The CProver User Manual

SatAbs – Predicate Abstraction with SAT

CBMC – Bounded Model Checking

Contents

1 Introduction

1.1 Motivation

Correctness of computer systems is critical in today's information society. Modern computer systems consist of both hardware and software components. The complexity of the software embedded in devices we use everyday has risen dramatically, and correctness of such embedded software is often a bigger problem than that of the underlying hardware. This is especially true of software that runs on computers controlling our transportation and communication infrastructure. Examples of serious software errors are easy to find.

Manual inspection of complex software is infeasible and costly, so tool support is in dire need. Many tools rely on engineers to provide test-vectors to uncover design flaws. Formal verification, on the other hand, is automated, and tools that implement it can check the behavior of a design for *any* vector of inputs.

Numerous tools to hunt down functional design flaws in silicon have been available for many years, mainly due to the enormous cost of hardware bugs. The use of such tools is wide-spread. In contrast, the market for tools that address the need for quality software is still in its infancy.

Research in software quality has an enormous breadth, and due to space restrictions, we focus the presentation using two criteria:

1. We believe that any form of quality requires a specific *guarantee*, in theory and practice.
2. The sheer size of software designs requires techniques that are highly *automated*.

In practice, quality guarantees usually do not refer to 'total correctness' of a design, as ensuring the absence of all bugs is too expensive for most applications. In contrast, a guarantee of the absence of specific flaws is achievable, and is a good metric of quality.

This manual documents two programs that try to achieve formal guarantees of the absence of specific problems: CBMC and SatAbs. The algorithms implemented by CBMC and SatAbs are complementary, and often, one tool is able to solve a problem that the other cannot solve.

Both CBMC and SatAbs are verification tools for ANSI-C/C++ programs. They verify array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. Both tools model integer arithmetic accurately, and are able to reason about machine-level artifacts such as integer overflow. CBMC and SatAbs are therefore able to detect a class of bugs that has so far gone unnoticed by many formal verification tools.

1.2 Bounded Model Checking with CBMC

CBMC implements a technique called *Bounded Model Checking* (BMC). In BMC, the transition relation for a complex state machine and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using an efficient SAT procedure. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure. If the formula is not satisfiable, the program can be unwound more to determine if a longer counterexample exists.

In many engineering domains, real-time guarantees are a strict requirement. An example is software embedded in automotive controllers. As a consequence, the loop constructs in these types of programs often have a strict bound on the number of iterations. CBMC is able to formally verify such bounds by means of *unwinding assertions*. Once this bound is established, CBMC is able to prove the absence of errors.

A more detailed description of how to apply CBMC to program verification is in Chapter 3.

→ 3

1.3 Automatic Program Verification with SatAbs

In many cases, lightweight properties such as array bounds do not rely on the entire program. A large fraction of the program is *irrelevant* to the property. SatAbs exploits this observation and computes an *abstraction* of the program in order to handle large amounts of code.

In order to use SatAbs it is not necessary to understand the abstraction refinement process. For the interested reader, a high-level introduction is provided in Chapter 4.1.

→ 4.1

Just as CBMC, SatAbs attempts to build counterexamples that refute the property. If such a counterexample is found, it is presented to the engineer to facilitate localization and repair of the program.

1.4 A Short Tutorial

In order to give a brief overview of the capabilities of SatAbs we start with a series of small examples. The description of the installation of the tool is postponed to Chapter 2.

→ 2

The issue of buffer overflows has brought wide public attention. A buffer is a contiguous allocated chunk of memory, represented by an array or a pointer in C. Programs written in C do not provide automatic bounds checking on the buffer, which means a program can – accidentally or maliciously – write past a buffer. The following example is a perfectly valid C program (in the sense that a compiler compiles it without any errors):

```
int main() {
    int buffer[10];
    buffer[20] = 10;
}
```

However, the write access to an address outside the allocated memory region can lead to unexpected behavior. In particular, such bugs can be exploited

to overwrite the return address of a function, thus enabling the execution of arbitrary user induced code. SATABS is able to detect this problem and reports that the "upper bound property" of the buffer is violated. SATABS is capable of checking the lower and upper bounds, even for arrays with dynamic size.

1.5 Hardware/Software Co-Verification

Software programs often interact with hardware in a non-trivial manner, and many properties of the overall design only arise from the interplay of both components. CBMC and SATABS therefore support *Co-Verification*, i.e., are able to reason about a C/C++ program together with a circuit description given in Verilog.

These co-verification capabilities can also be applied to perform refinement proofs. Software programs are often used as high-level descriptions of circuitry. While both describe the same functionality, the hardware implementation usually contains more detail. It is highly desirable to establish some form for equivalence between the two descriptions.

→ 8

Hardware/Software co-verification and equivalence checking with CBMC and SATABS are described in Chapter 8.

2 Installation

This Chapter provides step-by-step installation instructions for CBMC and SatAbs. CBMC and SatAbs are command line tools, but a graphical user interface is also available.

Both CBMC and SatAbs require a code pre-processing environment comprising of a suitable preprocessor and an a set of header files. These programs and files usually come with a compiler.

In addition to that, the SatAbs verification system relies on an additional model checker for the abstract models. It is modular and can be used with different model checking tools. Currently, SatAbs supports the model checkers Cadence SMV, NuSMV, CMU SMV, SPIN, and BOPPO.

The license for CBMC and SatAbs is provided in Chapter A.

Note that the Windows and Linux x86 binaries are also contained in the Eclipse plugin. Therefore, if you intend to run CBMC or SatAbs exclusively within Eclipse, you can skip the installation of the command line tools. However, you still have to install a model checking tool as described below.

2.1 Installing CBMC

1. Download CBMC. The binaries are available from:

   `http://www.cs.cmu.edu/~modelcheck/cbmc/`

   The Windows and Linux x86 binaries are also contained in the Eclipse plugin for CBMC. Therefore, if you intend to run CBMC exclusively within Eclipse, you can skip the extraction of the binaries from the archive.

2. Unzip/untar the archive in a directory of your choice. We recommend to add this directory to your `PATH`. The Windows version of CBMC requires the preprocessor `cl.exe`, which is part of Visual Studio. The path to `cl.exe` must be part of the `PATH` environment variable of your system.

2.2 Installing SatAbs

1. Download SatAbs. The binaries are available from:

   `http://www.cprover.org/satabs/`

   The Windows and Linux x86 binaries are also contained in the Eclipse plugin for SatAbs. Therefore, if you intend to run SatAbs exclusively within Eclipse, you can skip the extraction of the binaries from the archive. You still have to install a model checking tool as described below.

2. Unzip/untar the archive in a directory of your choice. We recommend to add this directory to your `PATH`. The Windows version of SatAbs requires the preprocessor `cl.exe`, which is part of Visual Studio. The path to `cl.exe` must be part of the `PATH` environment variable of your system.

3. You need to install a Model Checker in order to be able to run SatAbs. You can choose between following alternatives:

   (a) Cadence SMV. Available from

   `http://www.kenmcmil.com/smv.html`

   Cadence SMV is a commercial model checker. The free version that is available on the homepage above must not be used for commercial purposes (read the license agreement thoroughly before you download the tool). The documentation for SMV can be found in the directory where you unzip/untar SMV under `./smv/doc/smv/`. We recommend to add the `smv` binary (located in `./smv/bin/` relative to the path where you unpacked it) to your `PATH`. SatAbs uses Cadence SMV by default.

   (b) NuSMV. Available from

   `http://nusmv.irst.itc.it/`

   NuSMV is the open source alternative to SMV. Installation instructions and documentation can be found on the NuSMV hompeage. We recommend to use NuSMV on platforms where SMV is not available (e.g., MAC OS X). Again, the NuSMV binary should be added to your `PATH`. Use the option `--modelchecker nusmv` to instruct SatAbs to use NuSMV.

   (c) BOPPO. Available from

   `http://www.cprover.org/satabs/`

   BOPPO is a model checker that uses SAT-solving algorithms. BOPPO relies on a built-in SAT solver and Quantor, a solver for quantified boolean formulas that is currently bundled with BOPPO, but also available separately from

   `http://fmv.jku.at/quantor/`

   We recommend to add both tools to your `PATH`. By default, SatAbs uses the Cadence SMV model checker. Use the option `--modelchecker boppo` when you call SatAbs and want it to use BOPPO instead of SMV.

   (d) SPIN. Available from

   `http://spinroot.com/spin/whatispin.html`

   SPIN is an explicit state model checker. Since predicate abstraction generates abstract programs with many uninitialized variables, explicit state model checking algorithms do not scale very well for this purpose. We recommend to use one of the symbolic model checking tools mentioned above.

→ 1.4

4. Now you can execute SatAbs. Try running `satabs` on the small examples presented in the tutorial (Section 1.4). If you use the SMV model checker, the only parameters you have to specify are the names of the files that contain your program.

2.3 Installing the Eclipse Plugin

As mentioned above, we provide a graphical user interface, which is realized as a plugin to the Eclipse framework. Eclipse is available at `http://www.eclipse.org`. We do not provide installation instructions for Eclipse (basically, you only have to download the current version and extract the files to your hard-disk) and assume that you have already installed the current version. You need version 3.2 or better; the plugin does not work with version 3.1. In case you are running Windows, make sure that the path containing the Visual Studio is in the PATH environment variable.

To install the Eclipse SatAbs plugin, perform following steps:

1. In Eclipse, open the menu *Help→ Software Updates→Find and Install*.
2. Select the radio button "Search for new features to install".
3. In the window that pops up, select "New remote site" and enter the URL `http://www.cprover.org/satabs/plugin/lin/` (for Linux) or `http://www.cprover.org/satabs/plugin/win/` (for Windows) or `http://www.cprover.org/satabs/plugin/osx/` (for MacOS X) into the URL field. Provide a name for the SatAbs update site, e.g., SatAbs plugin (see Figure 2.1).

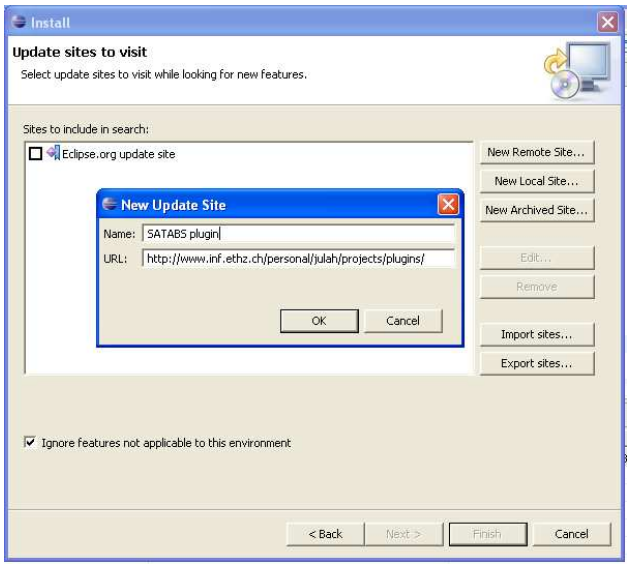

Figure 2.1: Installing the Eclipse plugin for SatAbs

4. Select the newly added update site, and press "Finish".

5. Select the feature `org.feature.CProver_version`, and clock "Next". Read the license thoroughly before you agree (see also Chapter A), and install the plugin by clocking "Finish". You will see a warning that the plugin is not digitally signed; confirm with "Install". The plugin will be downloaded automatically. It contains the Windows and Linux executables for SATABS. Note that we do not provide the SMV executable, nor the preprocessors!

6. Unless you have already added the model checker of your choice (e.g., SMV) to your `PATH`, you should do so *now*. In the Eclipse window, select the menu point *Windows→Preferences*, choose the `CBMC and SATABS` preferences and add the corresponding `PATH` environment variable.

→ 5 A small tutorial on how to use the Eclipse plugin is provided in Chapter 5.

2.4 Installing goto-cc

1. Download GOTO-CC. The binaries are available from:

   `http://www.cprover.org/goto-cc/`

2. Unzip/untar the archive in a directory of your choice. We recommend to add this directory to your `PATH`. The Windows version of GOTO-CC requires the preprocessor `cl.exe`, which is part of Visual Studio (Express). The path to `cl.exe` must be part of the `PATH` environment variable of your system.

→ 6 Chapter 6 covers the integration of GOTO-CC into build systems.

3 CBMC: Bounded Model Checking for C/C++

3.1 A Short Tutorial

3.1.1 First Steps

Like a compiler, CBMC takes the names of `.c` files as command line arguments. CBMC then translates the program and merges the function definitions from the various `.c` files, just like a linker. But instead of producing a binary for execution, CBMC performs symbolic simulation on the program.

As an example, consider the following simple program, named `file1.c`:

```
int puts(const char *s) { }

int main(int argc, char **argv) {
  int i;

  if(argc>=1)
    puts(argv[2]);
}
```

Of course, this program is faulty, as the `argv` array might have only one element, and then the array access `argv[2]` is out of bounds. Now, run CBMC as follows:

```
cbmc file1.c --show-claims
```

CBMC will print the list of properties it checks. Note that it prints a claim labeled with "array `argv` upper bound" together with the location of the faulty array access. As you can see, CBMC largely determines the property it needs to check itself.[1] Examples for user-specified properties are given in Sec. 7.5. Note that these claims need not necessarily correspond to bugs – these are just *potential* flaws. Whether one of these claims corresponds to a bug needs to be determined by further analysis.

The first step of this analysis is symbolic simulation, which corresponds to a translation of the program into a formula. The formula is then combined with the property. Let's run the symbolic simulation:

```
cbmc file1.c --show-vcc
```

With this option, CBMC performs the symbolic simulation and prints the verification conditions on the screen. A verification condition needs to be proven to be valid in order to assert that the corresponding property holds. Let's run the verification:

[1] This is realized by means of a preliminary static analysis, which relies on computing a fixed point on an abstract domain.

```
cbmc file1.c
```

CBMC transforms the equation you have seen before into CNF and passes it to a SAT solver. It can now detect that the equation is actually not valid, and thus, there is a bug in the program. It prints a counterexample trace, i.e., a program trace that ends in a state which violates the property. In our example, the program trace ends in the faulty array access. It also shows the values the input variables must have for the bug to occur. In this example, `argc` must be one to trigger the out-of-bounds array access. If you change the branch condition in the example to `argc>=2`, the bug is fixed and CBMC will report a successful verification run.

3.1.2 Verifying Modules

In the example above, we used a program that starts with a `main` function. However, CBMC is aimed at embedded software, and these kinds of programs usually have different entry points. Furthermore, CBMC is also useful for verifying program modules. Consider the following example, called `file2.c`:

```
int array[10];

int sum() {
  unsigned i, sum;

  sum=0;

  for(i=0; i<10; i++)
    sum+=array[i];
}
```

In order to set the entry point to the `sum` function, run

```
cbmc file2.c --function sum
```

3.1.3 Loop Unwinding

You will note that CBMC unwinds the `for` loop in the program. As CBMC performs Bounded Model Checking, all loops have to have a finite upper runtime bound in order to guarantee that all bugs are found. CBMC actually checks that enough unwinding is performed. As an example, consider the program `binsearch.c`:

```
int binsearch(int x) {
  int a[16];
  signed low=0, high=16;

  while(low<high) {
    signed middle=low+((high-low)>>1);

    if(a[middle]<x)
      high=middle;
    else if(a[middle]>x)
```

```
      low=middle+1;
    else // a[middle]=x !
      return middle;
  }

  return -1;
}
```

If you run CBMC on this function, you will notice that the unwinding does not stop. The built-in simplifier is not able to determine a run time bound for this loop. The unwinding bound has to be given as a command line argument:

```
cbmc binsearch.c --function binsearch --unwind 5
```

CBMC not only verifies the array bounds (note that this actually depends on the result of the right shift), but also checks that enough unwinding is done, i.e., it proves a run-time bound. For any lower unwinding bound, there are traces that require more loop iterations. Thus, CBMC will produce an appropriate counterexample.

3.1.4 Unbounded Loops

However, CBMC can also be used for programs with unbounded loops. In this case, CBMC is used for bug hunting only; CBMC does not attempt to find all bugs. Consider the following program:

```
_Bool nondet_bool();
_Bool LOCK = 0;

_Bool lock() {
  if(nondet_bool()) {
    assert(!LOCK);
    LOCK=1;
    return 1; }

  return 0;
}

void unlock() {
  assert(LOCK);
  LOCK=0;
}

int main() {
  unsigned got_lock = 0;
  int times;

  while(times > 0) {
    if(lock()) {
      got_lock++;
      /* critical section */
    }
```

```
    if(got_lock!=0)
      unlock();

    got_lock--;
    times--;
} }
```

The `while` loop in the `main` function has no (useful) run-time bound. Thus, the –unwind parameter has to be used in order to prevent infinite unwinding. However, you will note that CBMC will detect that not enough unwinding is done and aborts with an unwinding assertion violation.

In order to disable this test, run CBMC with the parameter

```
--no-unwinding-assertions
```

For an unwinding bound of one, no bug is found. But already for a bound of two, CBMC detects a trace that violates an assertion. Without unwinding assertions, CBMC does not prove the program correct, but it can be helpful to find program bugs.

