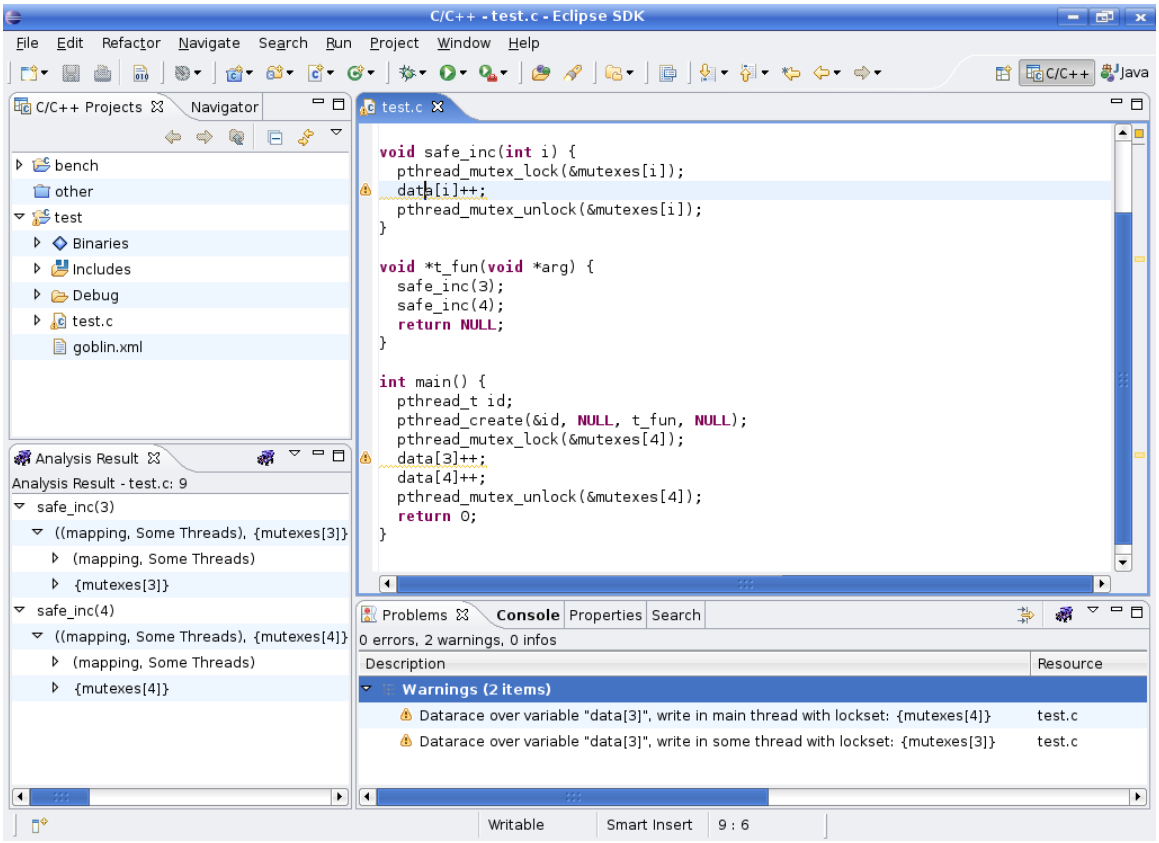


Figure 1.1.: Screenshot of Goblint's Eclipse plug-in [3]

[1] http://goblint.in.tum.de

2. Theory and Goblint

2.1. Program analysis

The goal of program analysis is to gain knowledge about certain properties of a program, which is useful for testing, verification and program optimization. This can be done in two ways: by observing its execution which is referred to as dynamic analysis or by analyzing its code which is called static analysis.

dynamic analysis Testing is normally done with a range of different inputs. The inputs are chosen so that as much of the code as possible is executed. This is measured as *code coverage*. Testing every possible combination might take very long, so that choosing the input classes is essential for the effectiveness of the test and might require some experience. Parameters that can not be chosen, like how threads are scheduled by the operating system, make it difficult to test for race conditions. Also, the act of testing itself could influence the system in a way that problems only occur when not testing.

This approach has several disadvantages: effort of writing good test suites, errors can remain undiscovered and their absence can not be proven. The advantage is that it only finds true errors. The run-time of the analysis is directly proportional to the execution time of the program.

static analysis The source code of the program, or something derived from it, is analyzed without being executed. The simplest form is to look for certain patterns in the code, which is not very flexible and mostly used to check style or coding conventions.

A more powerful approach is abstract interpretation (originally proposed by Cousot and Cousot [4]), which tries to derive semantics from the code. The problem is that there is no general and effective method to do so. The analysis could possibly not terminate, which is why sometimes the semantics has to be approximated in order to avoid non-termination. In general, the analysis can be accelerated with the cost of getting less precise.

For a sound analysis this means that it would find more errors than there actually are in the program, i.e. it overestimates. The advantage is that it will find all real errors. So if it does not find any errors, the program is guaranteed to be error-free.

A disadvantage is that the analysis could need much more time and memory than the execution of the program, so that subsystems have to be analyzed separately in order to remain within acceptable boundaries.

2.2. Complete lattices

An important component for static analysis through abstract interpretation are complete lattices, which will be used as abstract domains later on.

Definition 2.1 (Partial order) *A set $\mathbb{D}$ and a binary relation $\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$ which is reflexive, anti-symmetric and transitive.*

Definition 2.2 ((Least) upper bound) *An element $d \in \mathbb{D}$ is called upper bound of a subset $X \subseteq \mathbb{D}$ if $x \sqsupseteq d$ for all $x \in X$. It is called least upper bound $\bigsqcup X$ if it is an upper bound and $d \sqsupseteq y$ for every upper bound y of X.*

Definition 2.3 (Complete lattice) *A partial order where every subset $X \subseteq \mathbb{D}$ has a least upper bound $\bigsqcup X \in \mathbb{D}$.*

The counterpart to the *least upper bound* is the *greatest lower bound*. They are also called *join* and *meet*, respectively written as $\bigsqcup X$ and $\bigsqcap X$ for a set X.

A lattice is called *bounded* if it has a greatest and a least element, which are referred to as *top* ($\top$) and *bottom* ($\bot$).

Every complete lattice has

- a least element $\bot = \bigsqcup \emptyset \in \mathbb{D}$
- a greatest element $\top = \bigsqcup \mathbb{D} \in \mathbb{D}$.

For example, $\mathbb{D} = \mathbb{Z}$ with the relation "=" is not a complete lattice since it has no least upper bound or greatest lower bound. However, the lattice $= \mathbb{Z} \cup \{\bot, \top\}$ with "=" (shown in Figure 2.1) is complete. A lattice of this form is called *flat*.

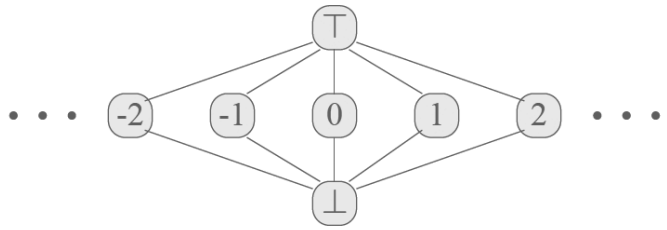


Figure 2.1.: Flat lattice [11]

2.3. Operational semantics and abstract interpretation

C programs consist of a finite set of procedures $Proc$, including the main procedure ($main \in Proc$), which is executed first.

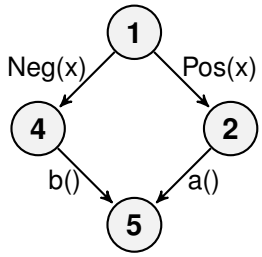
A control flow graph G_p for a procedure $p \in Proc$ is a tuple (N_p, E_p, e_p, r_p):

- N_p is the finite set of nodes, which represent program points
- E_p is the finite set of edges, which represent steps of computation

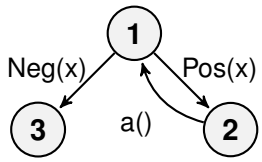
- $e_p \in N_p$ is the start node, which represents the entry point
- $r_p \in N_p$ is the end node, which represents the return point

An edge from node u to v with a label lab is defined as (u, lab, v). For tests, the edge label $Pos(e)$ is used for the branch where the expression e evaluates to `true` and $Neg(e)$ for the branch where it evaluates to `false`. This notion is also used for representing loops as shown in Figure 2.2.

```
if(x){
  a();
}else{
  b();
}
```




```
while(x){
  a();
}
```




```
for(int i=0; i<42; i++){
  a();
}
```

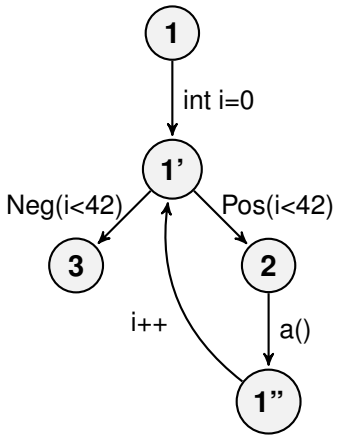


Figure 2.2.: Control flow graphs with branching

A path π is a sequence of edges. Computations follow paths in the graph and transform the current state $s \in S$. Every edge $k = (u, lab, v)$ defines a partial transformation

$$[\![k]\!] = [\![lab]\!] : S \to S \tag{2.1}$$

of the state. This is called the *concrete effect* of the edge.

The result of the computation of a path $\pi = k_1\pi_1$ on a state s is defined as

$$[\![k_1\pi_1]\!]\ s = [\![\pi_1]\!]\ ([\![k_1]\!]\ s) \tag{2.2}$$

and the result for the empty path $\pi = \epsilon$ as

$$[\![\epsilon]\!]\ s = s. \tag{2.3}$$

The *concrete semantics* then specifies the type of state and the transfer functions for edges. Since we are interested in the state at a given program point and there might be multiple paths leading to that point, we have to consider all the paths, which results in a powerset of possible states. This means that we are looking for a mapping

$$\sigma : N \to 2^S. \tag{2.4}$$

This is called the *collecting semantics*, which contains all the possible states at program points N. Analyses like constant propagation can be used to improve the precision by excluding paths (e.g. branches that can never be taken).

However, cycles in the control flow graph might lead to infinitely many paths, which is why the collecting semantics can not be computed in general.

Abstract interpretation To solve this problem, the concrete semantics is soundly approximated by the abstract semantics, consisting of a domain $\mathbb{D}$ which has to be a complete lattice, and abstract transfer functions which have to be monotonic. As monotonic functions on a complete lattice which satisfies the ascending chain condition are guaranteed to converge to a least fixed point, computability is gained at the cost of precision.

A concrete state $s \in S$ is described by an abstract state $d \in \mathbb{D}$ if both are in the *description relation* $\Delta \subseteq S \times \mathbb{D}$:

$$s \ \Delta \ d \tag{2.5}$$

If a concrete state $s \in S$ is described by $d_1 \in \mathbb{D}$, it is also described by a greater $d_2 \in \mathbb{D}$:

$$s \ \Delta \ d_1 \ \wedge \ d_1 \sqsubseteq d_2 \implies s \ \Delta \ d_2 \tag{2.6}$$

A *concretization function* $\gamma : \mathbb{D} \to 2^S$ is used to map an abstract state to all the concrete states it simulates:

$$\gamma \ d = \{s \mid s \ \Delta \ d\} \tag{2.7}$$

It holds that a concretized abstract state is always described by the original and that the concretization of an abstracted concrete state will be a super set of the original:

$$s \in \gamma \ d \implies s \ \Delta \ d \tag{2.8}$$

$$s \ \Delta \ d \implies s \subset \gamma \ d \tag{2.9}$$

The abstract effect of an edge k is

$$[\![k]\!]^\sharp = [\![lab]\!]^\sharp : \mathbb{D} \to \mathbb{D} \tag{2.10}$$

and must always simulate the concrete effect, i.e.

$$s \ \Delta \ d \implies ([\![k]\!] \ s) \ \Delta \ ([\![k]\!]^\sharp \ d) \tag{2.11}$$

Paths are defined analog to concrete paths. Since the abstract effect for edges keeps up the description relation, it also holds that

$$s \ \Delta \ d \implies ([\![\pi]\!] \ s) \ \Delta \ ([\![\pi]\!]^\sharp \ d) \tag{2.12}$$

$$s \ \Delta \ d \implies ([\![\pi]\!] \ s) \in \gamma([\![\pi]\!]^\sharp \ d) \tag{2.13}$$

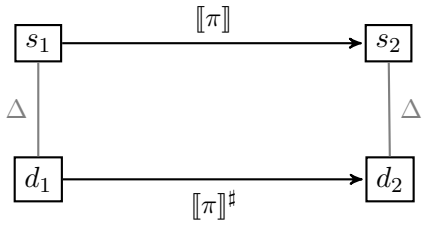


Figure 2.3.: Description relation between concrete and abstract paths [11]

which is illustrated in Figure 2.3.

Although the above is enough to guarantee safe abstraction, it might not be optimal. Analogously to the concretization function γ, there is an abstraction function $\alpha : 2^S \to \mathbb{D}$:

$$\alpha\ S = d \in \mathbb{D}.\ \forall s \in S.\ s\ \Delta\ d \tag{2.14}$$

The monotonic abstraction and concretization functions form a Galois connection between concrete and abstract states:

$$2^S \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{D} \tag{2.15}$$

Defining the abstract semantics such that

$$[\![k]\!]^\sharp\ d = \alpha\ \{[\![k]\!]\ s \mid s \in \gamma\ d\} \tag{2.16}$$

guarantees not only safe, but also optimal abstraction.

Computability and solving The collecting semantics is abstracted by the so called Merge Over all Paths (MOP) solution: for a point v, start point $start$ and start state $d_0 \in \mathbb{D}$ it is defined as

$$\mathcal{I}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\ d_0 \mid \pi : start \to^* v\}. \tag{2.17}$$

Instead of this join over an possibly infinite set, a constraint system is used to guarantee computability. Since the abstract state is a complete lattice and the transformations are monotonic, the Knaster-Tarski theorem states that the set of fixed points must also be a complete lattice, which implies the existence of a least/greatest fixed point (complete lattices can not be empty). This solution is computable if the complete lattice has finite height or if it does not contain infinite strictly ascending chains. Under these premises the constraint system will always converge to a least fixed point solution that is an upper bound for the MOP solution [8, 7].

The constraint system for a start point $start$ and start state $d_0 \in \mathbb{D}$ is set up as

$$\mathcal{I}[start] \sqsupseteq d_0 \tag{2.18}$$

$$\mathcal{I}[v] \sqsupseteq [\![k]\!]^\sharp\ (\mathcal{I}[u]) \qquad \forall k = (u, \_, v) \in E \tag{2.19}$$

and can be solved using various iteration schemes. A solution of the constraint system then approximates the MOP solution:

$$\mathcal{I}[v] \sqsupseteq \mathcal{I}^*[v] \qquad \forall v \in N \tag{2.20}$$

If all effects of edges $f = [\![k]\!]^\sharp$ are distributive, i.e.

$$f(\bigsqcup X) = \bigsqcup \{f(x) \mid x \in X\} \qquad \forall \emptyset \neq X \subseteq \mathbb{D} \tag{2.21}$$

the two solutions are even equal [9]:

$$\mathcal{I}[v] = \mathcal{I}^*[v] \qquad \forall v \in N \tag{2.22}$$

Interprocedural While these concepts work fine intraprocedurally, programs with multiple procedures need special attention. For simplification we assume that variables are uniquely named and parameters and return values are handled as assignments to global variables. A procedure `p` has exactly one definition `void p(){...}` and can be called via `p()`. We introduce call edges from a call site to the start of the procedure definition and return edges from the return point to the point after the call site. Figure 2.4 shows such a combined graph of multiple procedures.

```
1  #include <stdio.h>
2
3  int x;
4
5  void f(){
6      x++;
7  }
8
9  void g(){
10     f();
11 }
12
13 void main(){
14     x = 0;
15     f();
16     g();
17     printf("%i", x);
18 }
```

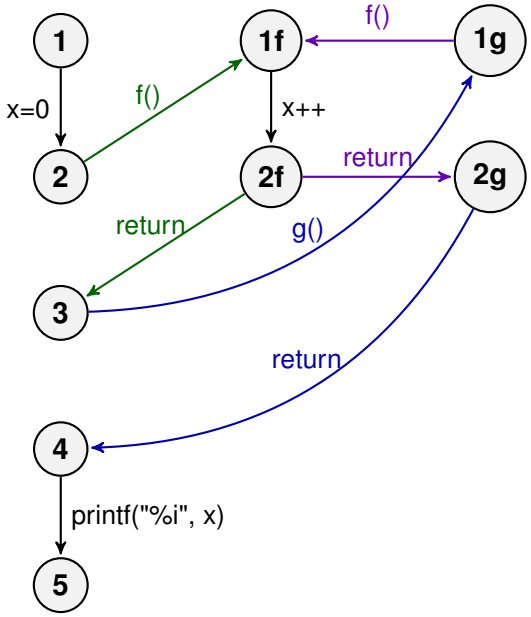


Figure 2.4.: Control flow graph with multiple procedures

The problem now is that not all paths are valid, i.e. some paths would be considered for the solution despite not being possible executions of the program. This happens because we do not ensure that function calls return to the right place. In the example, there is only one valid path (1-2-1f-2f-3-1g-1f-2f-2g-4-5), but we would consider infinitely many paths because of the cycle (3-1g-1f-2f-3).

An interprocedural flow graph G^*, as defined in [12], solves this problem by only allowing so called interprocedurally valid paths ($IVP(G^*)$). It is defined as (N^*, E^*, e_{main}) where e_{main} is the entry point of the program.

An interprocedurally valid path $\pi \in IVP(G^*)$ can be determined by its call-string $cs(\pi)$, which is the subsequence of edges that have not been returned yet. Both can be inductively defined on the length of π:

- if $\pi = \epsilon$ then $\pi \in IVP(G^*)$ and $cs(\pi) = \epsilon$
- if $\pi = \pi' k$ and $\varsigma = cs(\pi')$ then $\pi \in IVP(G^*)$ iff $\pi' \in IVP(G^*)$ and one of the following holds:

  k is neither a call nor a return edge. $cs(\pi) = \varsigma$.

  k is a call edge. $cs(\pi) = \varsigma k$.

  k is a return edge, ς is of the form $\varsigma' k'$ and k corresponds to the call edge k'. $cs(\pi) = \varsigma'$.

The domain is then augmented by this information and the transfer functions for call and return edges modified to ignore invalid edges based on it. Therefore all paths over edges from E^* are guaranteed to be valid.

Although Goblint uses a different approach, this concept of interprocedural paths is used in the following, since it allows to apply the same methods for both settings.

2.4. Soundness vs. precision

Since the program behavior is merely over- or under-approximated, one has to differentiate between information that may or must be true. Although this applies to all domains, we take may- and must-sets as an example since they are used later on in the implemented analysis. Both domains with their corresponding join operation and the meaning of the empty set are described below.

Must-set Property must be true for all elements, but not all elements with the property must be in the set. $\sqcup = \cap$, $\emptyset = \top$.

May-set Property may be true or not for each element, but all elements for which it is true must be in the set. $\sqcup = \cup$, $\emptyset = \bot$.

If the sets contain elements we want to warn about, then the difference is

Must-set Precision: every warning is an error, but the program may still have other errors.