3.1.5 A Note About Compilers and the ANSI-C Library

Most C programs make use of functions provided by a library; instances are functions from the standard ANSI-C library such as `malloc` or `printf`. The verification of programs that use such functions has two requirements:

1. Apppropriate header files have to be provided. These header files contain *declarations* of the functions that are to be used.
2. Appropriate *definitions* have to be provided.

Most C compilers come with header files for the ANSI-C library functions. We briefly discuss how to obtain/install these library files.

Linux Linux systems that are able to compile software are usually equipped with the appropriate header files. Consult the documentation of your distribution on how to install the compiler and the header files. First try to compile some siginificant program before attempting to verify it.

Windows On Microsoft Windows, CBMC/SATABS are preconfigured to use the compiler shipped with Microsoft's Visual Studio. Visual Studio Express is sufficient, and is available for download for free from the Microsoft webpage. Visual Studio installs the usual set of header files together with the compiler. However, Visual Studio requires a large set of environment variables for the compiler to function correctly. It is therefore recommended to run CBMC or SATABS from the "Visual Studio Command Prompt", which can be found in the menu "Visual Studio Tools".

Note that in both cases, only header files are available. CBMC/SATABS only come with a small set of definitions, which includes functions such as `malloc`. Detailed information about the built-in definitions is in Chapter 6.

→ 6

3.2 Command Line Interface

This section describes the command line interface of CBMC. Like a C compiler, CBMC takes the names of the `.c` source files as arguments. Additional options allow to customize the behavior of CBMC.

Option	Description
`-I path`	set include path (C/C++)
`-D macro`	define preprocessor macro (C/C++)
`--program-only`	only show program expression
`--function` name	set main function name
`--all-claims`	keep all claims
`--unwind` nr	unwind nr times
`--unwindset` nr	unwind given loop nr times
`--show-claims`	only show claims
`--dimacs`	generate CNF in DIMACS format
`--document-subgoals`	generate subgoals documentation
`--slice`	remove unrelated assignments
`--no-assertions`	ignore assertions
`--no-unwinding-assertions`	do not generate unwinding assertions
`--no-bounds-check`	do not do array bounds check
`--no-div-by-zero-check`	do not do division by zero check
`--no-pointer-check`	do not do pointer check
`--bound` nr	number of transitions
`--beautify-greedy`	beautify the counterexample (greedy heuristic)
`--beautify-pbs`	beautify the counterexample (PBS)
`--cvc`	output subgoals in CVC syntax
`--smt`	output subgoals in SMT syntax
`--outfile` Filename	output subgoals to given file
`--16`, `--32`, `--64`	set width of machine word
`--little-endian`	allow little-endian word-byte conversions
`--big-endian`	allow big-endian word-byte conversions
`--show-symbol-table`	show symbol table
`--show-goto-functions`	show goto program
`--floatbv`	use genuine IEEE floating point arithmetic
`--ppc-macos`	set MACOS/PPC architecture
`--i386-macos`	set MACOS/I386 architecture
`--i386-linux`	set Linux/I386 architecture (default)
`--no-arch`	don't set up an architecture
`--arrays-uf-none`	never turn arrays into uninterpreted functions
`--arrays-uf-always`	always turn arrays into uninterpreted functions
`--interpeter`	do concrete execution

Structured output can be obtained from CBMC using the option –xml-ui. Any output from CBMC (e.g., counterexamples) will then use an XML representation.

4 Verifying C/C++ Programs with SatAbs

4.1 Background

This section provides background information on how SatAbs operates. The reader who is only interested in running SatAbs may skip to the next section.

Even for very trivial C programs it is impossible to exhaustively examine their state space (which is potentially unbounded). However, not all details in a C program necessarily contribute to a bug, so in theory it is sufficient to examine the parts of the program that are somehow related to a bug. In practice, many static verification tools (such as `lint`) try to achieve this goal by applying heuristics. This approach comes at a cost: bugs might be overlooked because the heuristics do not cover all relevant aspects of the program. Therefore, the conclusion that a program is correct whenever such a static verification tool is unable to find an error is invalid.

4.1.1 Abstraction and Refinement

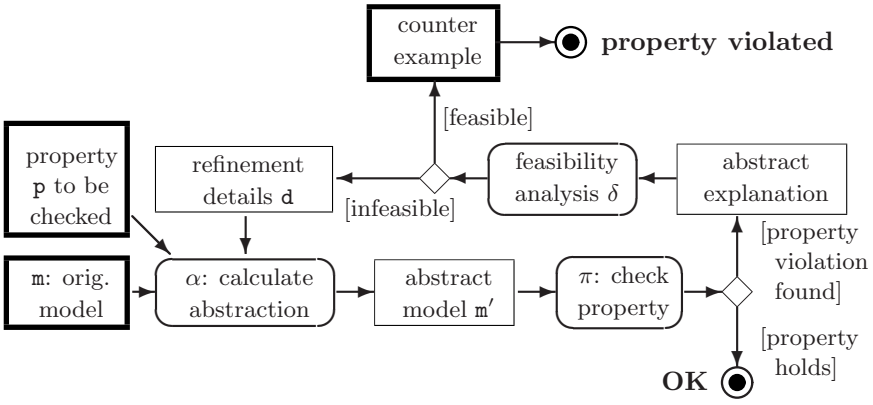

Figure 4.1: Counterexample-Guided Abstraction Refinement Scheme

A more sophisticated approach that has been very successful recently is to generate a *sound* abstraction of the original program (see Figure 4.1). In this context, *soundness* refers to the fact that the abstract program contains (at least) all relevant behaviours (i.e., bugs) that are present in the original program. In Figure 4.1, the component labelled α is responsible for stripping details from the original program. The number of possible behaviours increases as the number of details in the abstract program decreases. Intuitively, the reason is that whenever the model checking tool lacks the information that is necessary to make an accurate decision on whether a branch

of an control flow statement can be taken or not, both branches have to be considered. In the abstract program, a set of concrete states is subsumed by means of a single abstract state. Consider Figure 4.2: The concrete states x_1 and x_2 are mapped to an abstract state X, and similarly Y subsumes y_1 and y_2. However, all transitions that are possible in the concrete program are also possible in the abstract model. The abstract transition $X \to Y$ summarizes the concrete transitions $x_1 \to y_1$ and $x_1 \to y_2$, and $Y \to X$ corresponds to $y_2 \to x_2$. The behaviour $X \to Y \to X$ is feasible in the original program, because it maps to $x_1 \to y_2 \to x_2$. However, $Y \to X \to Y$ is feasible only in the abstract model.

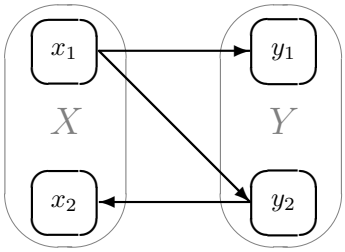

Figure 4.2: An Example Mapping from Concrete States to Abstract States

The consequence is that the model checker (labeled as π in Figure 4.1) possibly reports a *spurious* counterexample. We call a counterexample spurious whenever it is feasible in the current abstract model but not in the original program. However, whenever the model checker π is unable to find an execution trace that violates the given property `p` we can conclude that there is no such trace in the original program, either.

The feasibility of counterexamples is checked by symbolic simulation (performed by component δ in Figure 4.1). If the counterexample is indeed feasible, SatAbs found a bug in the original program and reports it to the user.

Infeasible counterxamples (that originate from abstract behaviours that result from the omission of details and are not present in the original program) are never reported to the user. Instead, the information is used in order to refine the abstraction such that the spurious counterexample is not part of the refined model anymore. For instance, the reason for the infeasibility of $Y \to X \to Y$ in Figure 4.2 is that neither y_1 nor y_2 can be reached from x_2. Therefore, the abstraction can be refined by partitioning X.

The refinement steps are illustrated in Figure 4.3. The first step (1) is to generate a very coarse abstraction with a very small state space. This abstraction is then successively refined $(2, 3, \dots)$ until either a feasible counterexample is found or the abstract program is detailed enough to show that there is no path that leads to a violation of the given property. The problem is that this point is not necessarily reached for every input program, i.e., it is possible that the the abstraction refinement loop never terminates. Therefore, SatAbs allows to specify an upper bound for the number of iterations.

! → Note that whenever this upper bound is reached and no counterexample was found, that does not necessarily mean that there is none. In this case, you cannot make any conclusions at all with respect to the correctness of the

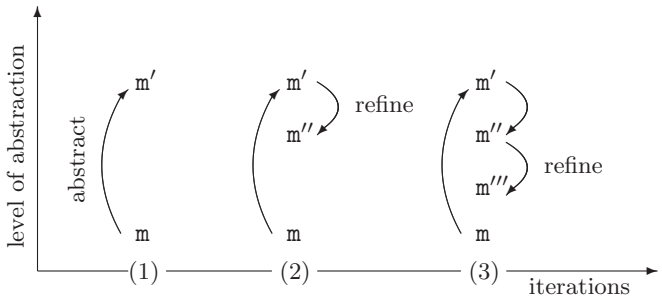

Figure 4.3: Iterative Abstraction Refinement

input program.

4.1.2 Properties

We have mentioned *properties* several times so far, but we never explained *what* kind of properties SatAbs can verify. We cover this topic in more detail in this section. While users of SatAbs almost never have to be concerned about the underlying refinement abstraction algorithms, understanding the classes of properties that can be verified is crucial.

SatAbs allows the verification of following properties:

- **Buffer overflows.** For each array, SatAbs checks whether the upper and lower bounds are violated whenever the array is accessed (i.e., whenever the program reads from or writes to the buffer).
- **Pointer safety.** SatAbs searches for `null`-pointer dereferences.
- **Divison by zero.** SatAbs checks whether there is a path in the program that executes a divison by zero.
- **User specified assertions.** This is the most generic class of supported properties. SatAbs checks for assertion violations. The user can use the `assert` function to specify arbitrary conditions that have to hold at certain points in the program.

(A more detailed list of properties checked by SatAbs can be found in Chapter 7.) All the properties described above are *reachability* properties. They are always of the form "Is there a path such that property ... is violated?". The counterexamples to such properties are always paths. Users of the Eclipse plugin for SatAbs can step through counterexamples in a way that is similar to debugging programs. The installation of this plugin is explained in Section 2.

In general, properties are specified by *predicates*. Examples for such predicates are

- $(i >= 0)$ and $!(i >= \texttt{MAX-1})$. Such predicates are automatically introduced by SatAbs whenever an array `a` of size `MAX` is accessed with index `i` (i.e., whenever `a[i]` occurs in the program). They make sure that the lower and upper array bounds are not violated.

- !($m\%2 == 0$). This predicate is introduced if the user has stated by means of an assertion that `m` has to be odd (i.e., if `assert (m % 2);` occurs in the program).

SatAbs calls these predicates *claims*. Each claim is associated to a specific line of code, i.e., a claim is violated when the predicate can become false at the corresponding program location. Claims are (implicitely) stated within the program: Currently there is no possibility to introduce claims other than by adding assertions to the program. The claims for a program can be inspected by using the `--show-claims` option of SatAbs. SatAbs is able to check several claims at once. However, as soon as a violation of one claim is found, it is reported. A single claim can be verifie by using the `--claim <n>` option of SatAbs, where `<n>` denotes the index of the claim in the list obtained by calling SatAbs with the `--show-claims` flag. Whenever a claim is violated, SatAbs reports a feasible path that leads to a state in which the predicate that corresponds to the violated claim evaluates to false.

! → SatAbs cannot check programs that use functions that are only available in binary (compiled) form[1]. At the moment, (libarary) functions for which no C source code is available have to be replaced by stubs. The usage of stubs and harnesses (as known from unit testing) also allows to check more complicated properties (like, for example, if function `fopen` is always called before `fclose`). This technique is explained in detail in Section 4.3.

4.2 Using the Command Line Interface

This section explains the usage of the command line version of SatAbs. The executable `satabs` is called with a set of (optional) parameters, which are described below, followed by the file(s) that shall be verified. Currently, SatAbs cannot verify the files of the program under test separately. If a program comprises more than one file, then all these files have to be verified in one step by providing all file names in one single command line.

Example:

```
satabs --show-claims file1.c file2.c
```

This call instructs SatAbs to generate and display a list of claims for the program that is contained in `file1.c` and `file1.c`. If, for instance, you wanted to verify the 2^{nd} claim in this list, you could do this by calling

```
satabs --claim 2 file1.c file2.c
```

Verifying claims separately is highly recommended for reasons of scalability.

SatAbs Parameters There are several command line parameters that allow you to control the behaviour of SatAbs. We divide them into two groups: The switches in the first group have an impact on how SatAbs actually behaves. The remaining switches tell SatAbs to provide more information and are therefore useful for debugging or inspecting the verification results. Calling `satabs --help` lists the switches described below.

[1] This restriction is not imposed by the verification algorithms that are used by SatAbs– they also work on assembly code. The reason is simply that so far no assembly language frontend is available for SatAbs.

Parameters to modify behavior

`--16, --32`	The `int` type of the system you run your program on is assumed to have a size of 16 bits (and 32 bits, respectively). This information is necessary in order to model overflows accurately.
`--function` name	Set the name of the main function (i.e., the entry point of the program). By default, `main` will be considered to be the main function.
`--claims <n>`	Verify only a single claim. `<n>` denotes an index into the list of claims that SatAbs prints when called with the parameter `--show-claims`. *It is recommended to verify claims separately in order to avoid scalability problems.*
`--modelchecker` name	Specify the model checker that SatAbs shall use. Currently the values `boppo`, `cadence-smv`, `nusmv`, `cmu-smv`, `spin`, and `satmc` are supported.
`--iterations` #	Allows you to specify the maximum number of refinement iterations. *By default, the number of iterations is 50.*

Parameters to increase verbosity

`--show-claims`	Lists the claims (see Section 4.1.2) that SatAbs will try to verify for the given program.
`--show-goto-program`	Allows you to inspect the "goto-program" that corresponds to your input program. Goto-programs correspond to the original input program, but the function calls have been inlined, and control flow statements have been replaced by conditional jumps ("guareded `gotos`") to basic blocks.
`--show-value-sets`	Show the data used for pointer analysis

Structured output can be obtained from SatAbs using the option –xml-ui. Any output from SatAbs (e.g., counterexamples) will then use an XML representation.

4.3 Tutorials

Similar to unit testing, the model checking approach requires the user to clearly define what parts of the program should be tested. This requirement has following reasons:

- Despite recent advances, the size of the programs that model checkers can cope with is still restricted.
- Typically, you want to verify *your* program and not the libraries that it uses (these are usually assumed to be correct).
- SatAbs cannot verify binary libraries.
- SatAbs does not provide a model for the hardware (e.g., hard disk, input/output devices) the tested program runs on. Since SatAbs is supposed to examine the behaviour of the tested program for *all*

possible inputs and outputs, it is reasonable to model input and output by non-deterministic choice.

This section provides an introduction to model checking "real" C programs with SatAbs. It starts with a small example that is based on Linux device drivers.

4.3.1 Example: Reference Counting in Linux Device Drivers

Microsoft's Slam[2] toolkit has been successfully used to find bugs in Windows device drivers. Slam automatically verifies device driver whether a device driver adheres to a specifications. Slam provides a test harness for device drivers that calls the device driver dispatch routines in a non-deterministic order. Therefore, the model checker examines all combinations of calls. Motivated by the success this approach, we provide an example based on Linux device drivers.

Dynamically loadable modules enable the Linux Kernel to load device drivers on demand and to release them when they are not needed anymore. When a device driver is registered, the kernel provides a major number that is used to uniquely identify the device driver. The corresponding device can be accessed through special files in the filesystem; they are conventionally located in the `/dev` directory. If a process accesses a device file the kernel calls the corresponding `open`, `read` and `write` functions of the device driver. Since a driver must not be released by the kernel as long as it is used by at least one process, the device driver must maintain a usage counter (in modern Linux kernels, this is done automatically, however, drivers that must maintain backward compatibility have to adjust this count).

We provide a skeleton of such a driver. The driver contains following functions:

1. `register_chrdev`: (in `spec.c`, Figure 4.4) Registers a character device. In our implementation, the function sets the variable `usecount` to zero and returns a major number for this device (a constant, if the user provides 0 as argument for the major number, and the value specified by the user otherwise).
2. `unregister_chrdev`: (in `spec.c`, Figure 4.4) Unregisters a character device. This function asserts that the device is not used by any process anymore (we use the macro `MOD_IN_USE` to check this).
3. `dummy_open`: (in `driver.c`, Figure 4.5) This function increases the `usecount`. If the device is locked by some other process `dummy_open` returns -1. Otherwise it locks the device for the caller.
4. `dummy_read`: (in `driver.c`, Figure 4.5) This function "simulates" a read access to the device. In fact it does nothing, since we are currently not interested in the potential buffer overflow that may result from a call to this function. Note the usage of the function `nondet_int` (in line 17): This is an internal SatAbs function that nondeterministically returns an arbitrary integer value. The function `assume` (line 18)

[2] `http://research.microsoft.com/SLAM`

```
1  int usecount;
2  int dummy_major;
3  extern int locked;

4  int register_chrdev (unsigned int major, const char* name)
5  {
6        usecount = 0;
7        if (major == 0)
8             return MAJOR_NUMBER;
9        return major;
10 }

11 int
   unregister_chrdev (unsigned int major, const char* name)
12 {
13       if (MOD_IN_USE) {
14 ERROR:     assert (0);
15       }
16       else
17            return 0;
18 }
```

Figure 4.4: Sources (`spec.c`) for the (un)registering of device drivers

tells SatAbs to ignore all traces that do not adhere to the given assumption. Therefore, whenever the lock is held, `dummy_read` will return a value between 0 and `max`. If the lock is not held then `dummy_read` returns -1.