May-set Soundness: there might be false positives, but if there are no warnings, then the program is error-free.

To summarize: the must-set is precise but maybe unsound and the may-set is sound but maybe not very precise.

2.5. Goblint

Overview Goblint uses a recursive demand-driven solver, i.e. constraints get evaluated recursively once they are needed. Results can be shared between different analyses using a query system.

For interprocedural analysis global invariants are used to collect side-effects of functions. These invariants are then used for all calls.

The results can be formatted as XML, JSON or HTML. The XML-output can be displayed in Eclipse via plugin. CIL is used for processing input files and generating a control flow graph (see Figure 2.5).

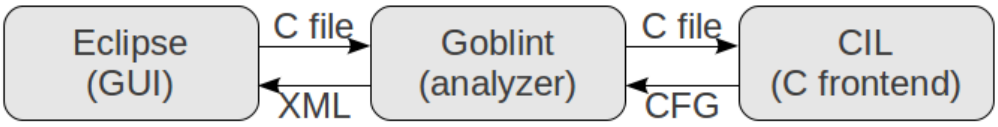


Figure 2.5.: Components used by Goblint [13]

CIL CIL stands for C Intermediate Language and is an infrastructure for C program analysis and transformation [10]. It is used to generate a high-level representation of the input C program by doing a a source-to-source transformation that simplifies valid C programs into core constructs with very clean semantics. CIL's output can be displayed by invoking Goblint with the option `--set justcil true`. It is able to process ANSI-C programs and also programs using Microsoft C or GNU C extensions. Modules for control flow graphs, data flow and some analyses are also included.

Query system Analyses in Goblint can be run on demand, i.e. the constraints need only be evaluated once a value is needed. All the analyses are combined into a master analysis.

Among others the query system can be used to get information about

- expression evaluation
- expression equality
- pointer analysis
- reachability
- locksets

Expression evaluation and points-to information will be used later on in the implementation.

Domains Since complete lattices are needed for termination, all domains must offer functions for $\leq, \sqcup, \sqcap, \bot, \top$. Custom functions for widening and narrowing can be specified to make the constraint system converge faster.

Basic data structures like sets and maps are already available for use as domains. Since CIL values can not be used directly, there are also domains for representing C constructs like structures, arrays, L-values and so on.

Analyses For implementing an analysis the following transfer functions have to be defined and are called by the master analysis. The used domain is `D` and `D.t` its type. The types `lval` (L-value), `exp` (expression), `fundec` (function declaration) and `varinfo` (variable) come from CIL.

`assign (lval:lval) (rval:exp)`
: Assignment of an expression `rval` to a L-value `lval`.

`branch (exp:exp) (tv:bool)`
: Enter a branch where the condition `exp` is either true or false, depending on `tv`.

`body (f:fundec)`
: Called when the body of a function is entered.

`return (exp:exp option) (f:fundec)`
: Called once a function returns, `exp` contains the expression if one is returned.

`enter (lval: lval option) (f:varinfo) (args:exp list)`
: Enter a function `f` with arguments `args` and the returned value optionally being saved to `lval`.

`combine (lval:lval option) fexp (f:varinfo) (args:exp list) (au:D.t)`
: Leave a function `f` and combine the updated domain `au` with the context of the call site. Counterpart to `enter`.

`special (lval: lval option) (f:varinfo) (arglist:exp list)`
: Called for functions that are not defined in the program.

Functions: enter and combine As mentioned in Section 2.3 Goblint uses a different approach than interprocedurally valid paths; instead, it represents program executions as computation forests.

For all calls $(x, f(), y) \in E_p$ to a function f with start node e_f and return node r_f inside a procedure $p \in Proc$, we set up the constraints

$$\sigma[e_f] \sqsubseteq enter\ \sigma[x] \tag{2.23}$$

$$\sigma[y] \sqsubseteq combine\ (\sigma[x], \sigma[r_f]) \tag{2.24}$$

Depending on the analysis, $enter$ and $combine$ can then be defined accordingly. For a simple value analysis without escaping of variables, they might be

$$enter\ d = d|_{Globals} \tag{2.25}$$

$$combine\ (d_1, d_2) = (d_1|_{Locals}) \oplus (d_2|_{Globals}) \tag{2.26}$$

which means that $enter$ only keeps the global variables in the domain and $combine$ merges the locals from the call-site with the globals from the return point of the function.

Since Goblint uses calling contexts (denoted as a below) to differentiate between function calls at a node, the real constraints are different, but the idea is similar:

$$\sigma[e_f, a] \sqsubseteq a \tag{2.27}$$

$$\sigma[y, a] \sqsubseteq combine\ (\sigma[x, a], \sigma[r_f, enter\ \sigma[x]]) \tag{2.28}$$

The context helps to improve precision since a call to f with context $a \in \mathbb{D}$ will be treated separately from a call with context $b \neq a$.

In summary the result of this approach is similar to that of interprocedurally valid paths: $enter$ and $combine$ make sure that functions return to their call-site and the context distinguishes different calls.

3. Verifying correct usage of file handles

3.1. Common problems using files

The following examples for common problems served as a guideline for the implementation and contain comments starting with `// WARN:` that indicate where warnings would be output.

Listing 3.1 shows opening a file `log.txt` and appending the line "Testing..." to it. At the end the file is closed.

```
1 #include <stdio.h>
2
3 int main(){
4   FILE *fp;
5   fp = fopen("log.txt", "a");
6   fprintf(fp, "Testing...\n"); // log something
7   fclose(fp);
8   // do important things
9 }
```

Listing 3.1: Append text to file. Everything fine?

Opening files This might seem fine, since the file will be created if it does not exist, but what happens if the file can not be written to? If the file exists but we do not have write access, running the code will result in a segmentation fault at `fprintf`. We forgot to check the result of `fopen`, which returns a null pointer if the file could not be opened successfully. Accessing a file that does not exist for reading also results in a segmentation fault.

This case is handled by the manual implementation and can also be checked using the specification language. For this example warnings should be issued that the file handle might not be open after line 5.

A corrected version could look like Listing 3.2.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5   FILE *fp;
6   fp = fopen("log.txt", "a");
7   if(fp){
8     printf("file opened");
9     fprintf(fp, "Testing...\n");
10    fclose(fp);
11  }else{
12    perror("failed to open file");
```

```
13      return EXIT_FAILURE;
14    }
15    return EXIT_SUCCESS;
16  }
```

Listing 3.2: Success check for fopen

For the sake of brevity a success check is omitted in the following examples and it is assumed that the file could be opened without errors. The specification version can be easily adjusted to conform to this by just replacing a few states, whereas the manual implementation would always warn about maybe unopened file handles, unless the option `ana.file.optimistic` is set to `true`, in which case the analysis will assume opening files never fails (see Section B.3 on how to set options). Listing 3.3 shows what happens if the file handle was not opened before using it. Dereferencing an uninitialized pointer is undefined behavior, but when running the program, this usually leads to a segmentation fault.

```
1  #include <stdio.h>
2
3  FILE *fp;
4
5  int main(){
6    fprintf(fp, "Testing...\n"); // WARN: writing to unopened file
       handle fp
7    fclose(fp); // WARN: closeing unopened file handle fp
8  }
```

Listing 3.3: Missing fopen

Closing files Not closing files is not necessarily an error since file handles are closed at the end of the program anyway, but it is bad practice and might lead to unwanted behavior. Imagine a program that is done writing important information to a file but does not close it. What happens if the program gets stuck in calculations or on user input and other programs want to use the file? See Listing 3.4 for example. Without the call to `fclose`, the written content might not be flushed until the program terminates. Starting with some content in the file resulted in an empty file at the point of user input.

```
1  #include <stdio.h>
2
3  FILE *fp;
4
5  int main(){
6    char text[20];
7    fp = fopen("test.txt", "w");
8    fprintf(fp, "Testing...");
9    // fclose();
10   printf("enter some text: ");
11   fgets(text, sizeof text, stdin);
12   printf("text = \"%s\"\n", text);
13 }
```

Listing 3.4: A reason for closing files: flushing

Listing 3.5 has comments for warnings that would be issued. There is a warning where the file was opened and a summary of unclosed files at the end of the program.

```
#include <stdio.h>

FILE *fp;

int main(){
  fp = fopen("test.txt", "a");  // WARN: file is never closed
  fprintf(fp, "Testing...\n");
} // WARN: unclosed files: fp
```

Listing 3.5: Missing fclose

Open mode Writing to a file which is opened read-only as demonstrated in Listing 3.6 is another problem. Bugs of this kind might be hard to find, since this executes without errors - there is just nothing written to the file. Analogously, reading from a file that is opened write-only is the same as reading an empty file.

```
#include <stdio.h>

FILE *fp;

int main(){
  fp = fopen("test.txt", "r");
  fprintf(fp, "Testing...\n"); // WARN: writing to read-only file
   handle fp
  fclose(fp);
}
```

Listing 3.6: Wrong open mode: writing to a read-only file

Table 3.1 lists the modes in which a file can be opened. To open a file in binary mode the character 'b' is appended to the mode string or inserted before '+', e.g. 'rb', 'r+b' or 'rb+'. So 'r' and 'rb' are the only modes that do not support write operations. All the modes that contain 'r' need the file to exist prior to the call.

Mode	Description
r	reading
w	writing (truncate or create file)
a	appending (start at end or create file)
r+	reading and writing (start at beginning)
w+	reading and writing (truncate or create file)
a+	reading and writing (start at end or create file)

Table 3.1.: Possible modes for opening a file [6]

Other functions like `fscanf`, `fputc`, `fgetc`, `fwrite`, `fread` are not shown here, since the problems are similar to `fprintf`.

3.2. Concrete and abstract semantics

For the concrete semantics we are only interested in the effects of statements on file handles. Other parts of the semantics are intentionally left undefined. File handles are L-values pointing to a structure `FILE` which keeps information about the file descriptor, the stream position, a pointer to the stream's buffer and some status flags. We differentiate between the states

- unopened (pointer not initialized)
- opened a file in a specific mode
- could not open (NULL pointer is returned)
- closed

The type of the concrete state is then

$$S' : Lval \to (File * Mode) \cup \{Error, Closed\} \tag{3.1}$$
$$S : S' * Other \tag{3.2}$$

where S' is a mapping from L-values to the states opened, error or closed. A L-value is unopened if it is not mapped. The types $File$ and $Mode$ are strings. $Other$ represents the semantics of everything other than file handles and can be used for evaluating expressions.

The `FILE` structure is only supposed to be accessed by functions defined in `stdio.h` or `wchar.h`. Since there are many functions for reading and writing files, only `fscanf` and `fprintf` are shown here as an example.

For the open mode we define the predicates (with $\mathcal{L}(regex)$ being the set generated by the regular expression $regex$)

$$readwrite(m) = m \in \mathcal{L}(\text{[rwa] (\+ | \+b | b\+)}) \tag{3.3}$$
$$readable(m) = readwrite(m) \vee m \in \{\text{r}, \text{rb}\} \tag{3.4}$$
$$writable(m) = readwrite(m) \vee m \in \mathcal{L}(\text{[wa] b?}) \tag{3.5}$$
$$unkown(m) = \neg readable(m) \wedge \neg writable(m) \tag{3.6}$$

The transfer function is overloaded depending on the state it operates on. We assume that for $o : Other$ everything is well defined, so that we can use $[\![e]\!]\ o$ to evaluate an expression e and need not worry about capturing side effects for other states. That means we do not need to modify $Other$ in the transfer function on S since this is done separately.

In the following we define the transfer function for file handles on the concrete state $(s, o) : S$ which yields a set of new states 2^S. The only case where we return a set with more than one element is `fopen`: the result is either a state where the file was successfully opened or a state where it was not.

Expressions that are evaluated in $Other$ can be arbitrarily complex, e.g. the last case [[p1 = p2]] also includes statements like `p1++`, `p1-=3`, `p[0].file=fun()+1` and so on.

Side effects that we are interested in are denoted after the state.

$$
\begin{aligned}
[\![\text{p = fopen(f, m)}]\!]\ (s,o) &= \{(s \oplus \{[\![\text{p}]\!]\ o \mapsto ([\![\text{f}]\!]\ o, [\![\text{m}]\!]\ o)\}, o), (s,o)\} \\
[\![\text{fclose(p)}]\!]\ (s,o) &= \{(s \oplus \{[\![\text{p}]\!]\ o \mapsto \mathit{Closed}\}, o)\} \\
[\![\text{fprintf(p, args)}]\!]\ (s,o) &= \{(s,o)\} \\
&\quad \text{and} \begin{cases}
\text{something written} & \text{if } s([\![\text{p}]\!]\ o) = (f,m) \wedge \mathit{writable}(m) \\
\text{nothing written} & \text{if } s([\![\text{p}]\!]\ o) = (f,m) \wedge \neg \mathit{writable}(m) \\
 & \quad \vee\ s([\![\text{p}]\!]\ o) = \mathit{Closed} \\
\text{segmentation fault} & \text{if } s([\![\text{p}]\!]\ o) = \mathit{Error} \\
\text{undefined} & \text{else}
\end{cases} \\
[\![\text{fscanf(p, args)}]\!]\ (s,o) &= \{(s,o)\} \\
&\quad \text{and} \begin{cases}
\text{something read} & \text{if } s([\![\text{p}]\!]\ o) = (f,m) \wedge \mathit{readable}(m) \\
\text{nothing read} & \text{if } s([\![\text{p}]\!]\ o) = (f,m) \wedge \neg \mathit{readable}(m) \\
 & \quad \vee\ s([\![\text{p}]\!]\ o) = \mathit{Closed} \\
\text{segmentation fault} & \text{if } s([\![\text{p}]\!]\ o) = \mathit{Error} \\
\text{undefined} & \text{else}
\end{cases} \\
[\![\text{p1 = p2}]\!]\ (s,o) &= \\
&\quad \begin{cases}
\{(s \oplus \{[\![\text{p1}]\!]\ o \mapsto x\}, o)\} & \text{if } ([\![\text{p2}]\!]\ o, x) \in s \\
\{(s \ominus \{[\![\text{p1}]\!]\ o \mapsto y\}, o)\} & \text{if } ([\![\text{p2}]\!]\ o, x) \notin s \wedge ([\![\text{p1}]\!]\ o, y) \in s \\
\{(s,o)\} & \text{else}
\end{cases}
\end{aligned}
$$

All other statements are assumed to not affect the state of file handles or affect it in a way that is not relevant for us, e.g. changing the position indicator inside the file with `fseek` or similar functions.

Theory Remembering the interprocedural flow graph $G^* = (N^*, E^*, e_{main})$ we are now looking for a function

$$\sigma : N^* \to 2^S \tag{3.7}$$

that gives us the set of possible states for a program point. That means that for every valid path $\pi \in \mathit{IVP}(G^*)$ from e_{main} to some $n \in N^*$ and start state s_0 the concrete state must be included in the set:

$$\sigma[n] \supseteq \{[\![\pi]\!]\ s_0\} \qquad \forall n \in N^*, \pi = (e_{main}, \_, \_)...(\_, \_, n) \in \mathit{IVP}(G^*) \tag{3.8}$$

To describe concrete values $s \in 2^S$, we introduce a domain with abstract values $d \in \mathbb{D}$ and a description relation Δ with

$$s\ \Delta\ d_1\ \wedge\ d_1 \sqsubseteq d_2\ \implies\ s\ \Delta\ d_2 \tag{3.9}$$

3.3. A domain for representing file handle usage

Since it should be possible to track multiple file handles, we choose our domain $\mathbb{D}$ to be a a map $\mathbb{M}$ from L-values K to another domain $\mathbb{V}$. The bottom value for $\mathbb{M}$ is the empty map. The domain $\mathbb{V}$ represents one file handle. Listing 3.7 shows how its type `t` is defined.

```
type loc = location list
type mode = Read | Write
type state = Open of string*mode | Closed | Error
type record = { key: Lval.CilLval.t; loc: loc; state: state }
type t = record Set.t * record Set.t (* must, may *)
```

Listing 3.7: Type of the file handle domain

t is a tuple consisting of a must- and a may-set of records.

record contains the L-value that was used as a key, the location stack and the state.

state can be `Open(filename, mode)`, `Closed` or `Error`.

mode can be `Read` if the file is opened read-only or `Write` for all other modes.

loc is a stack of locations from the latest assignment down to the use of the stdio-function.

Filename and location stack are not needed do determine the correct state but instrumentalization to get more helpful warning messages.

Each key *must* have at most one state but *may* have at least one state. In other words: the must-set starts with one element and can only shrink to zero elements; the may-set also starts with one element and can only grow. Although the must-set could be replaced by a more efficient type, it is easier to work with sets for both.

Assume the must-set is empty and the may-set contains multiple elements. Even if the correct state is not known, these alternatives can be used to answer questions about the state during the analysis.

As an example let the may-set contain records with the states `Open(..., Read)` and `Open(..., Write)`. In this case it is safe to say that the file is opened - if it is writable on the other hand is unknown. Another example: if all states are `Closed` but with different locations, it is safe to say that the file is closed.

Since an empty may-set would never occur, we can use it to encode the case where we have no knowledge anymore and the state could be anything (e.g. after an unsupported operation like pointer arithmetic). For a file handle $\mathbb{M}[k]$ with key k we therefore define

$$\mathbb{M}[k] = \bot \Leftrightarrow k \notin \mathbb{M} \tag{3.10}$$

$$\mathbb{M}[k] = \top \Leftrightarrow \mathbb{M}[k] = (\emptyset, \emptyset). \tag{3.11}$$

The ordering and the join operation for two values (a, b) and (c, d) are defined as

$$(a, b) \leq (c, d) \Leftrightarrow c \subset a \wedge b \subset d \tag{3.12}$$

$$(a, b) \sqcup (c, d) = (a \cap c, b \cup d). \tag{3.13}$$

The location stack is kept because the location of the stdio-function might not always be the location where the warning should be issued. Listing 3.8 defines a custom function for opening files. The warnings should be placed at the call to this function instead of at the call to `fopen`.

```c
#include <stdio.h>
#include <stdlib.h>

FILE* myfopen(char* filename){
  FILE *fp;
  fp = fopen(filename, "a");
  if(fp == NULL){
    printf("Error opening file");
    exit(EXIT_FAILURE);
  }else{
    return fp;
  }
}

int main(){
  FILE *fp1;
  FILE *fp2;
  fp1 = myfopen("test1.txt");
  fp2 = myfopen("test2.txt"); // WARN: file is never closed

  fprintf(fp1, "Testing...\n");
  fclose(fp1);
  fprintf(fp2, "Testing...\n");
  // fclose(fp2);
} // WARN: unclosed files: fp2
```

Listing 3.8: Location of warning when using custom function for opening files

Termination However, using a normal stack could lead to infinite strictly ascending chains as shown in Listing 3.9. Once the uninitialized variable b contains 0, the file will be opened. This normally happens pretty fast before overflowing the call stack. So the program runs fine, but the analysis would get stuck with an ever growing location stack. To avoid this, the location stack behaves like an ordered set, i.e. if a location is already contained in the stack, it will not be pushed.