5. `dummy_release`: (in `driver.c`) If the lock is held, then `dummy_release` decreases the `usecount`, releases the lock, and returns 0. Otherwise, the function returns -1.

We now want to check if any *valid* sequence of calls of the dispatch functions (Figure 4.5) can lead to the violation of the assertion in line 14 in Figure 4.4. Obviously, a call to `dummy_open` that is immediately followed by a call to `unregister_chrdev` violates the assertion. Therefore, we rule out this invalid sequence of calls by ensuring (see line 26 in Figure 4.6) that no device is unregistered while still being locked.

The model checking harness that calls the dispatching functions is shown in Figure 4.6.

The function `main` in `spec.c` gives an example of how these functions are called. First, a character device "`dummy`" is registered. The major number is stored in the `inode` structure of the device. The values for the file structure are assigned non-deterministically. In line 14 the variable `random` is assigned non-deterministically. Subsequently, in line 15, the value of `random` is restricted to be $0 \leq$`random`≤ 3 by a call to `assume`. Whenever the value of `random` is not in this interval, the corresponding execution trace is simply

pruned by SatAbs. Depending on the value of `random`, the harness calls either `dummy_open`, `dummy_read` or `dummy_close`. Therefore, if there is a sequence of calls to these three functions that leads to a violation of the assertion in `unregister_chrdev` in line 28, then SatAbs will eventually consider it.

If we ask `satabs` to show us the verification claims (`--show-claims`) for our example, we obtain

1. *Claim 1*: In File `driver.c` (Figure 4.5), line 7: assertion `(*inode).i_rdev >> 8 == dummy_major`
2. *Claim 2*: In File `spec.c` (Figure 4.4), line 14: assertion `FALSE`

It seems obvious that claim 1 can never be violated. SatAbs confirms this assumption: We call

```
satabs --claim 1 driver.c spec.c
```

and SatAbs reports `VERIFICATION SUCCESSFUL` after a few iterations. However, if we try to verify claim 2, SatAbs reports that the property in line 14 in file `spec.c` (Figure 4.4), function `unregister_chrdev` is violated (i.e., the assertion is `FALSE`, therefore the `VERIFICATION FAILED`). Furthermore, SatAbs provides a detailed description of the counterexample (i.e., the execution trace that violates the property). On this trace, `dummy_open` is called *twice*, leading to a `usecount` of 2 (the second call of course fails with `rval=-1`, but the counter is increased nevertheless). Then, `dummy_release` is called to release the lock on the device. Finally, the loop is left and the call to `unregister_chrdev` results in a violation of the assertion (since `usecount` is still 1, even though `locked=0`).

4.3.2 Example: Buffer Overflow in a Mail Transfer Agent

The example presented in Section 4.3.1 is obviously a toy example and can hardly be used to convince your project manager to use static verification in your next project. Even though we recommend to use formal verification and specification already in the early phases of your project, the sad truth is that in most projects verification (of any kind) is still pushed to the very end of the development cycle. Therefore, this section is dedicated to the verification of legacy code. However, the techniques presented here can also be used for unit testing.

We explain how to model check Aeon[3] version 0.2a, a small mail transfer agent written by Piotr Benetkiewicz. freshmeat.net claims that Aeon is a "good choice for *hardened* or minimalistic boxes".

Our first naive attempt to verify Aeon using

```
satabs --show-claims aeon.c base64.c lib_aeon.c
```

fails due to the following reasons:

1. *Parsing errors.* On some of the architectures that SatAbs supports, the frontend of SatAbs fails to parse the ANSI-C standard header files

[3] (available for download on the SatAbs homepage and on `http://freshmeat.net/projects/aeon/`)

that are distributed with your compiler of choice (e.g., GCC makes use of extensions of the C programming language that cannot be handled by SatAbs).

2. *Missing library functions.* As already stated on page 19, SatAbs is unable to find the source code for library functions like `strcmp`, `getenv` and `strtok`.

To solve the first of these problems, we provide a set of header files that are suitable for the SatAbs C parser. The header files can be downloaded from the SatAbs homepage. Assuming that you have unpacked these files to the directory `../include` (relative to the source files of Aeon) you can now instruct SatAbs to use these header files as follows:

```
satabs --show-claims -I ../include aeon.c base64.c lib_aeon.c
```

After complaining that the bodies of approximately 30 library functions are missing, SatAbs will provide more than 150 claims for Aeon.

Now, do you have to provide a body for all missing library functions? There is no easy answer to this question, but a viable answer would be "most likely not". It is necessary to understand how SatAbs handles functions without bodies: It simply assumes that such a function returns an arbitrary value, but that no other locations than the one on the left hand side of the assignment are changed. Obviously, there are cases in which this assumption is unsound, since the function potentially modifies all memory locations that it can somehow address. Consider the first few lines of the `main` function of Aeon in Figure 4.7 and the function `getConfig` in Figure 4.8. The function `getConfig` makes calls to `strcpy`, `strcat`, `getenv`, `fopen`, `fgets`, and `fclose`. It is very easy to provide an implementation for the functions from the string library (`string.h`), and we do so in Figure 4.9.

The implementation of `getenv` is not so straight forward. The man-page of `getenv` (which we obtain by entering `man 3 getenv` in a Unix or cygwin command prompt) tells us:

`'getenv'` searches the list of environment variable names and values (using the global pointer `"'char **environ'"`) for a variable whose name matches the string at `NAME`. If a variable name matches, `'getenv'` returns a pointer to the associated value.

SatAbs has no information whatsoever about the content of `environ`. Even if SatAbs could access the environment variables on your computer, a successful verification of Aeon would then only guarantee that the claims for this program hold on your computer with a specific set of environment variables. We have to assume that `environ` contains environment variables that have an arbitrary content of arbitrary length. The content of environment variables is not only arbitrary but could be malefic, since it can be modified by the user.

SatAbs provides several functions to model user input, one of which, namely `nondet_uint`, is used in the implementation of `getenv` in Figure 4.10. The prototype declaration of the function `nondet_uint` tells us that `nondet_uint` will return an unsigned integer. Since we do not provide a body to this function, SatAbs will consider all possible return values. In Figure 4.10 we use `nondet_uint` to determine the length of the the string that `getenv`

returns. The subsequent call to `assume` guarantees that `buf_size` is at least one (since we need to zero-terminate the string) and is bounded by `SATABS_MAX_BUF_LEN`. In our first approximation of the behaviour of `getenv` we completely ignore the content of the string.

We could model the function `fgets` in a similar manner, but before we do this, let us have another look at the claims that `Satabs` generates if we provide the implementations from the string library and for `getenv`. (The file `stubs.c` can be downloaded from the SatAbs homepage.)

```
satabs --show-claims -I ../include aeon.c base64.c \
lib_aeon.c ../stubs/stubs.c
```

Now SatAbs will generate approximately 700 claims. Most of these claims require that we verify that the upper and lower bounds of buffers or arrays are not violated. Let us look at the first few claims that SatAbs generates:

- Claim 4:
  file stubs.c line 8 column 10 function c::strcpy
  dereference failure: array 'home' lower bound
  `!(i < 0) || !(dest == &home[0])`

- Claim 5:
  file stubs.c line 8 column 10 function c::strcpy
  dereference failure: array 'home' upper bound
  `!(dest == &home[0]) || !(i >= 512)`

The variable `home` looks familiar; We encountered it line 7 of the function `getConfig` in Figure 4.8. The function `getenv` in combination with functions `strcpy`, `strcat` or `sprintf` is indeed often the source for buffer overflows. Therefore, we try to use SatAbs to check if the upper bound of the array `home`:

```
satabs --claim 5 -I ../include aeon.c base64.c \
lib_aeon.c ../stubs/stubs.c
```

SatAbs runs for quite a while and will eventually give up, telling us that it's upper bound for abstraction refinement iterations has been exceeded. This is not exactly the result we were hoping for, and we could now increase the bound for iterations with help of the `--iterations` command line switch of SatAbs.

Before we do this, let us investigate why SatAbs failed to provide a satisfying result. The function `strcpy` contains a loop that counts from 1 to the length of the input string. Predicate abstraction, the mechanism SatAbs is based on, is unable to detect such loops and will therefore unroll the loop body as often as necessary. The array `home` has `MAX_LEN` elements, and `MAX_LEN` is defined to be 512 in `aeon.h`. Therefore, SatAbs would have to run through at least 512 iterations, only to verify (or reject) one of the more than 700 claims! Does this fact defeat the purpose of static verification?

SatAbs provides a mechanism to bypass this problem by providing a detection for deep loops. For this purpose, SatAbs has to rely on a close cooperation with the abstract model checker. Currently, the only model

checker that provides loop detection is BOPPO (please refer to Chapter 2 to find out where to obtain and how to install BOPPO).

The switch `--loop-detection` tells SatAbs to activate the detection of deep loops. Furthermore, we tell SatAbs to use BOPPO instead of the default model checker SMV:

```
satabs --claim 5 --modelchecker boppo --loop-detection \
-I ../include aeon.c base64.c lib_aeon.c ../stubs/stubs.c
```

This time, SatAbs will tell us that it found a potential buffer overflow:

- Violated property:
  file stubs.c line 8 column 10 function c::strcpy
  dereference failure: array 'home' upper bound
  `!(dest == &home[0]) || !(i >= 512)`

Furthermore, SatAbs provides a counterexample trace that demonstrates how the buffer overflow be reproduced. If you use the Eclipse plugin (as described in Chapter 5), you can step through this counterexample, although it might be a bit tedious to step through 512 iterations of the loop in `strcpy`.

4.3.3 Unit Testing with SatAbs

As mentioned in Section 4.3.2, it is highly recommendable to use formal verificat as early as possible in your development cycle. Unit testing is used in most software development projects, and static verification with SatAbs can be very well combined with this task. Unit testing relies on a number test cases that yield the desired code coverage. Such test cases are imlemented by a software testing engineer in terms of a test harness (aka test driver) and a set of function stubs. Typically, a slight modification to the test harness allows it to be used with SatAbs. Replacing the explicit input values with non-deterministic inputs (as explained in sections 4.3.1 and 4.3.2 guarantees that SatAbs will try to achieve *full* path and state coverage (due to the fact that predicate abstraction implicitely detects equivalence classes). However, it is not guaranteed that SatAbs terminates in all cases. Keep in mind that you must not make any assumptions about the validity of the claims if SatAbs did not run to completion!

```
1  extern int dummy_major;
2  int locked;
3  int init_module (void) {
4        locked = FALSE;
5  }

6  int dummy_open (struct inode *inode, struct file *filp) {
7        assert (MAJOR (inode->i_rdev) == dummy_major);
8        MOD_INC_USE_COUNT;
9        if (locked)
10           return -1;
11       locked = TRUE;
12       return 0; /* success */
13 }

14 unsigned int dummy_read (struct file *filp,
                            char *buf, int max) {
15       int n;
16       if (locked) {
17           n = nondet_int ();
18           assume ((n >= 0) && (n <= max));
19           /* writing to the buffer is not modeled here */
20           return n;
21       }
22       return -1;
23 }

24 int dummy_release (struct inode *inode,
                           struct file *filp) {
25       if (locked) {
26           MOD_DEC_USE_COUNT;
27           locked = FALSE;
28           return 0;
29       }
30       return -1;
31 }
```

Figure 4.5: Sources (`driver.c`) for the "dummy" device driver

```
1  int main (int argc, char** argv) {
2        int rval, size;
3        struct file my_file;
4        char *buffer; /* we do not model this buffer */
5        struct inode inode;
6        unsigned char random;

7        dummy_major = register_chrdev (0, "dummy");
8        inode.i_rdev = dummy_major << MINORBITS;

9        init_module ();
10       /* assign arbitrary values */
11       my_file.f_mode = nondet_uint ();
12       my_file.f_pos = nondet_uint ();

13       do {
14            random = nondet_uchar ();
15            assume (0 <= random && random <= 3);

16            switch (random) {
17            case 1:
18                 rval = dummy_open (&inode, &my_file);
19                 break;
20            case 2:
21                 count = dummy_read (&my_file, buffer,
                                       BUF_SIZE);
22                 break;
23            default:
24                 dummy_release (&inode, &my_file);
25            }
26       } while (random || locked);

27       cleanup_module ();
28       unregister_chrdev (dummy_major, "dummy");

29       return 0;
30 }
```

Figure 4.6: Sources for the model checking harness

```
1  int main(int argc, char **argv)
2  {
3        char settings[MAX_SETTINGS][MAX_LEN];
4        ...
5        numSet = getConfig(settings);
6        if (numSet == -1) {
7             logEntry(M̈issing config file!);̈
8             exit(1);
9        }
10       ...
```

Figure 4.7: First few lines of `main` of Aeon (`aeon.c`)

```
1  /* reading rc file, handling missing options */
2  int getConfig(char settings[MAX_SETTINGS][MAX_LEN])
3  {
4        char home[MAX_LEN];
5        FILE *fp; /* .rc file handler */
6        int numSet = 0; /* number of settings */
7        strcpy(home, getenv("HOME")); /* get home path */
8        strcat(home, "/.aeonrc"); /* full path to rc file */
9        fp = fopen(home, "r");
10       if (fp == NULL) return -1; /* no cfg - ERROR */
11            while (fgets(settings[numSet], MAX_LEN-1, fp)
12                 && (numSet < MAX_SETTINGS)) numSet++;
13       fclose(fp);
14       return numSet;
15 }
```

Figure 4.8: Function `getConfig` of Aeon (`lib_aeon.c`)

```
1  char *strcpy (char *dest, const char *src)
2  {
3        int i;
4        for (i = 0 ;; i++) {
5             dest[i] = src[i];
6             if (src[i] == '\0')
7                  break;
8        }
9  }

10 char *strcat(char *dest, const char *src)
11 {
12       int i, j;
13       i = 0; j = 0;
14       while (dest[i] != '\0')
15            i++;
16       do {
17            dest[i] = src[j];
18            i++; j++;
19       } while (src[j] != '\0');
20       return dest;
21 }
```

Figure 4.9: Functions `strcpy` and `strcat` (`stubs.c`)

```
1  unsigned int nondet_uint (void);
2  #define SATABS_MAX_BUF_LEN (65535)

3  char *getenv(const char *name)
4  {
5        char* buffer;
6        size_t buf_size = nondet_uint ();
7        assume (buf_size > 1
8             && buf_size <= SATABS_MAX_BUF_LEN+1);
9        buffer = (char*)malloc (buf_size*sizeof(char));
10       buffer[buf_size-1]=0;
11       return buffer;
12 }
```

Figure 4.10: Function `getenv` (`stubs.c`)

5 The Eclipse User Interface

Many readers will agree that command line interfaces are a bit archaic nowadays. Therefore, we provide a user interface for CBMC and SATABS that integrates the command line tools seamlessly into Eclipse.[1]. We assume that you have already installed Eclipse as well as the SATABS plugin. The installation of the plugin is covered by Chapter 2,and installation instructions for Eclipse can be found on the Eclipse homepage. We assume some basic familiarity with Eclipse. This section provides a short, step by step introduction to the user interface and is based on the SATABS example in Section 4.3.1.The files used for the example are available for download at `http://www.cprover.org/satabs/examples/linux_toy_driver/`.

→ 2

→ 4.3.1

1. Start Eclipse.
2. Create a new SATABS project as shown in Figure 5.1. Name your project `DummyDriver`.

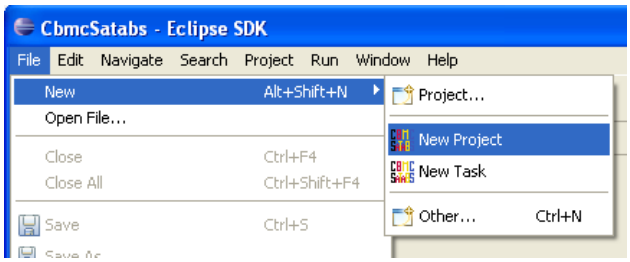

Figure 5.1: Creating a new SATABS project.

3. Once you have created a SATABS project, right click on the element `DummyDriver` in the Eclipse Navigator (or, alternatively, click on the File menu) and select *New* → *New Task* to create a new SATABS task. Name the task `usecount`.
4. Double click the element `usecount.tsk` in the Navigator to show the Eclipse view that provides the settings for this task. Figure 5.2 shows the *Files Selection* tab, which allows you to select the source files you want to run SATABS on. Click on the *Change* button to select the directory where the source files you want to verify are located. Choose the driver example described in the previous section.
5. Select the source files `spec.c` and `driver.c` (see Figure 5.2). Do not select the header files, as the C preprocessor already includes the files.

[1] Eclipse is an open-sourced integrated development environment and can be downloaded from `http://www.eclipse.org`.

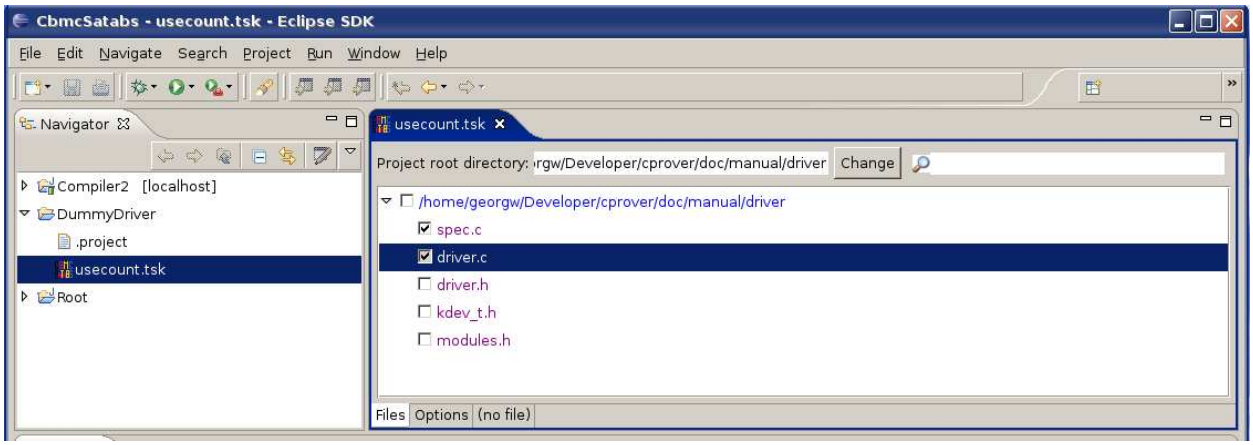

Figure 5.2: Selecting source files using the Eclipse plugin for SATABS

6. The *Includes and Defines* tab can be used to specify include paths and add definitions. This is useful if you want to replace the standard header files by a specific, SATABS enhanced version. (Due to the GNU extensions to C, SATABS fails to parse some of the GCC header files.) In a later section, we provide a more complex example where this is necessary.

7. The *Options* tab (see Figure 5.3) allows you to specify the command line options for SATABS. This tab reflects exactly the parameters presented in Section 4.2. Make sure that checking of assertions is activated.

8. Create a launch configuration by clicking the green run button (). Figure 5.4 shows the dialog that allows you to create a new run configuration for SATABS. Name the new configuration `Satabs` and make sure that the `satabs` executable is selected (not `cbmc`).

9. Now run SATABS on `usecount.tsk` by clicking  again. SATABS generates two claims for the driver example, one for each assertion.

10. Verify the first assertion[2] by clicking on `driver.c` with the right mouse button and selecting *Check Selection*. As in the previous section, SATABS takes a few seconds the verify that the assertion cannot be violated. This is indicated by the symbol ✔ that is displayed in the left-most column of the claim. Furthermore, assertions that have already been verified are highlighted in green in the source file.

11. Repeat the same step for the second assertion. This time, SATABS takes a bit longer to come to a conclusion. The log view in Figure 5.6 shows that SATABS needs several refinement abstraction iterations.

[2] In the source file view, the assertion is highlighted yellow, indicating that it has not been checked so far.

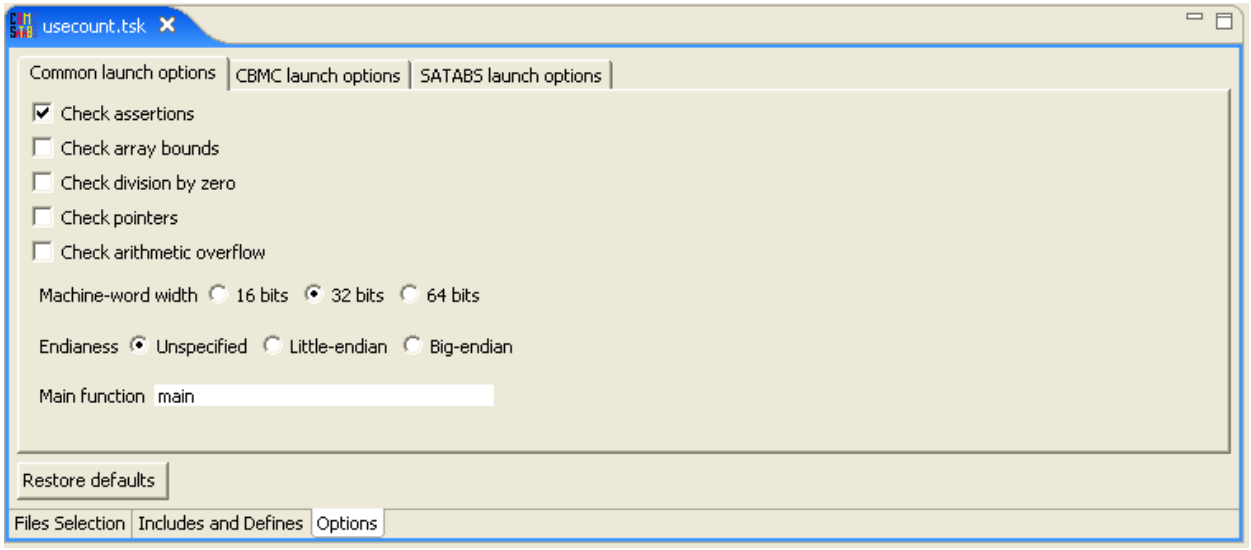

Figure 5.3: Specifying the command line options for SATABS

After some time, SatAbs should report that the claim can be violated (see Figure 5.7).

12. SatAbs provides a detailed execution trace that explains how the assertion can be violated. The Trace view shown in Figure 5.7 can be used to step through the counterexample and provides the values of the variables in each step. Since the program is not actually executed,[3] you can step forwards as well as backwards, a feature which makes it much easier to understand the counterexample and find the error in the program.

[3] Therefore, you cannot change the values of the variables, as you can do in a debugger.

Figure 5.4: Running SATABS in Eclipse

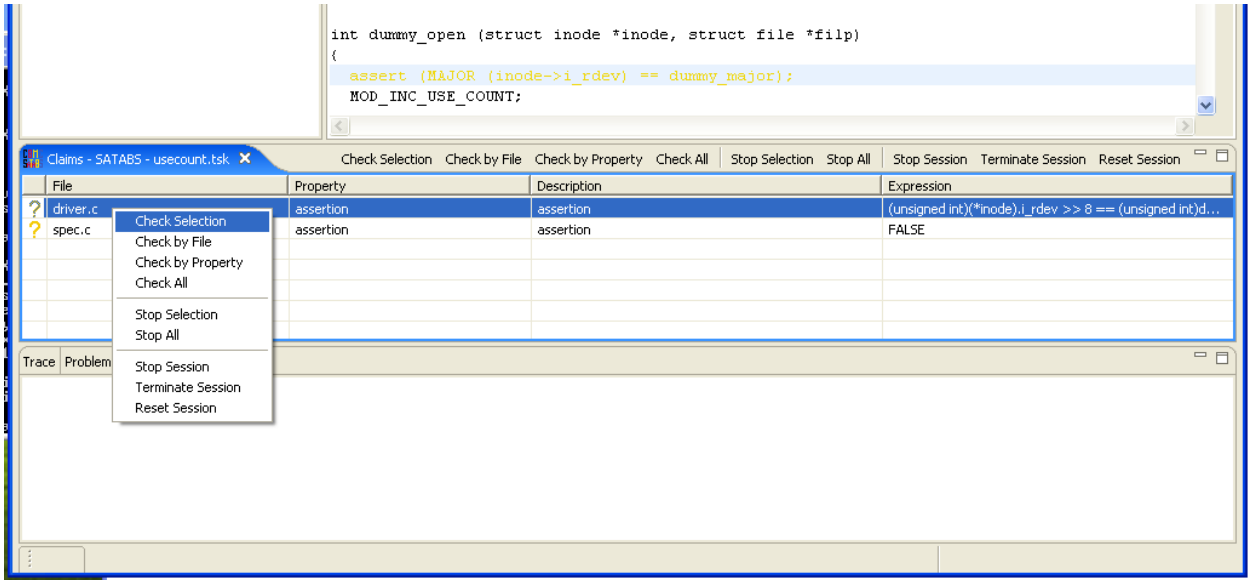

Figure 5.5: Using SATABS to verify an assertion

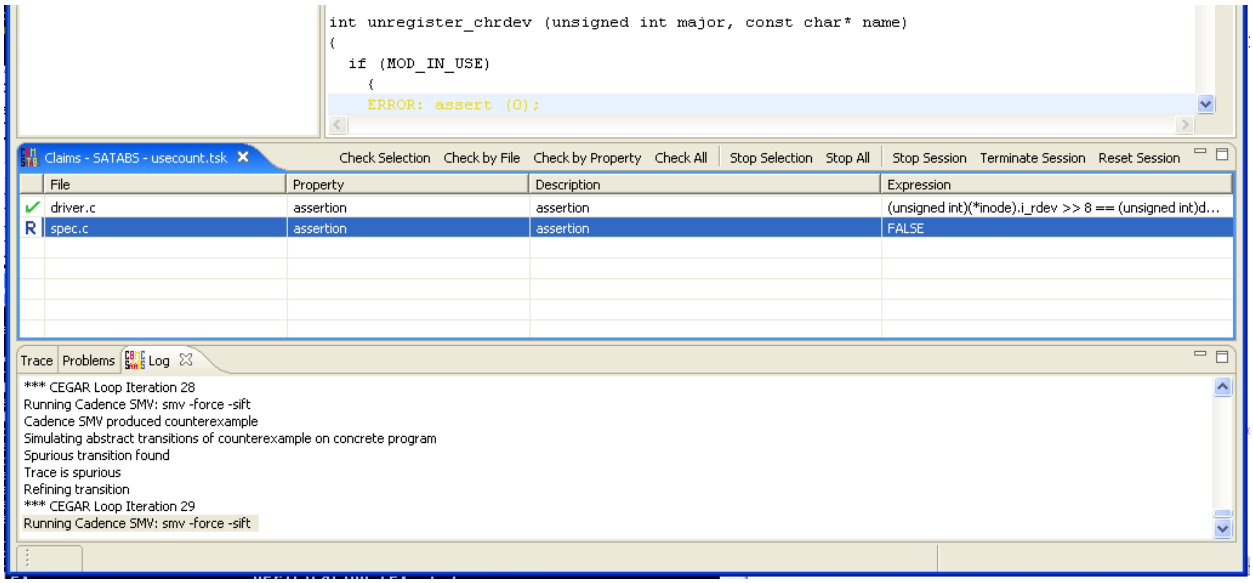

Figure 5.6: SATABS is checking an assertion (iterations shown in log)

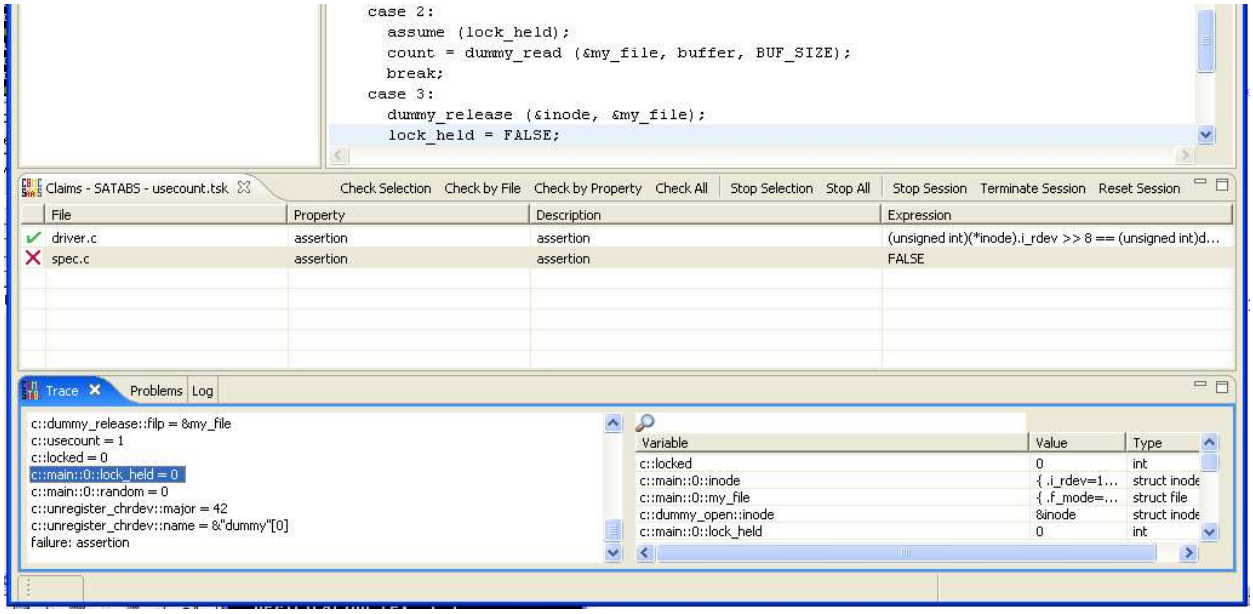

Figure 5.7: Stepping through a counterexample in Eclipse

6 Build Systems and Libraries

6.1 Integration into Build Systems with goto-cc

Existing software projects usually do not come in a single source file that may simply be passed to a model checker. They rather come in a multitude of source files in different directories and refer to external libraries and system-wide options. A build system then collects the configuration options from the system and compiles the software according to build rules.

The most prevalent build tool on Unix (-based) systems surely is the *make* utility. This tool uses build rules given in a *Makefile* that comes with the software sources. Running software verification tools on projects like these is greatly simplified by a compiler that first collects all the necessary models into a single model file. GOTO-CC is such a model file extractor, which can seamlessly replace `gcc` in Makefiles. The normal build system for the project may be used to build the software, but the result will be a model file with suitable detail for verification, as opposed to a flat executable program.

6.1.1 Example: Building wu-ftpd

1. Download the sources of wu-ftpd from
   `ftp://ftp.wu-ftpd.org/pub/wu-ftpd/wu-ftpd-current.tar.gz`
2. Unpack the sources by running
   `tar xfz wu-ftpd-current.tar.gz`
3. Change to the source directory, by entering, e.g.,
   `cd wu-ftpd-2.6.2`
4. Configure the project for verification by running
   `./configure YACC=byacc CC=goto-cc --host=none-none-none`
5. Build the project by running
   `make`
   This creates multiple model files in the `src` directory. Among them is a model for the main executable `ftpd`.
6. Run a model-checker, e.g., CBMC, on the model file:
   `cbmc --binary src/ftpd`

6.1.2 Important Notes

More elaborate build or configuration scripts often make use of features of the compiler or the system library to detect configuration options automatically, e.g., in a `configure` script.

Replacing `gcc` by GOTO-CC at this stage may confuse the script, or detect wrong options. For example, missing library functions do not cause GOTO-CC to throw an error (only to issue a warning). Because of this, configuration

scripts sometimes falsely assume the availability of a system function or library.

In the case of this or similar problems, it is more advisable to configure the project using the normal routine, and replacing the compiler setting manually in the generated Makefiles, e.g., by replacing lines like `CC=gcc` by `CC=goto-cc`.

A helpful command that accomplishes this task successfully for many projects is the following:

```
for i in ‘find . -name Makefile‘; do
  sed -e ’s/^\(\s*CC[ \t]*=\)\(.*$\)/\1goto-cc/g’ -i $i
done
```

6.2 Libraries

6.2.1 Linking libraries

Some software projects come with their own libraries; also, the goal may be to analyze a library by itself. For this purpose it is possible to use GOTO-CC to link multiple model files into a library of model files. An object file can then be linked against this model library. For this purpose, GOTO-CC also supports a pure linker mode.

To apply this linker mode, create a link to the GOTO-CC binary by the name of `goto-ld` (alternatively copy the GOTO-CC binary, if your system does not support links). The `goto-ld` tool can now be used as a seamless replacement for the `ld` tool present on most Unix (-based) systems.

The default linker may need to be replaced by `goto-ld` in the build script, which can be achieved in much the same way as replacing the compiler (see Section 6.1.2).

6.2.2 Abstractions for Zero-Terminated Strings

Overview

CBMC, SATABS, and GOTO-CC come with a built-in library for some of the string functions defined in `string.h`. In case the input files do not provide function definitions for any of those functions, the built-in libraries are automatically added to improve the precision of the analysis. Also, the abstract function bodies contain assertions enforcing preconditions of the string functions.

The default library for these functions is not an exact model; it rather only provides an over-approximation, which means that it may allow behavior not present in a real implementation. The goal of the over-approximation is to track the *length* of the strings, as opposed to their content. Any detail that relates to the content of zero-terminated strings is therefore lost.