```c
#include <stdio.h>

FILE* myfopen(char* filename){
  int b;
  if(b)
    return myfopen(filename);
  else
    return fopen(filename, "a");
}

int main(){
  FILE *fp;
  fp = myfopen("test.txt");
  fprintf(fp, "Testing...\n");
  fclose(fp);
```

```
16 }
```

Listing 3.9: Infinitely growing location stack

In Listing 3.10 it is not the call stack, but the filename that might escalate. The analysis is not precise enough to know that `test-odd.txt` must be opened but it will end up with a may-set containing both filenames. This is acceptable since it still knows that the file handle must be open, the problem however is that this could lead to infinite strictly ascending chains if the number of possible filenames is unbounded. This is avoided by the fact that only string literals can be used for filenames - any operation on the string will make it $\top$. Since there can only be finitely many string literals in the source code, termination is not jeopardized.

However there is the possibility that strings are manipulated in memory or by external functions, in which case the value for the filename could be incorrect. Since filenames are currently not displayed in warnings and it does not violate soundness, this is just an inconvenience.

```
1  #include <stdio.h>
2
3  FILE* myfopen2(int i);
4  FILE* myfopen1(int i){
5    if(i>0)
6      return myfopen2(i-1);
7    else
8      return fopen("test-even.txt", "a");
9  }
10 FILE* myfopen2(int i){
11   if(i>0)
12     return myfopen1(i-1);
13   else
14     return fopen("test-odd.txt", "a");
15 }
16
17 int main(){
18   FILE *fp1;
19   fp1 = myfopen1(5);
20   fprintf(fp1, "Testing...\n");
21   fclose(fp1);
22 }
```

Listing 3.10: Mutually recursive functions

3.4. An analysis for checking file handle usage

The analysis uses the following transfer functions, which are called by the framework, in order to transform the abstract state for different statements and issue warnings according to the side effects specified in the concrete semantics.

```
assign (lval:lval) (rval:exp)
```

Warn about changed file pointer if `lval` $\in$ `D` and set the entry to $\top$. To improve precision aliasing of file pointers was implemented for simple cases (join of two aliases still yields $\top$).

`branch (exp:exp) (tv:bool)`
Used to handle error-case of `fopen`. If `exp` compares an L-value `lval` with an integer and the expression can be transformed into `lval==0` with `tv` being true, then change the state to `Error`.

`return (exp:exp option) (f:fundec)`
If the returning function is `main`, print out a summary of unclosed files if there are any. If a L-value is returned, save it as a special entry `return_var` in the domain. Finally remove all non-escaping formals and locals of the function from the domain.

`enter (lval: lval option) (f:varinfo) (args:exp list)`
Save the current location to the location stack if the function is not `main`. Only keep globals and variables that are reachable from `args` in the domain for `f`.

`combine (lval:lval option) fexp (f:varinfo) (args:exp list) (au:D.t)`
Pop the top element from the location stack. Add all entries from the updated domain `au` to the local domain. If `return_val` is set and there is an `lval` which is assigned to, save the entry `return_val` points to with `lval` as a new key in the domain.

`special (lval: lval option) (f:varinfo) (arglist:exp list)`
Add the current location to the location stack. Issue warnings and/or modify domain depending on `lval` and the called function.

The implementation of these functions constitutes the abstract effects $[\![k]\!]^\sharp\ d$ on $d \in \mathbb{D}$ for all statements k. For example

$$[\![\text{p1 = p2}]\!]^\sharp\ d = \texttt{assign p1 p2} \tag{3.14}$$

As the effects resemble those specified in the concrete semantics, most of the work is done in `special`. Analogously to $Other$ in the concrete semantics, the implementation relies on the query system to get may-point-to information and to evaluate expressions.

The analysis is sound if the abstract effects describe the concrete effects correctly for all edges k:

$$s\ \Delta\ d \implies [\![k]\!]\ s\ \Delta\ [\![k]\!]^\sharp\ d \tag{3.15}$$

Since the transitions between states of file handles are treated the same as in the concrete semantics, the analysis is sound for the states and issued warnings. Filenames, although not used, might be unsound.

4. A specification language for regular safety properties

4.1. Representing the state of properties using automata

Our goal is to abstract the semantics of a program in order to verify properties given by a specification. The behavior of a system can be described using state diagrams, consisting of a finite number of states and transitions between those states. Such state diagrams can be formalized by so called finite state machines or finite automata. The transitions can be deterministic (at most one transition for each state and input) or nondeterministic (multiple possible next states for each state and input). Both deterministic finite automata (DFA) and nondeterministic finite automata (NFA) are usually defined by a 5-tuple $(S, \Sigma, \delta, s_0, F)$, consisting of

- a finite set of states (S)
- a finite set of input symbols called the alphabet (Σ)
- a transition function ($\delta : S \times \Sigma \to S$)
- a start state ($s_0 \in S$)
- a set of accept states ($F \subseteq S$).

The automaton then accepts a string $w = a_1 a_2 ... a_n$ over the alphabet Σ if there is a sequence of states $r_0, r_1, ..., r_n$ in S with

- $r_0 = s_0$
- $r_{i+1} = \delta(r_i, a_{i+1})$, for $i = 0, ..., n-1$
- $r_n \in F$.

Such an automaton can either accept or not accept a given input. In our case this would require us to construct an automaton for every property we want to warn about. For each input we would do the transitions for all automata and every time an automaton reaches an end state, we would issue the corresponding warning and reset the automaton. Since we are only interested in the warnings this is not the best approach.

A better suited solution for our purpose is a finite state transducer, which has two tapes: one for input and one for output. For defining the output function, there are two possibilities:

- a Moore machine determines the output values by its current state,

- a Mealy machine determines the output values by its current state and the current input.

The Mealy machine was chosen as a better fit for the specification since it is more flexible and avoids introducing intermediate states that are used solely for output. Furthermore it keeps the number of states low (n^2 vs. n possible output values for n states), which is good for visualization.

Compared to a finite automaton a Mealy machine is a 6-tuple $(S, s_0, \Sigma, \Lambda, T, G)$, consisting of

- a finite set of states (S)
- a start state ($s_0 \in S$)
- a finite set called the input alphabet (Σ)
- a finite set called the output alphabet (Λ)
- a transition function ($T : S \times \Sigma \to S$)
- an output function ($G : S \times \Sigma \to \Lambda$).

The transition and output functions can be coalesced into a single function ($T' : S \times \Sigma \to S \times \Lambda$), which is meant when referring to transitions from now on. A transition, which corresponds to an edge in the graph, therefore consists of an input and an output value.

For the specification we use a Mealy machine where

- the states define the properties (e.g. file handle is open or closed),
- the input alphabet consists of the statements of the program,
- the output alphabet consists of the warnings and the empty element ϵ to avoid output.

Using concrete statements for the transitions would not be very flexible, which is why constraints are used instead. The constraints for each state form an extra automaton and work similar to pattern matching in functional languages: they can contain identifiers for binding values and wildcards for matching everything. Once a constraint matches the input statement, the transition is taken. For string constants regular expressions are also supported. This allows a very concise specification of alternatives.

Theory Next we will examine how the Mealy machine relates to concrete and abstract semantics of the specification.

First let T, G and T' be defined for paths:

$$T(s, \epsilon_\Sigma) = s \tag{4.1}$$
$$T(s, k\pi') = T(T(s, k), \pi') \tag{4.2}$$
$$G(s, \epsilon_\Sigma) = \emptyset \tag{4.3}$$
$$G(s, k_1 k_2 \pi') = G(s, k_1) \cup G(T(s, k_2), \pi') \tag{4.4}$$
$$T'(s, \pi) = (T(s, \pi), G(s, \pi)) \tag{4.5}$$

Furthermore we expect them to work on edges $k \in E^*$, in which case they just use the label lab.

That means that the concrete state at the end of a path is a tuple with the current state of the Mealy machine and a set of outputs that happened along the path. Each output has its own location derived from the edge where it happened. We keep a set in order to be able to consolidate the outputs of different paths at the end of the program, which is only needed in case of multiple returns.

Given an interprocedural flow graph $G^* = (N^*, E^*, e_{main})$ and a Mealy machine $(S, s_0, \Sigma = E^*, \Lambda, T, G)$, the collecting semantics for a node $v \in N^*$ is then

$$\mathcal{I}'[v] = \bigcup \{[\![\pi]\!]\ (s_0, \emptyset) \mid \pi : e_{main} \to^* v\} \tag{4.6}$$

where

$$[\![\pi]\!]\ (s, o) = (s', o \cup o') \text{ with } (s', o') = T'(s, \pi). \tag{4.7}$$

Since this solution can not be computed in general, we define abstract semantics used for the MOP solution

$$\mathcal{I}^*[v] = \bigsqcup \{[\![\pi]\!]^\sharp\ (d_0, \emptyset) \mid \pi : e_{main} \to^* v\} \tag{4.8}$$

which is approximated by the solution of a constraint system that can be computed:

$$\mathcal{I}[start] \sqsupseteq (d_0, \emptyset) \tag{4.9}$$

$$\mathcal{I}[v] \sqsupseteq [\![lab]\!]^\sharp\ (\mathcal{I}[u]) \qquad \forall (u, lab, v) \in E^* \tag{4.10}$$

More concretely we can set up constraints for $\sigma' : N^* \to 2^S$ for abstracting T, and $\omega' : N^* \to 2^{2^\Lambda}$ for abstracting G:

$$\sigma'[v] \sqsupseteq \bigsqcup_{s \in \sigma'[u]} \{T(s, lab)\} \qquad \forall (u, lab, v) \in E^* \tag{4.11}$$

$$\omega'[v] \sqsupseteq \bigsqcup_{s \in \sigma'[u]} \{G(s, lab)\} \qquad \forall (u, lab, v) \in E^* \tag{4.12}$$

The abstract state of the actual implementation is different because it uses must- and may-sets for states and it prints most warnings directly, but the idea is similar.

Both domain and analysis can be seen as generalized versions of those used in Chapter 3. Next we will describe the domain; the implementation details of the analysis are omitted.

4.2. A domain for representing the state of properties

Properties refer to an object - this could be the whole program or something inside the program, which can be addressed by a L-value. For the file handles this was a L-value at a certain position in the checked statements and allowed to differentiate between multiple handles. Apart from L-values from statements, special keys are used to guarantee that the state is always assigned to some key. One such special key is used for global constraints,

i.e. constraints that define no key. This could be used to verify that one function is always called before the other globally. One could also implement other special keys, e.g. to refer to the current function or thread.

The domain for the specification therefore is very similar to the domain for file handles. It consists of a map $\mathbb{M}$ with L-values as a key and a domain $\mathbb{V}$ for its values. $\mathbb{V}$ is a tuple of must- and may-set, each containing records with a key, location stack and state (see Listing 4.1).