The over-approximation is realized as follows: the input model is augmented by additional variables that track the termination status, the buffer size, and the length of a string. These variables may be accessed by using the following three macros:

- `_Bool __CPROVER_is_zero_string(const void *str)`
  (Returns true if `str` is zero-terminated.)
- `unsigned __CPROVER_zero_string_length(const void *str);`
  (Returns the position of a zero-character in `str`. Note that this is only an upper bound for the string length.)
- `unsigned __CPROVER_buffer_size(const void *buffer);`
  (Returns the capacity of `buffer`.)

These macros are automatically inserted into the model wherever necessary. This means that all three properties of strings are soundly tracked throughout the program, in an abstract fashion. While this not only increases the precision of the analysis (compared to the absence of a model for the string functions), it also improves the verification performance over full concrete implementations of the string functions.

Example: `strlen`

Figure 6.1 shows the principle of the abstract string functions in the built-in libraries.

```
1 inline unsigned strlen(const char *s)
2 {
3       __CPROVER_HIDE:
4       __CPROVER_assert(__CPROVER_is_zero_string(s),
5                        "strlen zero-termination");
6       return __CPROVER_zero_string_length(s);
7 }
```

Figure 6.1: The abstract definition of `strlen`.

Programs may call the standard library function `strlen` without linking to a model file of the system library. The abstract definition of the function is automatically added to the input model. Note that in this example, the return value of the `strlen` function is not computed by looping over the string. The function simply returns a value read modeled by additional program variables, which may be non-deterministic.

Disabling built-in libraries and string abstraction

The addition of the built-in libraries, or the abstraction of strings, can can be disabled using the following two command-line options:

- `--no-library`
  (Prevents built-in libraries from being added.)
- `--no-string-abstraction`
  (Disables abstract tracking of string properties.)

7 ANSI-C/C++ Language Features

7.1 Basic Datatypes

CBMC and SatAbs support the scalar data types as defined by the ANSI-C standard, including `_Bool`. By default, `int` is 32 bits wide, `short int` is 16 bits wide, and `char` is 8 bits wide. Using a command line option, these default widths can be changed. By default, `char` is signed. Since some architectures use an unsigned `char` type, a command line option allows to change this setting.

There is also support for the floating point data types `float`, `double`, and `long double`. By default, CBMC and SatAbs use fixed-point arithmetic for these types. Variables of type `float` have by default 32 bits (16 bits integer part, 16 bits fractional part), variables of type `double` and `long double` have 64 bits.

In addition to the types defined by the ANSI-C standard, CBMC and SatAbs support the following types, which are Microsoft C extensions: `__int8`, `__int16`, `__int32`, and `__int64`. These types define a bit vector with the given number of bits.

7.2 Operators

7.2.1 Boolean Operators

CBMC and SatAbs support all ANSI-C Boolean operators on scalar variables `a`, `b`:

Operator	Description
`!a`	negation
`a && b`	and
`a || b`	or

7.2.2 Integer Arithmetic Operators

CBMC and SatAbs support all integer arithmetic operators on scalar variables `a`, `b`:

Operator	Description
`-a`	unary minus, negation
`a+b`	sum
`a-b`	subtraction
`a*b`	multiplication
`a/b`	division
`a%b`	remainder
`a<<b`	bit-wise left shift
`a>>b`	bit-wise right shift
`a&b`	bit-wise and
`a|b`	bit-wise or
`a^b`	bit-wise xor
`a< b`	relation
`a<=b`	relation
`a> b`	relation
`a>=b`	relation

Note that the multiplication, division, and reminder operators are very expensive with respect to the size of the equation that is passed to the SAT solver. Furthermore, the equations are hard to solve for all SAT solvers known to us.

As an example, consider the following program:

```
int main() {
  unsigned char a, b;
  unsigned int result=0, i;

  a=nondet_uchar();
  b=nondet_uchar();

  for(i=0; i<8; i++)
    if((b>>i)&1)
      result+=(a<<i);

  assert(result==a*b);
}
```

The program nondeterministically selects two 8-bit unsigned values, and then uses shift-and-add to multiply them. It then asserts that the result (i.e., the sum) matches `a*b`. Although the resulting SAT instance has only about 1400 variables, it takes 12 minutes to solve using Chaff.

Properties Checked Optionally, CBMC and SatAbs allow checking for arithmetic overflow in case of signed operands. In case of the division and the remainder operator, CBMC and SatAbs check for division by zero. This check can be disabled using a command line option.

As an example, the following program nondeterministically selects two unsigned integers `a` and `b`. It then checks that either of them is non-zero and then computes the inverse of `a+b`:

```
int main() {
  unsigned int a, b, c;

  a=nondet_uint();
  b=nondet_uint();

  if(a>0 || b>0)
    c=1/(a+b);
}
```

However, due to arithmetic overflow when computing the sum, the division can turn out to be a division by zero. CBMC generates a counterexample as follows for the program above:

```
Initial State
-----------------------------------------------------
  c=0 (00000000000000000000000000000000)

State 1 file div_by_zero.c line 4 function main
-----------------------------------------------------
  a=4294967295 (11111111111111111111111111111111)

State 2 file div_by_zero.c line 5 function main
-----------------------------------------------------
  b=1 (00000000000000000000000000000001)

Failed assertion: division by zero file div_by_zero.c
line 8 function main
```

7.2.3 Floating Point Arithmetic Operators

CBMC and SatAbs support the following operators on variables of the types `float`, `double`, and `long double`:

Operator	Description
`-a`	unary minus, negation
`a+b`	sum
`a-b`	subtraction
`a*b`	multiplication
`a/b`	division
`a< b`	relation
`a<=b`	relation
`a> b`	relation
`a>=b`	relation

Note that the multiplication and division operators are very expensive with respect to the size of the equation that is passed to the SAT solver. Furthermore, the equations are hard to solve for all SAT solvers known to us. CBMC and SatAbs also support type conversions to and from integer types. Different rounding modes are currently not supported.

7.2.4 The Comma Operator

CBMC and SatAbs support the comma operator `a, b`. The operands are evaluated for potential side effects. The result of the operator is the right operand.

7.2.5 Type Casts

CBMC and SatAbs have full support for arithmetic type casts. As an example, the expression `(unsigned char)i` for an integer `i` is guaranteed to be between 0 and 255 in case of an eight bit character type.

Properties Checked For the unsigned data types, the ANSI-C standard requires modulo semantics, i.e., that no overflow exception occurs. Thus, overflow is not checked. For signed data types, an overflow exception is permitted. Optionally, CBMC and SatAbs check for such arithmetic overflows.

7.2.6 Side Effects

CBMC and SatAbs allow all side effect operators with their respective semantics. This includes the assignment operators (`=`, `+=`, etc.), and the pre- and post- increment and decrement operators.

As an example, consider the following program fragment:

```
unsigned int i, j;

i=j++;
```

After the execution of the program, the variable `i` will contain the initial value of `j`, and `j` will contain the initial value of `j` plus one. CBMC generates the following equation from the program:

$$\begin{aligned} i_1 &= j_0 \\ j_1 &= j_0 + 1 \end{aligned}$$

CBMC and SatAbs perform the implicit type cast as required by the ANSI-C standard. As an example, consider the following program fragment:

```
char c;
int i;
long l;

l = c = i;
```

The value of `i` is converted to the type of the assignment expression `c=i`, that is, `char` type. The value of this expression is then converted to the type of the outer assignment expression, that is, `long int` type.

Ordering of Evaluation The ANSI-C standard allows arbitrary orderings for the evaluations of expressions and the time the side-effect becomes visible. The only exceptions are the operators `&&`, `||`, and the ternary operator `?:`. For the Boolean operators, the standard requires strict evaluation

from left to right, and that the evaluation aborts once the result is known. The operands of the expression `c ? x : y` must be evaluated as follows: first, `c` is evaluated. If `c` is `true`, `x` is evaluated, and `y` otherwise.

As an example, assume that a pointer `p` in the following fragment may point to either `NULL` or a valid, active object. Then, an `if` statement as follows is valid, since the evaluation must be done from left to right, and if `p` points to NULL, the result of the Boolean AND is known to be `false` and the evaluation aborts before `p` is dereferenced.

```
if(p!=NULL && *p==5) {
  . . .
```

For other operators, such as addition, no such fixed ordering exists. As an example, consider the following fragment:

```
int g;

int f() {
  g=1;
  ...
}

...
  g=2;

  if(f()+g==1) {
    ...
```

In this fragment, a global variable `g` is assigned to by a function `f()`, and just before an `if` statement. Furthermore, `g` is used in an addition expression in the condition of the `if` statement together with a call to `f()`. If `f()` is evaluated first, the value of `g` in the sum will be one, while it is two if `g` is evaluated first. The actual result is architecture dependent.

Properties Checked CBMC and SatAbs model this problem as follows: One option allows setting a fixed ordering of evaluation for all operators. The other option allows checking for such artifacts: CBMC and SatAbs can assert that no side-effect affects the value of any variable that is evaluated with equal priority. This includes changes made indirectly by means of pointers. In the example, this is realized by write-protecting the variable `g` during the execution of `f`. This rules out programs that show architecture dependent behavior due to the ordering of evaluation. While such programs are still valid ANSI-C programs, we do not believe that programs showing architecture dependent behavior are desirable.

7.2.7 Function Calls

CBMC and SatAbs support functions by inlining. No modular approach is done. CBMC and SatAbs preserve the locality of the parameters and the non-static local variables by renaming.

As an example, the following program calls the functions `f()` and `g()` twice. While `f()` uses a static variable, which is not renamed between calls, `g()` uses a true local variable, which gets a new value for each call.

```
int f() {
  static int s=0;

  s++;

  return s;
}

int g() {
  int l=0;

  l++;

  return l;
}

int main() {
  assert(f()==1); // first call to f
  assert(f()==2); // second call to f
  assert(g()==1); // first call to g
  assert(g()==1); // second call to g
}
```

CBMC supports Recursion by finite unwinding, as done for `while` loops. CBMC checks that enough unwinding is done by means of an *unwinding assertion* (section 7.3.6 provides more details). SatAbs does not support recursion.

7.3 Control Flow Statements

7.3.1 Conditional Statement

CBMC and SatAbs allow the use of the conditional statement as described in the ANSI-C standard.

Properties Checked CBMC and SatAbs generate a warning if the assignment operator is used as condition of a control flow statement such as `if` or `while`.

7.3.2 `return`

The `return` statement without value is transformed into an equivalent `goto` statement. The target is the end of the function. The `return` statement with value is transformed into an assignment of the value returned and the `goto` statement to the end of the function.

Properties Checked CBMC and SatAbs enforce that functions with a non-void return type return a value by means of the return statement. The execution of the function must not end by reaching the end of the function. This is realized by inserting `assert(FALSE);` at the end of the function. CBMC and SatAbs report an error trace if this location is reachable.

As an example, consider the following fragment:

```
int f() {
  int c=nondet_int();

  if(c!=1)
    return c;
}

int main() {
  int i;
  i=f();
}
```

In this fragment, `f()` may exit without returning a value. CBMC produces the following counterexample:

```
State 1 file no-return.c line 2 function f
------------------------------------------------------
  c=1 (00000000000000000000000000000001)

Failed assertion: end-of-function assertion
file no-return.c line 6 function f
```

7.3.3 goto

While only few C programs make use of `goto` statements, CBMC and SatAbs provide full support for such programs. CBMC distinguishes forward and backward jumps. In case of backward jumps, the same technique used for loops is applied: the loop is unwound a given number of times, and then we check that this number of times is sufficient by replacing the `goto` statement by `assert(FALSE);`.

7.3.4 break and continue

The `break` and `continue` statements are replaced by equivalent `goto` statements as described in the ANSI-C standard.

7.3.5 switch

CBMC and SatAbs provide full support for `switch` statements, including fall-through.

7.3.6 Loops

In Bounded Model Checking, the transition system is unwound up to a finite depth. In case of C programs, this means that `for` and `while` are unwound

up to a certain depth. In many cases, CBMC is able to automatically detect the maximum number of times a loop can be executed. This includes `while` loops and loops with modifications to the loop counter inside the loop body, even when done indirectly using a pointer.

However, in case of loops that have no pre-set bound, e.g., loops iterating on dynamic data structures, the user must specify a bound by means of the `--unwind` command line argument. CBMC will then unwind the loops up to that bound and check that the number is large enough by means of an *unwinding assertion*.

SatAbs uses abstraction, and thus, requires no depth-bound. The verification result is valid for any number of iterations.

7.4 Non-Determinism

CBMC and SatAbs allow to model user-input by means of non-deterministic choice functions. The names of these functions have the prefix `nondet_`. The value range generated is determined by the return type of the function. As an example,

```
int nondet_int();
```

returns a nondeterministically chosen value of type `int`. The functions are built-in, i.e., the prototype is sufficient. CBMC and SatAbs will evaluate all traces arising from the possible choices.

7.5 Assumptions and Assertions

CBMC and SatAbs check assertions as defined by the ANSI-C standard: The `assert` statement takes a Boolean condition, and the tools check that this condition is true for all runs of the program. The logic for assertions is the usual ANSI-C expression logic. In addition to the `assert` statement, the `__CPROVER_assert` statement can be used to annotate the assertions with a comment:

```
__CPROVER_assert(!(x&1), "x divisible by 2");
```

CBMC and SatAbs also provide the `__CPROVER_assume` statement. The `__CPROVER_assume` statement restricts the program traces that are considered and allows assume-guarantee reasoning. Similar to an assertion, an assumption takes a Boolean expression as argument. Intuitively, the `__CPROVER_assume` statement aborts the program *successfully* if the condition evaluates to false. If the condition evaluates to true, the execution continues.

As an example, the following function first nondeterministically picks an integer value. It then assumes that the integer is in a specific range and returns the value.

```
int one_to_ten() {
  int value=nondet_int();
  __CPROVER_assume(value>=1 && value<=10);
  return value;
}
```

Note that the assume statement is not retro-active with respect to assertions. E.g.,

```
assert(value<10);
__CPROVER_assume(value==0);
```

may fail, while

```
__CPROVER_assume(value==0);
assert(value<10);
```

passes.

When using the `__CPROVER_assume` statement, it must be ensured that there still exists a program trace that satisfies the condition. Otherwise, any property will pass vacuously. This should be checked by replacing the property by `false`. If no counterexample is produced, the assumptions eliminate all program paths.

7.6 Arrays

CBMC and SatAbs allow arrays as defined by the ANSI-C standard. This includes multi-dimensional arrays and dynamically-sized arrays.

Dynamic Arrays The ANSI-C standard allows arrays with non-constant size as long as the array does not have static storage duration, i.e., is a non-static local variable. Even though such a construct has a potentially huge state space, CBMC and SatAbs provide full support for arrays with non-constant size. The size of the Boolean equation that is generated does not depend on the array size, but rather on the number of read or write accesses to the array.

Properties Checked CBMC and SatAbs check both lower and upper bound of arrays, even for arrays with dynamic size. As an example, consider the following fragment:

```
{
  unsigned size=nondet_uint();
  char a[size];

  a[10]=0;
}
```

In this fragment, an array `a` is defined, which has a nondeterministically chosen size. The code then accesses the array element with index 10. CBMC produces a counterexample with an upper array bound error on array `a`. The trace shows a value for `size` less then 10.

Furthermore, CBMC and SatAbs check that the size of arrays with dynamic size is non-negative. As an example, consider the following fragment:

```
signed size=nondet_int();
char a[size];
```

For this fragment, CBMC produces a counterexample in which the size of the array `a` is negative.

7.7 Structures

CBMC and SatAbs allow arbitrary structure types. The structures may be nested, and may contain arrays.

The `sizeof` operator applied to a structure type yields the sum of the sizes of the components. However, the ANSI-C standard allows arbitrary padding between components. In order to reflect this padding, the `sizeof` operator should return the sum of the sizes of the components *plus* a nondeterministically chosen non-negative value.

Recursive Structures Structures may be recursive by means of pointers to the same structure. As an example, consider the following fragment:

```
void *malloc(unsigned);

struct nodet {
  struct nodet *n;
  int payload;
};

int main() {
  unsigned i;
  struct nodet *list=(void *)0;
  struct nodet *new_node;

  for(i=0; i<10; i++) {
    new_node=malloc(sizeof(*new_node));
    new_node->n=list;
    list=new_node;
  }
}
```

The fragment builds a linked list with ten dynamically allocated elements.

Structures with Dynamic Array The last component of an ANSI-C structure may be an incomplete array (an array without size). This incomplete array is used for dynamic allocation. This is described in section 7.10.

7.8 Unions

CBMC and SatAbs allow the use of unions to use the same storage for multiple data types. Internally, CBMC actually shares the literals used to represent the variables values among the union members.

Properties Checked CBMC and SatAbs do not permit the use of unions for type conversion, as this would result in architecture dependent behavior. Specifically, if a member is read, the same member must have been used for writing to the union the last time.

7.9 Pointers

7.9.1 The Pointer Data Type

Pointers are commonly used in ANSI-C programs. In particular, pointers are required for call by reference and for dynamic data structures. CBMC and SatAbs provide extensive support for programs that use pointers according to rules set by the ANSI-C standard, including pointer type casts and pointer arithmetic.

The size of a pointer, e.g., `sizeof(void *)` is by default 4 bytes. This can be adjusted using a command line option.

Conversion of pointers from and to integers The ANSI-C standard does not provide any guarantees for the conversion of pointers into integers. However, CBMC and SatAbs ensure that the conversion of the same address into an integer yields the same integer. The ANSI-C standard does not guarantee that the conversion of a pointer into an integer and then back yields a valid pointer. CBMC and SatAbs do not allow this construct.

7.9.2 Pointer Arithmetic

CBMC and SatAbs support the ANSI-C pointer arithmetic operators. As an example, consider the following fragment:

```
int array[10], *p;

int main() {
        array[1] = 1;
        p = &array[0];
        p++;

        assert(*p == 1);
}
```

7.9.3 The Relational Operators on Pointers

The ANSI-C standard allows comparing to pointers using the relational operators `<=`, `<`, `>=`, `<`.

Properties Checked The standard restricts the use of these operators to pointers that point to the same object. CBMC and SatAbs enforce this restriction by means of an automatically generated assertion.

7.9.4 Pointer Type Casts

CBMC and SatAbs provide full support for pointer type casts as described by the ANSI-C standard. As an example, it is a common practice to convert a pointer to, e.g., an integer into a pointer to `void` and then back:

```
int i;
void *p;

p=&i;
...
*((int *)p)=5;
```

Note that pointer type casts are frequently used for architecture specific type conversions, e.g., to write an integer byte-wise into a file or to send it over a socket:

```
int i;
char *p;

p=(char *)&i;

for(j=0; j<4; j++) {
  /* write *p */
  p++;
}
```

The result is architecture-dependent. In particular, it exposes the endianess of the architecture. CBMC and SatAbs support these constructs when enabled by a command line option. The command line option specifies the memory model (little endian or big endian).

Properties Checked CBMC and SatAbs check that the type of the object being accessed matches the type of the dereferencing expression. For example, the following fragment uses a `void *` pointer to store the addresses of both `char` and `int` type objects:

```
int nondet_int();
void *p;
int i;
char c;

int main() {
  int input1, input2, z;

  input1=nondet_int();
  input2=nondet_int();

  p=input1? (void *)&i : (void *)&c;
```

```
  if(input2)
    z=*(int *)p;
  else
    z=*(char *)p;
}
```

CBMC produces the following counterexample:

```
Initial State
-----------------------------------------------------
  c=0 (00000000)
  i=0 (00000000000000000000000000000000)
  p=NULL

State 1 file line 10 function main
-----------------------------------------------------
  input1=0 (00000000000000000000000000000000)

State 2 file line 11 function main
-----------------------------------------------------
  input2=1 (00000000000000000000000000000001)

State 3 line 13 function main
-----------------------------------------------------
  p=&c

Failed assertion: dereference failure (wrong object type)
line 16 function main
```

Note that the ANSI-C standard allows the conversion of pointers to structures to another pointer to a prefix of the same structure. As an example, the following program performs a valid pointer conversion:

```
typedef struct {
  int i;
  char j;
} s;

typedef struct {
  int i;
} prefix;

int main() {
  s x;
  prefix *p;

  p=(prefix *)&x;

  p->i=1;
}
```

7.9.5 String Constants

ANSI-C implements strings of characters as an array. Strings are then often represented by means of a pointer pointing to the array. Array bounds violations of string arrays are the leading cause of security holes in Internet software such as servers or web browsers.

CBMC and SatAbs provide full support for string constants, usable either in initializers or as a constant. As an example, the following fragment contains a string array `s`, which is initialized using a string constant. Then, a pointer `p` is initialized with the address of `s`, and the second character of `s` is modified indirectly by dereferencing `p`. The program then asserts this change to `s`.

```
char s[]="abc";

int main() {
  char *p=s;

  /* write to p[1] */
  p[1]='y';

  assert(s[1]=='y');
}
```

Properties Checked CBMC and SatAbs perform bounds checking for string constants as well as for normal arrays. In the following fragment, a pointer `p` is pointing to a string constant of length three. Then, an input `i` is used as address of an array index operation. CBMC and SatAbs assert that the input `i` is not greater than four (the string constant ends with an implicit zero character).

```
char *p="abc";

void f(unsigned int i) {
  char ch;

  /* results in bounds violation with i>4 */
  ch=p[i];
}
```

In addition to that, CBMC and SatAbs check that string constants are never written into by means of pointers pointing to them.

7.9.6 Pointers to Functions

CBMC and SatAbs allow pointers to functions, and calls through such a pointer. The function pointed to may depend on nondeterministically chosen inputs. As an example, the following fragment contains a table of three function pointers. The program uses a function argument to index

the table and then calls the function. It then asserts that the right function was called.

```
int global;

int f() { global=0; }
int g() { global=1; }
int h() { global=2; }

typedef int (*fptr)();
fptr table[] = { f, g, h };

void select(unsigned x) {
  if(x<=2) {
    table[x]();
    assert(global==x);
  }
}
```

7.10 Dynamic Memory

CBMC and SatAbs allow programs that make use of dynamic memory allocation, e.g., for dynamically sized arrays or data structures such as lists or graphs. As an example, the following fragment allocates a variable number of integers using `malloc`, writes one value into the last array element, and then deallocates the array:

```
void *malloc(unsigned);

void f(unsigned int n) {
  int *p;

  p=malloc(sizeof(int)*n);

  p[n-1]=0;

  free(p);
}
```

Properties Checked CBMC and SatAbs check array bounds of dynamically allocated arrays, and it checks that a pointer pointing to a dynamic object is pointing to an active object (i.e., that the object has not yet been freed and that it is not a static object). Furthermore, CBMC and SatAbs check that an object is not freed more than once.

In addition to that, CBMC can check that all dynamically allocated memory is deallocated before exiting the program, i.e., CBMC can prove the absence of "memory leaks".

As an example, the following fragment dynamically allocates memory, and stores the address of that memory in a pointer `p`. Depending on an input `i`, this pointer is redirected to a local variable `y`. The memory pointed to by `p` is then deallocated using `free`. CBMC detects that there is an illegal execution trace in case that the input `i` is true.

```
void *malloc(unsigned);

void f(_Bool i) {
  int *p;
  int y;

  p=malloc(sizeof(int)*10);

  if(i) p=&y;

  /* error if p points to y */
  free(p);
}
```

Notice that the standard semantics of `malloc` allow a return value of `NULL` if the allocation fails for any reason. This is *not* part of the model that ships with CBMC and SatAbs, as too many programs rely on such faults not occurring. Thus, bugs relating to out-of-memory scenarios may be missed by CBMC or SatAbs. A simple (but effective) way to mend this is to replace `malloc` by a custom function, say `my_malloc`, that returns `NULL` non-deterministically:

```
void *malloc(unsigned);
_Bool nondet_bool();

void *my_malloc(unsigned s) {
  if(nondet_bool()) return 0;
  return malloc(s);
}
```

7.11 Concurrency

SatAbs is able to verify concurrent (multi-threaded) programs with shared-variable communication between the threads. Threads are created with the functions provided by the PThread library. As an example, consider the following program:

```
#include <pthread.h>

int g;

void *thread(void *arg) {
  g=2;
```

```
}

int main() {
  pthread_t id1;

  pthread_create(&id1, NULL, thread, NULL);

  // this may fail
  g=1;
  assert(g==1);
}
```

SatAbs also supports mutual exclusion via the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. It provides an option that automatically generates assertions for data-races.

CBMC currently does not support concurrency.

8 Hardware and Software Equivalence and Co-Verification

8.1 Introduction

A common hardware design approach employed by many companies is to first write a quick prototype that behaves like the planned circuit in a language like ANSI-C. This program is then used for extensive testing and debugging, in particular of any embedded software that will later on be shipped with the circuit. An example is the hardware of a cell phone and its software. After testing and debugging of the program, the actual hardware design is written using hardware description languages like VHDL or Verilog.

Thus, there are two implementations of the same design: one written in ANSI-C, which is written for simulation, and one written in register transfer level HDL, which is the actual product. The ANSI-C implementation is usually thoroughly tested and debugged.

Due to market constraints, companies aim to sell the chip as soon as possible, i.e., shortly after the HDL implementation is designed. There is usually little time for additional debugging and testing of the HDL implementation. Thus, an automated, or nearly automated way of establishing the consistency of the HDL implementation is highly desirable.

This motivates the verification problem: we want to verify the consistency of the HDL implementation, i.e., the product, using the ANSI-C implementation as a reference [4]. Establishing the consistency does not require a formal specification. However, formal methods to verify either the hardware or software design are still desirable.

Related Work There have been several attempts in the past to tackle the problem. In [7], a tool for verifying the combinational equivalence of RTL-C and an HDL is described. They translate the C code into HDL and use standard equivalence checkers to establish the equivalence. The C code has to be very close to a hardware description (RTL level), which implies that the source and target have to be implemented in a very similar way. There are also variants of C specifically for this purpose. The SystemC standard defines a subset of C++ that can be used for synthesis [6]. Further variants of ANSI-C for specifying hardware are SpecC and Handel C, among others.

The concept of verifying the equivalence of a software implementation and a synchronous transition system was introduced by Pnueli, Siegel, and Shtrichman [5]. The C program is required to be in a very specific form, since a mechanical translation is assumed.

In [2], Currie, Hu, and Rajan transform DSP assembly language into an equation for the Stanford Validity Checker. However, problems involving bit vector overflow are not detected and while loops are not supported.

The symbolic execution of programs for comparison with RTL is common practice [3, 1].

The previous work focuses on a small subset of ANSI-C that is particularly close to register transfer language. Thus, the designer is often required to rewrite the C program manually in order to comply with these constraints. We extend the methodology to handle the full set of ANSI-C language features. This is a challenge in the presence of complex, dynamic data structures and pointers that may dynamically point to multiple objects. Furthermore, our methodology allows arbitrary loop constructs.

8.2 A Small Tutorial

8.2.1 Verilog and ANSI-C

The following Verilog module implements a 4-bit counter:

```
module top(input clk);

  reg [3:0] counter;

  initial counter=0;

  always @(posedge clk)
    counter=counter+1;

endmodule
```

CBMC can take Verilog modules as the one above as additional input. Similar as in co-simulation, the data in the Verilog modules is available to the C program by means of global variables. For the example above, the following C fragment shows the definition of the variable that holds the value of the `counter` register:

```
struct module_top {
  unsigned int counter;
};

extern const struct module_top top;
```

Using this definition, the value of the `counter` register in the Verilog fragment above can be accessed as `top.counter`. Please note that the name of the variable **must** match the name of the top module. The C program only has a view of one state of the Verilog model. The Verilog model makes a transition once the function `next_timeframe()` is called.

As CBMC performs Bounded Model Checking, the number of timeframes available for analysis must be bounded (SatAbs has no such restriction). As it is desirable to change the bound to adjust it to the available computing capacity, the bound is given on the command line and not as part of the C program. This makes it easy to use only one C program for arbitrary bounds. The actual bound is available in the C program using the following declaration:

```
extern const unsigned int bound;
```

Also note that the fragment above declares a constant variable of struct type. Thus, the C program can only read the trace values and is not able to modify them. We will later on describe how to drive inputs of the Verilog module from within the C program.

As described in previous chapters, assertions can be used to verify properties of the Verilog trace. As an example, the following program checks two values of the trace of the counter module:

```
void next_timeframe();

struct module_top {
  unsigned int counter;
};

extern const struct module_top top;

int main() {
  next_timeframe();
  next_timeframe();
  assert(top.counter==2);
  next_timeframe();
  assert(top.counter==3);
}
```

The following CBMC command line checks these assertions with a bound of 20:

```
hw-cbmc counter.c counter.v --module top --bound 20
```

Note that a specific version of CBMC is used, called `hw-cbmc`. The module name given must match the name of the module in the Verilog file. Multiple Verilog files can be given on the command line.

The `--bound` parameter is not to be confused with the `--unwind` parameter. While the `--unwind` parameter specifies the maximum unwinding depth for loops within the C program, the `--bound` parameter specifies the number of times the transition relation of the Verilog module is to be unwound.

8.2.2 Counterexamples

For the given example, the verification is successful. If the first assertion is changed to

```
  assert(top.counter==10);
```

and the bound on the command line is changed to 6, CBMC will produce a counterexample. CBMC produces two traces: One for the C program, which matches the traces described earlier, and a separate trace for the Verilog module. The values of the registers in the Verilog module are also shown in the C trace as part of the initial state.

```
Initial State
----------------------------------------------------
  bound=6 (00000000000000000000000000000110)
  counter= 0, 1, 2, 3, 4, 5, 6

Failed assertion: assertion line 6 function main

Transition system state 0
----------------------------------------------------
  counter=0 (0000)

Transition system state 1
----------------------------------------------------
  counter=1 (0001)

Transition system state 2
----------------------------------------------------
  counter=2 (0010)

Transition system state 3
----------------------------------------------------
  counter=3 (0011)

Transition system state 4
----------------------------------------------------
  counter=4 (0100)

Transition system state 5
----------------------------------------------------
  counter=5 (0101)

Transition system state 6
----------------------------------------------------
  counter=6 (0110)
```

8.2.3 Using the Bound

The following program is using the bound variable to check the counter value in all cycles:

```
void next_timeframe();
extern const unsigned int bound;

struct module_top {
  unsigned int counter;
};

extern const struct module_top top;

int main() {
  unsigned cycle;

  for(cycle=0; cycle<bound; cycle++) {
    assert(top.counter==(cycle & 15));
    next_timeframe();
```

```
  }
}
```

CBMC performs bounds checking, and restricts the number of times that `next_timeframe()` can be called. SatAbs does not require a bound, and thus, `next_timeframe()` can be called aribrarily many times.

8.2.4 Synchronizing Inputs

The example above is trivial as there is only one possible trace. The initial state is deterministic, and there is only one possible transition, so the verification problem can be solved by mere testing. Consider the following Verilog module:

```
module top(input clk, input i);

  reg [3:0] counter;

  initial counter=0;

  always @(posedge clk)
    if(i)
      counter=counter+1;

endmodule
```

Using the C program above will fail, as the Verilog module is free to use zero as value for the input `i`. This implies that the counter is not incremented. The C program has to read the value of the input `i` in order to be able to get the correct counter value:

```
void next_timeframe();
extern const unsigned int bound;

struct module_top {
  unsigned int counter;
  _Bool i;
};

extern const struct module_top top;

int main() {
  unsigned cycle;
  unsigned C_counter=0;

  for(cycle=0; cycle<bound; cycle++) {
    assert(top.counter==(C_counter & 15));
    if(top.i) C_counter++;
    next_timeframe();
  }
}
```

Similarly, the C model has to synchronize on the choice of the initial value of registers if the Verilog module does not perform initialization.

8.2.5 Driving Inputs

The C program can also restrict the choice of inputs of the Verilog module. This is useful for adding environment constraints. As an example, consider a Verilog module that has a signal `reset` as an input, which is active-low. The following C fragment drives this input to be active in the first cycle, and not active in any subsequent cycle:

```
__CPROVER_assume(top.resetn==0);

for(i=1; i<bound; i++) {
  __CPROVER_assume(top.resetn);
  next_timeframe();
}
```

Care must be taken to avoid passing the property *vacuously*. This happens if no trace actually satisfies the assumptions.

Mapping Variables within the Module Hierarchy Verilog modules are hierarchical. The `extern` declarations shown above only allow reading the values of signals and registers that are in the top module. In order to read values from sub-modules, CBMC uses structures.

As an example, consider the following Verilog file:

```
module counter(input clk, input [7:0] increment);

  reg [7:0] counter;

  initial counter=0;

  always @(posedge clk)
    counter=counter+increment;

endmodule

module top(input clk);

  counter c1(clk, 1);
  counter c2(clk, 2);

endmodule
```

The file has two modules: a top module and a counter module. The counter module is instantiated twice within the top module. A reference to the register `counter` within the C program would be ambiguous, as the two

module instances have separate instances of the register. CBMC and SatAbs use the following data structures for this example:

```
void next_timeframe();
extern const unsigned int bound;

struct counter {
  unsigned char increment;
  unsigned char counter;
};

struct module_top {
  struct module_counter c1, c2;
};

extern const struct module_top top;

int main() {
  next_timeframe();
  next_timeframe();
  next_timeframe();
  assert(top.c1.counter==3);
  assert(top.c2.counter==6);
}
```

The `main` function reads both counter values for cycle 3. A deeper hierarchy (modules in modules) is realized by using additional structure members. Writing these data structures for large Verilog designs is error prone, and thus, CBMC can automatically generate them. The declarations above are generated using the command line

```
hw-cbmc --gen-interface --module top hierarchy.v
```

Mapping Verilog Vectors to Arrays or Scalars In Verilog, a definition such as

```
wire [31:0] x;
```

can be used for arithmetic (e.g., `x+10`) and as array of Booleans (e.g., `x[2]`). ANSI-C does not allow both, so when mapping variables from Verilog to C, the user has to choose one option for each such variable. As an example, the C declaration

```
unsigned int x;
```

will allow using `x` in arithmetic expressions, while the C declaration

```
_Bool x[32];
```

will allow accessing the individual bits of `x` using the syntax `x[bit]`. The `--gen-interface` option of CBMC will generate the first variant if the vector has the same size as one of the standard integer types, and the second option if not so. This choice can be changed by adjusting the declaration accordingly.