```
1 type state = string
2 type record = {key: Lval.CilLval.t; loc: location list; state: state}
3 type t = record Set.t * record Set.t (* must, may *)
```

Listing 4.1: Type of the specification domain

The main difference is that the state is a string instead of a sum type, which means that it is not fixed at compile-time but comes from the specification file at run-time.

4.3. Specification format

A specification file contains three types of definitions:

- warnings, consisting of an identifier and text
- edges, consisting of a start state, optional outputs, optional forwarding, an end state and a constraint
- state groups, consisting of a name for the group and a list of states (currently only used to specify end states).

Definitions are separated by line breaks and can be interleaved since the whole file is parsed and split into a list of warnings and a list of edges. Empty lines and C-style comments are ignored.

Listing 4.2 gives a feel for the syntax using a small example for file handles. The type and amount of whitespace for separation is not important.

```
1 // warnings
2 w1  "file handle is not saved"
3 w2  "closeing unopened file handle"
4
5 // edges
6 a   -w1>    a   fopen(_)
7 a   -w2>    a   fclose($fp)
8 a   ->      b   $fp = fopen(_)
9 ...
```

Listing 4.2: A very small specification for file handles

The semantics and extensions to the syntax are described below.

warnings Multiple warnings can be specified like this: `a -w1,w2,w3> b c()` in which case `w1`, `w2` and `w3` will be output if the automaton is in state `a` and the constraint `c` matches.

states The states S of the Mealy machine are implicitly defined by the start and end states used by edges.

start state The start state of the first transition defines the start state of the automaton.

end states End states are an extension that allows to warn about certain states at the end of the program. They are specified as a list `end: a, b, c`. At the end of the program the warnings with the identifiers `!end` and `!end@return` are issued for all states that are not marked as an end state. The difference between the two is the location for the warning: `!end` places it at the location for that state, `!end@return` places it at the return points of the `main` function. For the latter `$` can be used as a placeholder for the list of keys.

wildcard An edge with `_` as a constraint matches everything. Wildcards can also be used inside expressions.

forwarding Edges with a two-headed arrow like `->>` (or `-w1,w2>>` etc.) are forwarding edges, which will continue matching the same statement for the target state.

4.4. Specification parser

A simplified version of the grammar for parsing the specification is shown below in a modified Backus-Naur-Form where the symbols $*, +, ?$ are used as in regular expressions. The implementation is based on the lexer and parser generators ocamllex and ocamlyacc. Tokens - despite being defined in the lexer file - are interspersed in the grammar below. Single- and multi-line comments are supported and already filtered out by the lexer.

⟨*file*⟩ ::= ⟨*definition*⟩ EOL `/* definitions are seperated by line breaks */`
| ⟨*definition*⟩ EOF
| EOL `/* end of line */`
| EOF `/* end of file */`

⟨*word*⟩ ::= `[_0-9a-zA-Z]`

⟨*identifier*⟩ ::= `[_a-zA-Z]` ⟨*word*⟩* `/* e.g. foo, _foo, _1, but not 1a */`

⟨*ws*⟩ ::= `[ \t]` `/* whitespace: space or tab */`

⟨*string*⟩ ::= ... `/* single- or double-quoted, backslash escapes */`

⟨*node*⟩ ::= ⟨*word*⟩ ⟨*ws*⟩+ ⟨*string*⟩

⟨*edge*⟩ ::= ⟨*word*⟩ ⟨*ws*⟩* '-' (⟨*word*⟩ (',' ⟨*word*⟩)*)? '>'? '>' ⟨*ws*⟩* ⟨*word*⟩ ⟨*ws*⟩+

⟨*definition*⟩ ::= ⟨*node*⟩
| ⟨*edge*⟩ ⟨*stmt*⟩

⟨*stmt*⟩ ::= ⟨*var*⟩ '=' ⟨*expr*⟩
| ⟨*expr*⟩

⟨*key*⟩ ::= '$' ⟨*word*⟩

⟨*var*⟩ ::= ⟨*key*⟩
| ⟨*identifier*⟩

⟨*regex*⟩ ::= 'r' ⟨*string*⟩

⟨*arguments*⟩ ::= ⟨*expr*⟩
| ⟨*arguments*⟩ ',' ⟨*expr*⟩

⟨*binop*⟩ ::= '<'
| '>'
| '=='
| '!='
| '<='
| '>='
| '+'
| '-'
| '*'
| '/'

⟨*expr*⟩ ::= '(' ⟨*expr*⟩ ')'
| ⟨*regex*⟩
| ⟨*string*⟩
| ⟨*bool*⟩ `/* true or false */`
| ⟨*var*⟩
| ⟨*identifier*⟩ '(' ⟨*arguments*⟩ ')' `/* function call */`
| '_' `/* wildcard */`
| ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩

The grammar used for the implementation also evaluates numerical expressions and comparisons as far as possible. This has been omitted above for clarity.

4.5. Making the specification more concise

Even for something rather small like the file handle example, the automaton can become very big and hard to read for humans.

In order to avoid redundant parts, forwarding is supported. Forwarding edges are displayed as dotted lines in the generated graphs and can also contain constraints. If such an edge is taken, the current input is again evaluated in the target state.

Another feature are wildcards. In each state pattern matching is done on the constraints and the transistion of the first matching constraint is taken. Wildcards can be used anywhere, e.g. as a function argument or as a last constraint which always matches.

5. Example use cases

5.1. File handles redux

Listing 5.1 shows a specification for file handles like implemented in Chapter 3. It is optimistic about `fopen`, i.e. there are no warnings for missing success checks.

```
w1 "file handle is not saved!"
w2 "closeing unopened file handle $"
w3 "writing to unopened file handle $"
w4 "writing to read-only file handle $"
w5 "closeing already closed file handle $"
w6 "writing to closed file handle $"
w7 "overwriting still opened file handle $"
w8 "unrecognized file open mode for file handle $"

1           -w1> 1           fopen(_)
1           -w2> 1           fclose($fp)
1           -w3> 1           fprintf($fp, _)

1           -> open_read     $fp = fopen(_, "r")
1           -> open_write    $fp = fopen(_, r"[wa]") // regex, see
    OCaml doc for details
1           -w8> 1           $fp = fopen(_, _)

open_read  -w4> open_read    fprintf($fp, _)

open_read  -w7>> 1           $fp = fopen(_, _)
open_write -w7>> 1           $fp = fopen(_, _)

open_read  -> closed         fclose($fp)
open_write -> closed         fclose($fp)

closed     -w5> closed       fclose($fp)
closed     -w6> closed       fprintf($fp, _)
closed     ->> 1             _ // let state 1 handle the rest

// setup which states are end states
end: 1, closed
// warning for all entries that are not in an end state
!end "file is never closed"
!end@return "unclosed files: $"
```

Listing 5.1: An optimistic specification for file handle usage

The resulting graph can be seen in Figure 5.1. While it might be helpful for visualizing relations between states, it fails to encompass the order of constraints, which is relevant for pattern matching.

To make the graph less terse, the warnings of reflexive edges (all but `w7`) are displayed separately and thought to have implicit back edges.

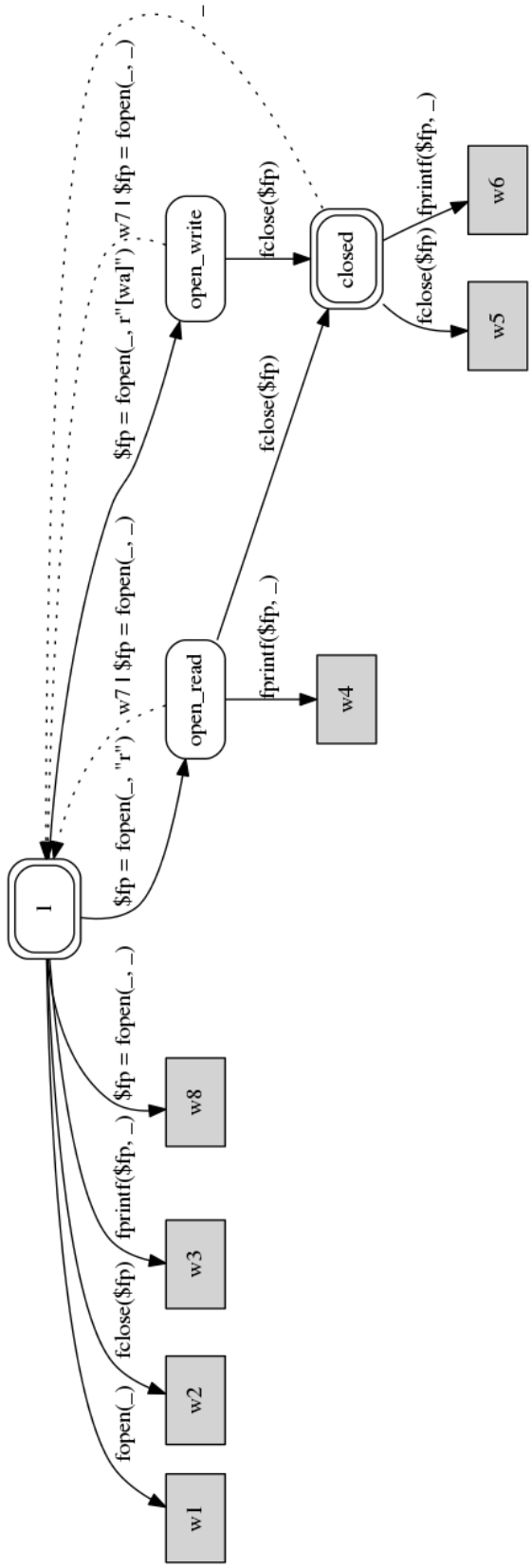


Figure 5.1.: Automaton for optimistic file handle usage

To make the analysis aware of possibly failing open operations, we have to extend it as shown in Listing 5.2, where `branch(exp, b)` serves as a special function to split the analysis. For every start node that should be split there must be two definitions with the same expression `exp`, different target nodes and `b` set to `true` and `false` respectively.