Appendix A CBMC and SATABS License Agreement

Appendix B Programming APIs

B.1 Language Frontends

B.1.1 Scanning and Parsing

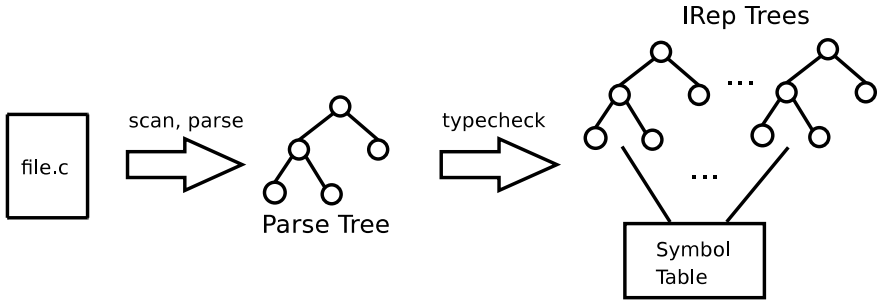

Figure B.1: From C source code file to IRep

The sources of the C frontend are located in the "ansi-c/" directory. It uses a standard Flex/Bison setup for scanning and parsing the files. The Bison grammar produces a tree representation of the input program. The typechecker annotates this parse tree with types and generates a symbol table.

The following code illustrates how to use the frontend for parsing files and for translating them into a symbol table. A call to *parse* generates the parse tree of the program. The conversion into the symbol table is performed during type checking, which is done by a call to the *typecheck* method. The symbol table is a map from identifiers to the *symbolt* data structure.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

#include <ansi-c/ansi_c_language.h>
#include <util/cmdline.h>
#include <util/config.h>

int main(int argc, const char* argv[])
{
  // Command line: parse -I incl_dir file1 ...
  cmdlinet cmdl;
  cmdl.parse(argc, argv, "I:");

  config.init();
```

```
    if(cmdl.isset('I'))
      config.ansi_c.include_paths=cmdl.get_values('I');

    // Set language to C
    std::auto_ptr<languaget> clang(new_ansi_c_language());

    // Symbol table
    contextt my_context;

    for(cmdlinet::argst::iterator sit=cmdl.args.begin();
        sit != cmdl.args.end();sit++)
    {
      // Source code stream
      std::ifstream in(sit->c_str());

      // Parse
      clang->parse(in, "", std::cerr);

      // Typecheck
      clang->typecheck(my_context, *sit, std::cerr);
    }

    // Do some final adjustements
    clang->final(my_context, std::cerr);

    my_context.show(std::cout);

    return 0;
}
```

B.1.2 IRep

The parse trees are implemented using a class called *irept*. Its declaration and definiton can be found in the files "util/irep.h" and "util/irep.cpp".

The code below shows some details of class *irept*:

```
class irept
{
public:
  typedef std::vector<irept> subt;
  typedef std::map<irep_name_string, irept> named_subt;
  ...

public:
  class dt
  {
  public:
    unsigned ref_count;
    dstring data;
    named_subt named_sub;
    named_subt comments;
    subt sub;
```

```
    ...
  };

protected:
  dt *data;
  ...
};
```

Every node of any tree is an object of class *irept*. Each node has a pointer to an object of class *dt*. The *dt* objects are used for storing the actual content of nodes. Objects of class *dt* are dynamically allocated and can be shared between nodes. A reference-counter mechanism is implemented to automatically free unreachable *dt* objects. Copying a tree is an $O(1)$ operation.

The field *data* of class *dt* is a (hashed) string representing the label of the nodes. The fields *named_sub*, *comments* and *sub* are links to childs. Edges are either labeled with a string or ordered. The string-labeled edges are stored in the map *comments* if their first character is '#'. Otherwise, they are stored in the map *named_sub*. The labels of edges are unique for a given node; however, their ordering is not preserved. The field *sub* is a vector of nodes that is used for storing the ordered children. The order of edges of this kind is preserved during copy.

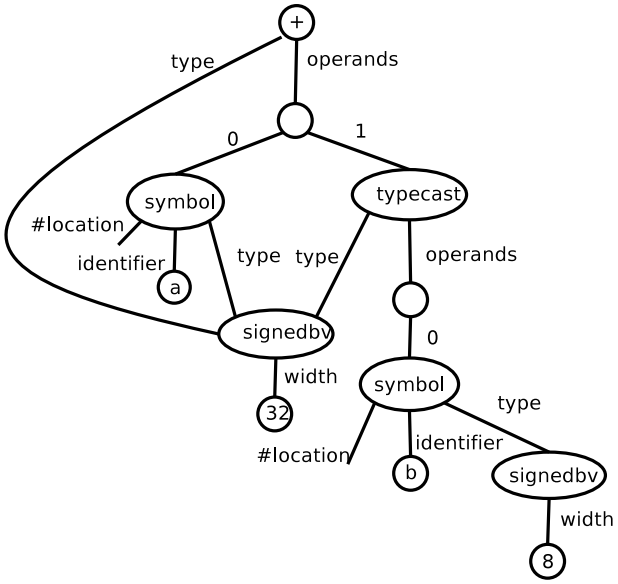

Figure B.2: Tree for the expression *a+b* with *int a; char b;*.

Interface of Class *irept*

```
virtual bool is_nil() const;
virtual bool is_not_nil() const;
```

The first method returns true if the label of the node is equal to "nil". The second method returns false if the label of the node is equal to "nil".

```
const irep_idt &id();
void id(const irep_idt &_data);
```

The first method returns a constant reference to the label of the node. The second method sets the label of the node.

```
const irept &find(const irep_namet &name) const;
irept &add(const irep_namet &name);
const irep_idt &get(const irep_namet &name) const;
```

- The first method looks for an edge with label *name* and returns the corresponding child. If no edge with label *name* is found, then *nil_rep* is returned.
- The second method does the same as the first except that if no edge with label *name* if found, then a new child is created and returned.
- The third method does the same as the first except that the label of the child is returned (instead of a reference). If no edge with label *name* is found, then an empty string is returned.

```
void set(const irep_namet &name,
         const irep_idt &value);
void set(const irep_namet &name, const long value);
void set(const irep_namet &name, const irept &irep);
```

These methods create a new edge with label *name*.

If the second argument is an object of class *irept*, then it is assigned to the new child.

If the second argument is a string, then it is set as node-label of the new child.

If the second argument is a number, then it is converted to a string and set as node-label of the new child.

```
void remove(const irep_namet &name);
```

This method looks for an edge with label *name* and removes it.

```
void move_to_sub(irept &irep);
void move_to_named_sub(const irep_namet &name, irept &irep);
```

The first method creates a new ordered edge with a child equal to *irep*. Then it sets *irep* to *nil*. The index of the edge is equal to the size of vector *sub* before the call.

The second method does the same but for labeled edges.

```
void swap(irept &irep);
```

Exchange the content of the invoked node with the one of *irep*.

```
void make_nil();
```

Set the label of the node to "nil" and remove all outgoing edges.

```
const subt &get_sub();
const named_subt &get_named_sub();
const named_subt &get_comments();
```

Return a constant reference to *sub*, *named_sub*, and *comments*, respectively.

B.1.3 Types

The class *typet* inherits from *irept*. Types may have subtypes. This is modeled with two edges named "subtype" and "subtypes". The class *typet* only add specialized methods for accessing the subtype information to the interface of *irept*.

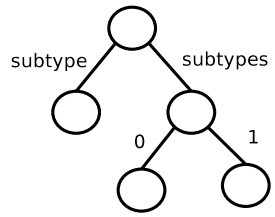

Figure B.3: A Type Tree

Interface of class *typet*

```
bool has_subtype() const;
bool has_subtypes() const;
```

The first method returns true if the a subtype node exists. is not *nil*. The second method returns true is a subtypes node exists.

```
typet &subtype();
typest &subtypes();
```

The first method returns a reference to the 'subtype' node. The second method returns a reference to the vector of subtypes.

B.1.4 Subtypes of `typet`

A number of subtypes of `typet` exist which allow convenient creation and manipulation of `typet` objects for special types.

Class	Description
util/std_types.h	
`bool_typet`	Boolean type
`symbol_typet`	Symbol type. Has edge "identifier" to a string value, which can be accessed with `get_identifier` and `set_identifier`.
`struct_typet`, `union_typet`	Represent a struct, resp. union types. Convenience functions to access components `components()`.
`code_typet`	The type of a function/procedure. Convenience functions to access `arguments()` and `return_type()`.
`array_typet`	Convenience function `size()` to access size of the array.
`pointer_typet`	Pointer type, subtype stores the type of the object pointed to.
`reference_typet`	Represents a reference type, subtype stores the type of the object referenced to.
`bv_typet`	Represents a bit vector type with variable width.
`fixed_bv_typet`	Represents a bit vector that encodes a fixed-point number.
`floatbv_typet`	Represents a bit vector that encodes a floating-point number.
`string_typet`	Represents a string type.

B.1.5 Location

The class *locationt* inherits from the class *irept*. It is used to store locations in text files. It adds specialized methods to manipulate the edges named "file", "line", "column", "function".

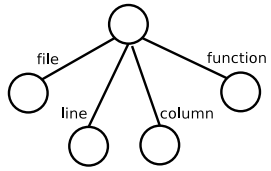

Figure B.4: A location tree

B.1.6 Expressions

The class *exprt* inherits from class *irept*. Expressions have operands and a type. This is modeled with two edges labeled "operands" and "type", respectively. The class *exprt* only adds specialized methods for accessing operands and type information to the interface of *irept*.

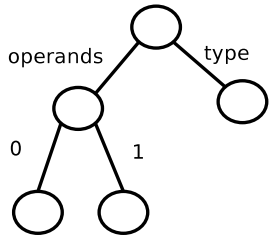

Figure B.5: A binary expression

Interface of class *exprt*

```
explicit exprt(const irep_idt &id);
```

Creates an *exprt* object with a given label and no type.

```
exprt(const irep_idt &id, const typet &type);
```

Creates an *exprt* object with a given label and type.

```
const typet &type() const;
typet &type();
```

Return a reference to the 'type' node

```
bool has_operands() const;
```

Return true if the expression has operands.

```
const operandst &operands() const;
```

Return a reference to the vector of operands.

```
const exprt &op0();
const exprt &op1();
const exprt &op2();
const exprt &op3();
exprt &op0();
exprt &op1();
exprt &op2();
exprt &op3();
```

Return a reference to a specific operand.

```
void make_true();
void make_false();
void make_bool(bool value);
```

Turn the current *exprt* instance into a expression of type "bool" with label "constant" and a single edge labeled "value", which points to a new node with label either "true" or "false".

```
void make_typecast(const typet &_type);
```

Turns the current *exprt* instance into a typecast. The old value of the instance is appended as the single operand of the typecast, i.e., the result is a typecast-expression of the old expression to the indicated type.

```
void make_not();
```

Turns the current *exprt* instance into an expression with label "not" of the same type as the original expression. The old value of the instance is appended as the operand of the "not"-node. If the original expression is of type "bool", the result represents the negation of the original expression with the following simplifications possibly applied:

- $\neg\neg f = f$
- $\neg$true = false
- $\neg$false = true

```
void negate();
```

Turns the current *exprt* instance into a negation of itself, depending on its type:

- For boolean expressions, `make_not` is called.

- For integers, the current instance is turned into a numeric negation expression "unary-" of its old value. Chains of "unary-" nodes and negations of integer constants are simplified.
- For all other types, `irept::make_nil` is called.

```
bool sum(const exprt &expr);
bool mul(const exprt &expr);
bool subtract(const exprt &expr);
```

Expect the "this" object and the function argument to be constants of the same numeric type. Turn the current `exprt` instance into a constant expression of the same type, whose "value" edge points to the result of the sum, product, or difference of the two expressions. If the operation fails for some reason (e.g., the types are different), `true` is returned.

```
bool is_constant() const;
```

Returns true if the expression label is "constant".

```
bool is_boolean() const;
```

Returns true if the label of the type is "bool".

```
bool is_false() const;
bool is_true() const;
```

The first function returns true if the expression is a boolean constant with value "false". The second function returns true for any boolean constant that is not of value "false".

```
bool is_zero() const;
bool is_one() const;
```

The first function returns true if the expression represents a zero numeric constant, or if the expression represents a null pointer. The second function returns true if the expression represents a numeric constant with value "1".

B.1.7 Subtypes of `exprt`

A number of subtypes of `exprt` provide further convenience functions for edge access or other specialized behaviour:

Class	Description
util/std_expr.h	
`transt`	Represents a SMV-style transition system with invariants `invar()`, initial state `init()` and transition function `trans()`.
`true_exprt`	Boolean constant true expression.
`false_exprt`	Boolean constant false expression.

Class	Description
`symbol_exprt`	Represents a symbol (e.g., a variable occurrence), convenience function for manipulating "identifier"-edge `set_identifier` and `get_identifier`
`predicate_exprt`	Convenience constructors to create expressions of type "bool".
`binary_relation_exprt : predicate_exprt`	Convenience functions to create and manipulate binary expressions of type "bool".
`equality_exprt : binary_relation_exprt`	Convenience functions to create and manipulate equality expressions such as "a == b".
`ieee_float_equal_exprt : binary_relation_exprt`	Convenience functions to create and manipulate equality expressions between floating-point numbers.
`index_exprt`	Represents an array access expression such as "a[i]". Convenience functions `array()` and `index()` for accessing the array expressions and indexing expression.
`typecast_exprt`	Represents a cast to the type of the expression.
`and_exprt`, `implies_exprt`, `or_exprt`, `not_exprt`	Representations of logical operators with convenience constructors.
`address_of_exprt`	Representation of a C-style `&a` address-of operation. Convenience function `object()` for accessing operand.
`dereference_exprt`	Representation of a C-style `*a` pointer-dereference operation. Convenience function `object()` for accessing operand.
`if_exprt`	Representation of a conditional expresion, with convenience functions `cond()`, `true_case()` and `false_case()` for accessing operands.
`member_exprt`	Represents a `some_struct.some_field` member access.

B.1.8 Symbols and the Symbol Table

Symbol

A symbol is an object of class *symbolt*. This class is declared in "util/symbol.h". The code below shows a partial declaration of the interface:

```
class symbolt
{
public:
  typet type;
  exprt value;
  std::string name;
```

```
  std::string base_name;
  ...
};
```

Symbol names are unique. Scopes are handled by adding prefixes to symbols:

```
int main(int argc, char* argv[]) {

                          // Symbol name: c::main::0::alice
  char alice = 0;         // Symbol base: alice

  {
                          // Symbol name: c::main::1::alice
    int alice = 0;        // Symbol base: alice
  }
}
```

Symbol Table

A symbol table is an object of class *contextt*. This class is declared in "util/-context.h". The code below shows a partial declaration of the interface:

```
class contextt
{
public:
                                    // Insert the symbol
  bool add(const symbolt &symb);
                                    // Insert symb into the
                                    // table and erase it.
                                    // New_symbol points to the
                                    // newly inserted element.
  bool move(symbolt &symbol, symbolt *&new_symbol);

                                    // Insert symb into the
                                    // table. Then symb is erased.
  bool move(symbolt &symb);

                                    // Return the entry of the
                                    // symbol with given name.
  const irept &value(const std::string &name) const;
};
```

B.2 Goto Programs

Goto programs are a representation of the control flow graph of a program that uses only guarded goto and assume statements to model non-sequential flow. The main definition can be found in "goto-programs/ goto_program_template.h", which is a template class. The concrete instantiation of the template that is used in the framework can be found in "goto-programs/ goto_program.h". A single instruction in a goto program is rep-

resented by the class *goto_programt:: instructiont* whose definition can be found again in "goto-programs/ goto_program_template.h".

In the class `goto_programt`, the control flow graph is represented as a mixture of sequential transitions between nodes, and non-sequential transitions at goto-nodes. The sequential flow of the program is captured by the list `instructions` that is a field of the class `goto_programt`. Transitions via goto statements are represented in the list `targets`, which is a field of the class `goto_programt::instructiont`, i.e., each goto-instruction carries a list of possible jump destinations. The latter list `targets` is a list of iterators which point to elements of the list `instructions`. An illustration is given in Figure B.6.

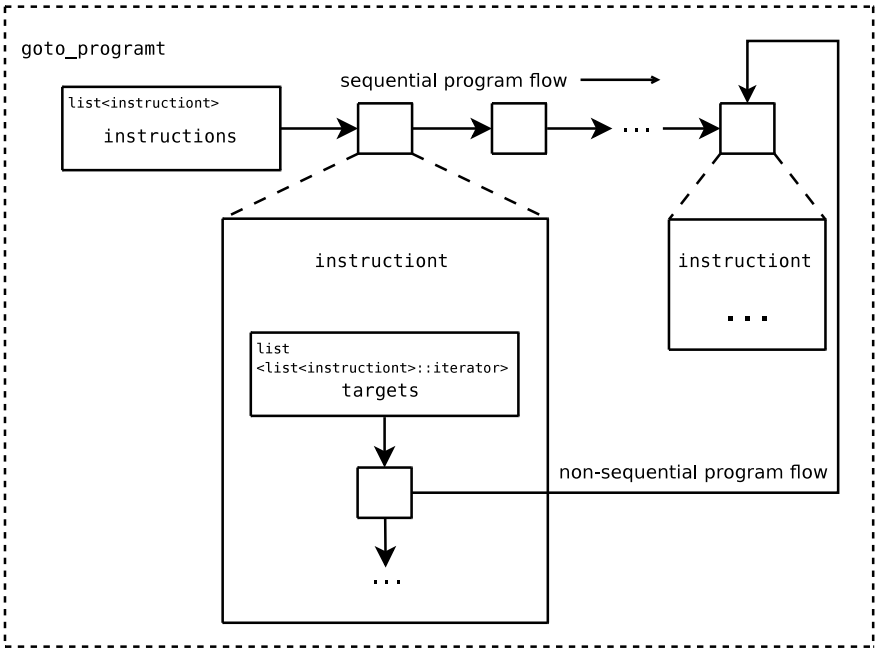

Figure B.6: Representation of program flow in `goto_programt`

Instructions can have a number of different types as represented by `enum goto_program_instruction_typet` and can be accessed via the field `type` in `instructiont`. These include:

`GOTO` : Represents a non-deterministic branch to the instructions given in the list `targets`. Goto statements are guarded, i.e., the non-deterministic branch is only taken if the expression in `guard` evaluates to true, otherwise the program continues sequentially. Guarded gotos can be used, for example, to model if statements. The guard is then set to the negated condition of the statement, and goto target is set to bypass the conditionally executed code if this guard evaluates to true.

`ASSUME` : An assumption statement that restricts viable paths reaching the instruction location to the ones that make the expression `guard` evaluate to true.

`ASSERT` : An assertion whose `guard` is checked for validity when the instruction is reached.

`RETURN` : A return statement in a function.

`FUNCTION_END` : Denotes the end of a function.

`ASSIGN` : A variable assignment.

`SKIP` : No operation.

`OTHER` : Any operation not covered by `enum goto_program_instruction_typet`.

A number of convenience functions in `instructiont`, such as `is_goto()`, `is_assume()`, etc., simplify type queries. The following code segment shows a partial interface declaration of `goto_program_template` and `instructiont`.

```
template <class codeT, class guardT>
class goto_program_templatet
{
public:
  //list of instruction type
  typedef std::list<class instructiont> instructionst;

  //a reference to an instruction in the list
  typedef typename
    std::list<class instructiont>::iterator targett;

  //Sequential list of instructions,
  //representing sequential program flow
  instructionst instructions;

  typedef typename
    std::map<const_targett, unsigned> target_numberst;

  //A map containing the unique number of each target
  target_numberst target_numbers;

  //Get the successors of a given instruction
  void get_successors(targett target, targetst &successors);

  ...


  class instructiont
  {
  public:
    codeT code;

    //identifier of enclosing function
    irep_idt function;

    //location in the source file
    locationt location;

    //type of instruction?
    goto_program_instruction_typet type;

    //Guard statement for gotos, assume, assert
    guardT guard;
```

```
    //targets for gotos
    targetst targets;

    //set of all predecessors (sequential, and gotos)
    std::set<targett> incoming_edges;

    // a globally unique number to identify a
    // program location. It is guaranteed to be
    // ordered in program order within one
    // goto_program
    unsigned location_number;

    // a globally unique number to identify loops
    unsigned loop_number;

    // true if this is a goto jumping back to an
    // earlier instruction in the sequential program
    // flow
    bool is_backwards_goto() const;
  };

}
```

B.3 Static Analysis

B.3.1 A Brief Introduction to Abstract Interpretation

The theory of abstract interpretation approximates the semantics of formulas and in our case of programs. Usually the semantics of a program is defined by a (concrete) domain D_c and the relationships within this domain (changing over time by executing some instructions). Given an interpretation I based on the domain D_c the question of whether a property expressed by a logical formula p holds, i.e., $(D_c, I) \models p$, can be answered via model checking. The state explosion entailed by the number of possible execution paths and the undecidability caused by infinite models, however, is a major hurdle toward scalability to large software systems.

To overcome these difficulties a particular approach is to approximate the concrete domain D_c via a so called abstract domain D_a, i.e., a concrete domain of values D_c is replaced by an (abstract) domain of descriptions of values D_a. Even if an abstract domain is not as precise as its concrete counterpart, it still permits to answer some questions using static analysis for instance.

The framework of abstract interpretation was introduced and formalized by P. Cousot and R. Cousot in 1977 [1]. It relies on lattice theory and Galois connections and we refer to [2] for an in-depth description of abstract interpretation. Nevertheless, we will now explain the basics needed to understand the class interface of `abstract_domain_baset`, `static_analysis` and its base class `static_analysis_baset`.

Definitions To be a useful abstraction an abstract domain D_a should form a *lattice*. The latter is a partially ordered set with the pre-order relation $\sqsubseteq$ where every subset $S \subseteq D_A$ has a least upper bound denoted by $\sqcup S$ as well as a greatest lower bound $\sqcap S$. The *supremum* $\top$ of D_a is the least upper bound of D_a and the $infimum$ $\bot$ of D_a is the greatest lower bound of D_a. The abstract domain is then denoted by $D_a(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$. In the context of abstract interpretation, the pre-order relation $\sqsubseteq$ can intuitively be interpreted as a precision pre-ordering which tells us if a description of values $a \in D_a$ is more (less or equally) precise than another description of values $b \in D_a$.

B.3.2 Class Interfaces

The base class `abstract_domain_baset`

This abstract class establishes the base for the data structure needed to abstract the concrete computation domain. Other classes will derive from it in order to refine the abstract domain we would like to implement.

Type definition	Description
`locationt`	A location is an iterator pointing to an instruction of an instruction list of a goto program.

Type Definitions

Attribute	Description
`bool:seen`	specifies if an instance (element) of the abstract domain `abstract_domain_baset` has already been visited by the `visit` member function of the abstract `static_analysis_baset` class.

Attributes

Member function	Description
`initialize (&ns, l)`	pure member function that will be defined in specializations of this class. It is intended to set the state (see below `statet`) associated to the location `l` to the infinmum $\bot$ of the lattice D_a (abstract domain).

Member function	Description
`transform (&ns, from_l, to_l)`	pure virtual member function that will be defined in specializations of this class. It is intended to strengthen the state of `this` with respect to the guard that applies when the program transitions from the location `from_l` to the successor location `to_l`. Generally these guards vary depending on the location type (i.e. if it is a conditional, assignment or declaration location).
`exprt:get_guard (from, to)`	returns the guard expression of a conditional location `from` (in the control flow of the corresponding goto program) jumping to `to`.
`exprt:get_return _lhs (to)`	if `to` is a location after a function call, returns the right hand side of the expression that assigns the returned values to the corresponding variables of the caller.

Member Functions

The base class `static_analysis_baset`

The class `static_analysist` and its base `static_analysis_baset` are used to perform the static analysis with the appropriate abstract domain.

Type definition	Description
`locationt`	a location is an instruction list of a goto program.
`statet`	as already mentioned above, a location is an iterator pointing to an instruction of an instruction list of a goto program.
`working_sett`	set of locations that still have to be visited by the `visit` member function during the calculation of the fixed point (by the `fixed_point` member function).

Type Definitions

Attribute	Description
`namespacet:ns`	namespace that is to be considered during the analysis of the corresponding goto program.
`bool:initialized`	specifies if for each location of the goto program a state has been generated (see the `generate_states` member function of the `static_analysis_baset` class).

Attributes

Member function	Description
`initialize (&goto_program)`	for each location of the `goto_program` this member function generates an associated state (see `generate_state` below) and sets the attribute `initialized` to `true`.
`initialize (&goto_function)`	for each location of each function in `goto_functions` this member function generates an associated state (see `generate_state` below) and sets the attribute `initialized` to `true`.
`locationt: successor(l)`	returns the successor location of `l` in the control flow graph (CFG).
`bool:visit(l, &working_set, goto_program, &goto_functions)`	marks the state associated to the location `l` as visited, then crawls through the successor locations of `l` and, for each pair consisting of the location `l` and one of its successor, merges their corresponding states (`abstract_domain_baset` element). If the state of `l` has been changed or its successor location has not yet been seen, then it puts this successor location in the `working_set`. Returns `true` if the state of `l` has been changed. See the definition of `merge` for more details.
`statet: get_state(l)`	pure member function defined in specializations of this class (see the interface of the specialization `static_analysist` below.
`generate_states (&goto_program)`	for each location of the `goto_program` this member function sets the corresponding state to the infinmum $\bot$ of the abstract domain D_a (see the member function `generate_state` of the `static_analysist` class below).
`generate_states (&goto_functions)`	for each location of each function in `goto_functions` this member function sets the corresponding state to the infinmum $\bot$ of the abstract domain D_a.
`insert(l)`	generates the state, that is an element of the abstract domain, at location `l` (see `generate_state` below).

Member Functions

The class `static_analysist`

This class performs the static analysis with the abstract domain specified through its parameter `T`. The latter should be an instantiatable class inher-

iting from abstract_domain_baset.

Type definition	Description
state_mapt	A state map is a location associated with its abstract domain element i.e. an instance of the abstract_domain_baset class (through some specialization/subtyping of type T).

Type Definitions

Attribute	Description
state_mapt: state_map	Maps an abstract domain element to a location of a goto program (see the type definition above).

Attributes

Member function	Description
statet: get_state (locationt l)	searches for the location l in the state_map and returns the associated state (which is an element of the abstract_domain_baset_class).
T:operator[] (locationt l)	searches for the location l in the state_map like get_state does, but returns the associated state which is now of type T.
bool: has_location(locationt l)	returns true if the location l is found in the state map.
clear()	this member function clears the state_map and sets the attribute initialized to false.
statet*: make_temporary_ state (statet &s)	casts and duplicates the state s to a state of type T (specialization of abstract_domain_baset).
bool: merge(statet &a, statet &b)	modifies the state a so to be the least upper bound of both states a and b i.e. $\sqcup\{a, b\}$ where $\{a, b\} \subseteq D_a$. It is needed to obtain the new state after a join in the control flow of a goto program for instance. The return value is true if a has been modified.
generate_state (locationt l)	sets the state associated to the location l to the infinmum of the abstract domain (concrete class of type T which is a specialization of the abstract class abstract_domain_baset).

Member function	Description
`fixedpoint (goto_programt goto_program, goto_functionst goto_functions)`	computes the least fixed point of the system of equations defined by the constraints on each state of each location of the goto program `goto_program`.

Member Functions

B.3.3 Examples

Before running a static analysis, one has to specify the abstract domain D_a which will approximate a concrete domain D_c. To do so, we need to refine the `abstract_domain_baset` class with some specialization. This instantiatable class will be passed via the parameter `T` of the `static_analysist` class. In the following sections we will show what has to be done in order to implement the simple following example borrowed from [3].

Sign Analysis. The goal of sign analysis is to track the sign of each expression in a program. To do so, we define the lattice $\{\bot, +, 0, -, \top\}$ where each element stands for subsets of the concrete domain of signed integers. + represents the set of strictly positive, - the set of strictly negative values whereas 0 represents the singleton set $\{0\}$ containing only zero. $\top$ denotes any value (i.e. the expression is known to have a varying sign) and $\bot$ denotes no value (i.e. the expression has an unknown value). Since we want to track the sign of each expression at a program location the full lattice is give by the map lattice $Vars \mapsto \{\bot, +, 0, -, \top\}$. Hence, at each program location `l` we assign a table that associates a sign to each variable.

Refining the `abstract_domain_baset` class

Class interface A derivation of the base class `abstract_domain_baset` implementing our sign analysis should have a data structure representing the sign lattice and the full map lattice we defined above. Furthermore it should define the following pure virtual member functions from the base class:

- `transform(namespacet &ns, locationt from_l, locationt to_l)` and,
- `initialize(namespacet &ns, locationt l)`,

which purposes are described above. Since the `merge` member function of the `static_analysist` class will call a `merge` member function of our `abstract_domain_baset` class, we also have to define such a `merge` function with the following interface:

- `bool:merge(statet &other)`: as one can guess from the interface description above, after a call to this member function the receiver `this` becomes the least upper bound of the `abstract_domaint` element `other` and the receiver `this`.

One possible implementation of such a derivation from the base class is showed in the following listing:

```
#ifndef SIGN_MAP_H_
#define SIGN_MAP_H_

#include "expr.h"
#include "irep.h"
#include "static_analysis.h"
#include "goto_functions.h"
#include <map>

class sign_mapt: public abstract_domain_baset
{
public:
    typedef enum signt
        {
            ZERO,
            PLUS,
            MINUS,
            TOP,
            BOTTOM
        } sign;
    typedef std::map<irep_idt, signt> sign_latticet;

    // an abstract domain element maps a sign to a given set
    // of expressions (typically those handling integers at a
    // given location}
    sign_latticet sign_lattice;


    virtual void initialize(const namespacet &ns, locationt l);

    virtual void transform(const namespacet &ns, locationt from_l, locationt to_l);

    bool merge(sign_mapt &other);

};

#endif /*SIGN_MAP_H_*/
```

Defining the `merge` member function

```
bool sign_mapt::merge(sign_mapt &other)
{
    // boolean tracking the changes to the state of "this"
    bool any_changes=false;
    typedef std::map<irep_idt, signt>::const_iterator map_itt;

    for(map_itt map_it= other.sign_lattice.begin(); map_it!=other.sign_lattice.end(); ++map_it)
    {   //if the lattice_map also has an entry for the term/vairable
        if(&sign_lattice[map_it->first]!= NULL)
        {
            switch(map_it->second)
            // select the corresponding lattice element (least upper bound
            // of the pair {"this", "other"}. Report any change
            {
                case BOTTOM:
                {   sign_lattice[map_it->first] = other.sign_lattice[map_it->first];
                    any_changes=true;
                }
                break;
```

```
                case TOP:
                 // do nothing
                break;

                case PLUS:
                 if(sign_lattice[map_it->first]==MINUS)
                 {
                    sign_lattice[map_it->first]=TOP;
                    any_changes=true;
                 }
                 else if(sign_lattice[map_it->first]==TOP)
                 {
                    sign_lattice[map_it->first]=TOP;
                    any_changes=true;
                 }

                case MINUS:
                 if(sign_lattice[map_it->first]==PLUS)
                 {
                    sign_lattice[map_it->first]=TOP;
                    any_changes=true;
                 }
                 // other cases:do nothing
            }
        }
        // if the lattice_map has no such entry, then simply add it
        else
        {
            sign_lattice[map_it->first]= other.sign_lattice[map_it->first];
        }
    }
    return any_changes;
}
```

Defining the `initialize` member function

```
void sign_mapt::initialize(const namespacet &ns, locationt l)
{
    typedef std::set<irep_idt>::const_iterator irep_itt;

    // for each local variable at location l add an entry (variable, BOTTOM)
    // in the sign_map (we initialize to the infimum of the lattice)
    for(irep_itt irep_it=l->local_variables.begin(); irep_it!=l->local_variables.end(); ++irep_it)
    {
        sign_lattice[*irep_it] = BOTTOM;
    }
}
```

B.3.4 Using the parametrized `static_analysist` class

Once the abstract domain D_a has been implemented through the specialization `sign_mapt`, we apply it by refining the parametrized class `static_analysist<sign_mapt>`.

B.4 Propositional Logic

Bibliography

[1] C. Blank, H. Eveking, J. Levihn, and G. Ritter. Symbolic simulation techniques — state-of-the-art and applications. In *International Workshop on High-Level Design, Validation, and Test*, pages 45–50. IEEE, 2001.

[2] David W. Currie, Alan J. Hu, and Sreeranga Rajan. Automatic formal verification of dsp software. In *Proceedings of the 37th Design Automation Conference (DAC 2000)*, pages 130–135. ACM Press, 2000.

[3] Kiyoharu Hamaguchi. Symbolic simulation heuristics for high-level design descriptions with uninterpreted functions. In *International Workshop on High-Level Design, Validation, and Test*, pages 25–30. IEEE, 2001.

[4] Carl Pixley. Guest Editor's Introduction: Formal Verification of Commercial Integrated Circuits. *IEEE Design & Test of Computers*, 18(4):4–5, 2001.

[5] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Int. Journal of Software Tools for Technology Transfer (STTT)*, 2(2):192–201, 1998.

[6] http://www.systemc.org.

[7] Luc Séméria, Andrew Seawright, Renu Mehra, Daniel Ng, Arjuna Ekanayake, and Barry Pangrle. RTL C-based methodology for designing and verifying a multi-threaded processor. In *Proc. of the 39th Design Automation Conference*, pages 123–128. ACM Press, 2002.