```
...
// go to unchecked states first
1           -> u_open_read  $fp = fopen(_, "r")
1           -> u_open_write $fp = fopen(_, r"[wa]") // regex, see
    OCaml doc for details
1           -w8> 1          $fp = fopen(_, _)

// define possible branches
u_open_read  -> 1           branch($fp==0, true)
u_open_read  -> open_read   branch($fp==0, false)
u_open_write -> 1           branch($fp==0, true)
u_open_write -> open_write  branch($fp==0, false)
...
```

Listing 5.2: Check for return value of `fopen`

5.2. Dynamic memory allocation

Besides statically and automatically managed memory, C offers dynamically allocated memory to allow data structures whose size can be set at run-time and to give more flexibility of their lifetime to the programmer. While automatically managed variables are kept on the stack, dynamically allocated memory is kept in the heap and accessed via pointers.

Table 5.1 lists the functions for dynamic memory allocation defined in `stdlib.h`. The allocation functions return a pointer to allocated space or a null pointer if allocation failed. Proper usage of these functions is critical: first allocate memory, handle potential error,

Function	Description
`void *malloc(size_t size)`	allocates `size` bytes
`void *realloc(void *ptr, size_t size)`	changes size of memory block, behaves like `malloc` if `ptr` is a null pointer
`void *calloc(size_t nmemb, size_t size)`	allocates space for an array of `nmemb` objects, each of `size` and initialize to all bits zero
`void free(void *ptr)`	deallocates space, no action for null pointer

Table 5.1.: Functions for dynamic memory allocation [6]

work with the memory and finally free the memory. Although there are only a few functions involved, dynamic memory allocation is a common source of bugs:

Missing success check The allocation functions are not guaranteed to succeed (e.g. no more memory available to process). Usage of the returned null pointer in case of an error leads to a segemntation fault.

Memory leaks Allocated memory that is not freed can no longer be used by the program, which wastes resources and can escalate to the point where every allocation fails because no more memory is left.

Double free Freeing already freed memory crashes the program with a double free error due to memory corruption.

Use of dangling pointers Dereferencing of a null pointer always leads to a segmentation fault because it has neither read nor write permissions. Using other dangling pointers (e.g. not allocated or already freed memory) is undefined behavior which may result in a segmentation fault or more subtle errors during further run-time of the program.

Listing 5.3 shows an example program and Listing 5.4 a specification for the correct usage of `malloc` (other allocation functions similar) and `free`. See Figure 5.2 for its graph. This is only sound as long as there are no other external functions used for allocating or freeing memory!

```c
#include <stdlib.h>
#include <stdio.h>

int main(){
  int *ip;
  //*ip = 5; // write: segfault
  //printf("%i", *ip); // read: segfault
  ip = malloc(sizeof(int)); // allocate memory
  if(ip == NULL) return 1; // success check

  *ip = 5; // work with memory

  free(ip); // free memory
  //free(ip); // crash: double free or corruption
  *ip = 5; // undefined but no crash
  printf("%i", *ip); // undefined but prints 5
  ip = NULL; // make sure the pointer is not used anymore
  //*ip = 5; // segfault
}
```

Listing 5.3: An example program using dynamic memory allocation with `malloc` and `free`

```
w1 "pointer is not saved [leak]"
w2 "freeing unallocated pointer $ [segfault?]"
w3 "writing to unallocated pointer $ [segfault?]"
w4 "overwriting unfreed pointer $ [leak]"

1        -w1> 1        malloc(_)
1        -w2> 1        free($p)
```

```
1          -w3> 1          *$p = _
1          ->   u_alloc    $p = malloc(_)

u_alloc    ->   1          branch($p==0, true)
u_alloc    ->   alloc      branch($p==0, false)

alloc      -w4> alloc      $p = malloc(_)
alloc      ->   freed      free($p)

freed      ->> 1           _ // let state 1 handle the rest

// setup which states are end states
end: 1, freed
// warning for all entries that are not in an end state
!end "pointer is never freed"
!end@return "unfreed pointers: $"
```

Listing 5.4: A specification for dynamic memory allocation with `malloc` and `free`

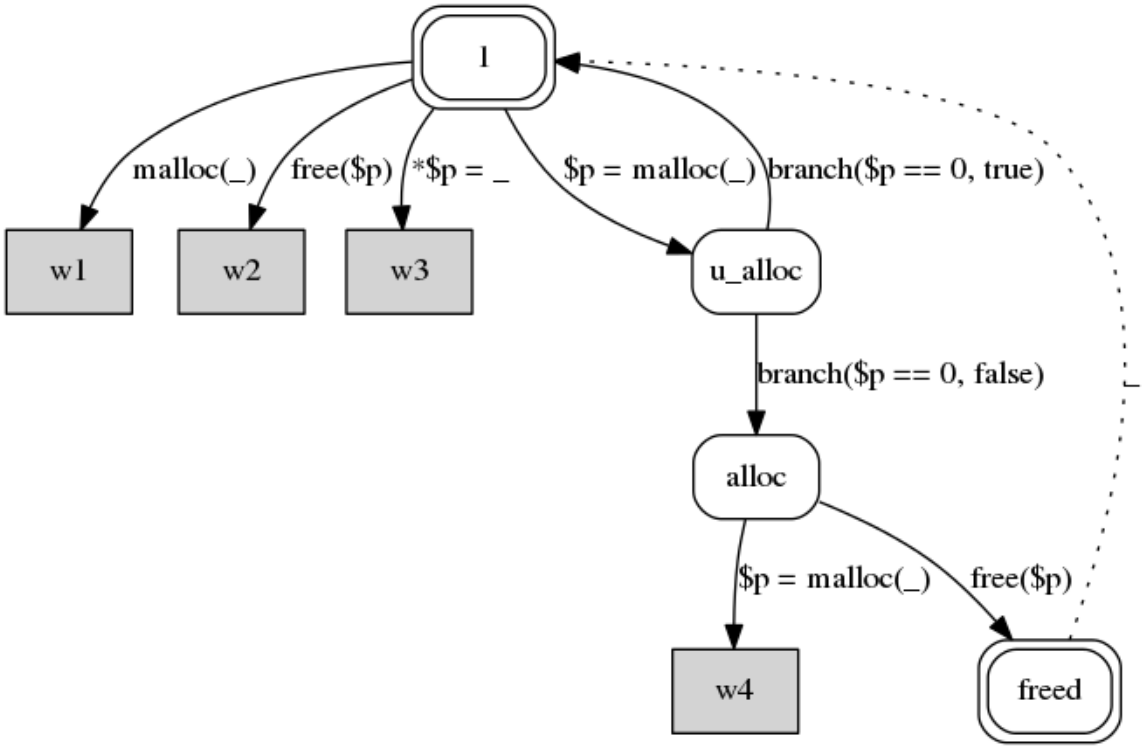


Figure 5.2.: Automaton for dynamic memory allocation with `malloc` and `free`

6. A web frontend

Although Goblint can be used entirely on the command line and also offers HTML output, a more integrated workflow for development and testing is preferable. A screenshot of the developed web frontend is shown in Figure 6.1. The main focus is on allowing easy use

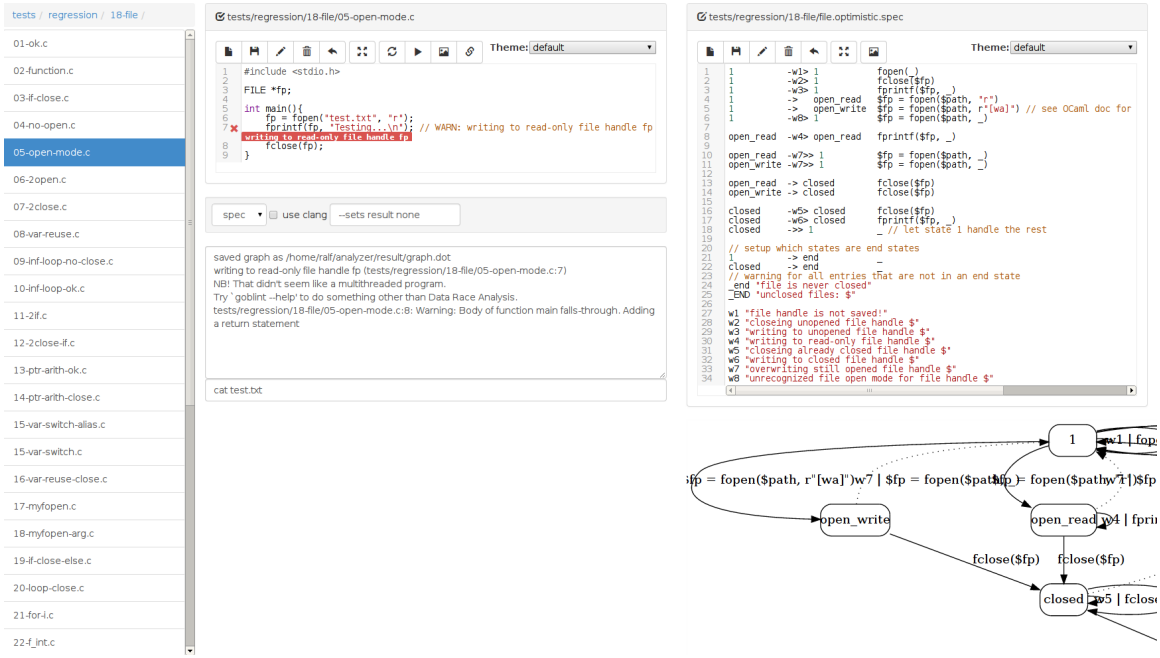


Figure 6.1.: A web frontend for Goblint

of the two developed analyses, but it can also be used for others.

On the left the local file system can be traversed and files can be selected. The editor on the left contains the selected C-file, the one on the right the specification file. Below the C-file editor there are options for the currently selected analysis. There are two presets: *file* for file handle analysis and *spec* for the specification analysis. The right half of the user interface is only shown if *spec* is selected.

The output window below the options is used for Goblint, compiler warnings and the output when running the program. Together with the prompt below it can also be used as a simple shell.

Warnings are displayed directly inside the editor: those that must be true are red, those that may be true are orange (see Figure 6.2). The specification is visualized below the editor as a graph. Both the warnings and the graph update when typing inside the editor. This allows to conveniently develop specifications and play around.

Both editors offer basic file controls as shown in Figure 6.3. The first two button groups

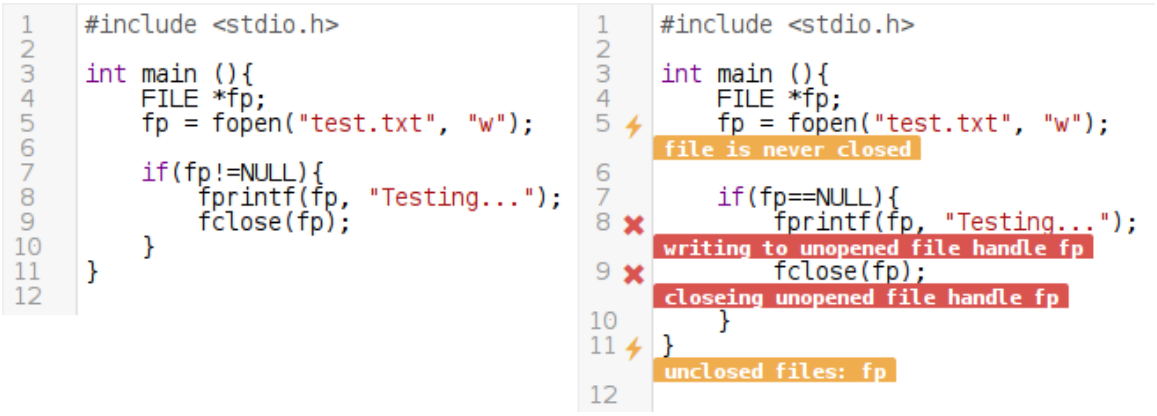


Figure 6.2.: Warnings after changing one character

exist in both editors. From left to right: new file, save, rename, delete, revert to version in git, fullscreen. Further button groups can be added for each editor. In this case specific for C-files: rerun the analysis (also happens upon typing), compile and run the program in a temporary directory, open an image of the control flow graph, open Goblint's HTML output.

Figure 6.3.: Web frontend controls for C-files

7. Conclusion and future work

The thesis started by introducing the basic concepts of abstract interpretation as used by the static analyzer Goblint. On top of this framework we first developed an analysis to verify proper usage of file handles, which then served as a starting point for the development of a generalized analysis that could do the same using a custom specification language. Regular properties can be verified with this language by describing an automaton with constraints and warnings on transitions between states. The specification language was then used to redo the manually implemented file handle analysis and verify basic dynamic memory allocation. Finally a web frontend was developed that helps with the creation of specification files by detecting errors and live visualization of the automaton. It also allows a test-driven approach as the result of the analysis is updated on each modification.

As a comparison: the specification for file handles is 42 lines, while the manual implementation is 642 lines. The specification code however is about 1159 lines.

Concerning limitations, one has to keep in mind that the programmer has full memory access in C programs. This leaves two possibilities when designing a specification:

1. assume that the analyzed state is not influenced by other means than specified (e.g. no other functions or direct memory manipulation) or

2. go to an error state for every unknown operation, which is only feasible if the set of valid statements is very small.

Therefore further work could be done in order to extend the specifications so that e.g. the file handle analysis takes into account all the functions from `stdio.h`.

Combining multiple analyses into one specification file is possible, yet a modular solution which allows to give a list of specification files would be better suited.

Currently only statements with functions and assignments can be verified with the specification. Although this is enough for the most common API usage problems, the query system offers much more information which could be incorporated (e.g. values of expressions).

Another obvious limitation is that only regular safety properties can be verified. Counting semaphores and recursive mutexes could not be handled since there is no way of keeping track of the count. Therefore supporting non-regular safety properties (e.g. by using push down automata) would be a useful extension.

Appendix

A. Setup

Install opam[1], then do

```
opam install ocamlfind camomile batteries cil xml-light
```

to install the latest versions of the dependencies for the current user.

After that you can build goblint:

```
git clone https://github.com/vogler/analyzer.git
cd analyzer
make
```

If something goes wrong, take a look at `scripts/travis-ci.sh` for an example setup or try the versions listed in `INSTALL`.

Alternatively you can use your system's package manager to install the dependencies globally or use `scripts/install_script.sh` to build everything from source without affecting any existing OCaml installation.

Virtual machine A ready-to-use virtual machine can be started and connected to using Vagrant [2]:

```
vagrant up  # download and provision VM
vagrant ssh # connect over ssh
sudo su -   # change to root
cd analyzer # goblint is ready here
```

Web frontend In order to setup the web frontend, make sure Node.js[3] is installed and do:

```
git submodule update --init --recursive # fetch code
cd webapp
sudo npm install -g coffee-script nodemon bower # install those
    globally if not already installed (nodemon and bower are optional)
npm install # locally install node and bower dependencies
```

Then run it using `coffee server.coffee` or `nodemon server.coffee` for automatic reloading during development.

A JavaScript version can be compiled using `coffee -c server.coffee`.

[1] http://opam.ocamlpro.com/
[2] http://www.vagrantup.com/
[3] http://nodejs.org/

B. Usage

B.1. Command-line options

```
Usage: goblint [options] source-files
Options
    -v                          Prints more status information.
    -o <file>                   Prints the output to file.
    -I <dir>                    Add include directory.
    -IK <dir>                   Add kernel include directory.

    --help                      Prints this text
    --version                   Print out current version information.

    --conf <file>               Merge the configuration from the <file>.
    --writeconf <file>          Write the effective configuration to <file>
    --set <jpath> <jvalue>      Set a configuration variable <jpath> to the specified <jvalue>.
    --sets <jpath> <string>     Set a configuration variable <jpath> to the string.
    --enable  <jpath>           Set a configuration variable <jpath> to true.
    --disable <jpath>           Set a configuration variable <jpath> to false.

    --print_options             Print out commonly used configuration variables.
    --print_all_options         Print out all configuration variables.

A <jvalue> is a string from the JSON language where single-quotes (') are used instead of double-quotes (").

A <jpath> is a path in a json structure. E.g. 'field.another_field[42]';
in addition to the normal syntax you can use 'field[+]' append to an array.
```

B.2. Generating a control flow graph

In order to generate a control flow graph for the program `test.c`, do the following:

```
./goblint --set justcfg true test.c
```

```
dot -Tpng cfg.dot -o cfg.png
```

B.3. Using the implemented analyses

B.3.1. File handles

To use the file analysis on `test.c` with HTML output in `./result`:

```
./goblint --sets ana.activated[0][+] file --sets result html test.c
```

The analysis can be configured to be optimistic about opening files. If this option is set to `true`, calls to `fopen` will be assumed to never fail:

```
./goblint --sets ana.activated[0][+] file --set ana.file.optimistic
    true --sets result html test.c
```

B.3.2. Specification

To use the spec analysis on `test.c` with specification file `test.spec`:

```
./goblint --sets ana.activated[0][+] spec --sets ana.spec.file test.
    spec --sets result html test.c
```

Parser The specification parser can be built and tested independent of Goblint. The following script compiles the parser and runs it on specification file `$spec`. On failure, it starts the parser again in REPL-mode for debugging.

```
bin=src/mainspec.native
spec=${1-"src/spec/file.spec"}
ocamlbuild -yaccflag -v -X webapp -no-links -use-ocamlfind $bin \
    && (./_build/$bin $spec \
        || (echo "$spec failed, running interactive now...";
            rlwrap ./_build/$bin
           )
       )
```

Graph The parser saves a graph representing the specification to `result/graph.dot` by default (see `src/spec/specUtil.ml`). To generate an image, the following could be used:

```
dot -Tpng result/graph.dot -o graph.png
```

Bibliography

[1] *Goblint: Path-sensitive data race analysis*, volume 30, 2009.

[2] Source lines of code. `http://en.wikipedia.org/wiki/Source_lines_of_code`, 2011.

[3] The Goblint Analyzer. `http://goblint.in.tum.de/`, 2011.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[5] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22:84–, March 1997.

[6] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[7] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, January 1976.

[8] John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[9] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[10] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[11] H. Seidl, R. Wilhelm, and S. Hack. *Übersetzerbau 3: Analyse und Transformation*. Übersetzerbau. Springer, 2009.

[12] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[13] Vesal Vojdani. *Static Data Race Analysis of Heap-Manipulating C Programs*. PhD thesis, University of Tartu., December 2010.