FoREnSiC

A Formal Repair Environment for Simple C

Version 0.9

Authors
Roderick Bloem, Rolf Drechsler, Görschwin Fey, Alexander Finder,
Georg Hofferek, Robert Könighofer, Jaan Raik, Urmas Repinski, André Sülflow

Notices

This tool has been developed within the DIAMOND European project, contract number FP7-2009-IST-4-248613 (`http://www.fp7-diamond.eu/`). The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

For more information about the tool, contact the FoREnSiC developers directly (`forensic@lists.iaik.tugraz.at`).

Contents

List of Figures

List of Tables

List of Abbreviations

ANSI	-	American National Standards Institute
API	-	Application Programming Interface
AST	-	Abstract Syntax Tree
GCC	-	GNU Compiler Collection
GIMPLE	-	An intermediate representation used in GNU compilers
ISO	-	International Organization for Standardization
LHS	-	Left-hand side
MAX-SAT	-	Maximum Satisfiability
RHS	-	Right-hand side

RTL	-	Register Transfer Level
SAT	-	(Boolean) Satisfiability
SMT	-	Satisfiability Modulo Theories
SSA	-	Static Single Assignment
UML	-	Unified Modeling Language

Chapter 1

Introduction

Many methods and tools exist to automate error detection in software or hardware. The techniques range from automatic test case generation and execution to formal methods like model checking. But once an error has been detected, the difficult part of the debugging work only begins: the error has to be located and corrected. This is usually done manually. Manually locating and correcting errors is time-consuming, frustrating, and costly. On the other hand, performing these tasks automatically is difficult, both regarding methodology and computational complexity. FoREnSiC addresses these challenges.

FoREnSiC is a tool to automate error localisation and correction for C programs. FoREnSiC stands for "**Fo**rmal **R**epair **En**vironment for **Si**mple **C**", but actually the title is not fully accurate any more. FoREnSiC has grown to be more. First of all, FoREnSiC is not purely formal. The techniques implemented in FoREnSiC range from simulation-based methods to semi-formal and formal ones. Second, it does not only address repair of programs but also error detection and localisation. Finally, FoREnSiC cannot only debug C programs but also hardware, with a C program functioning as a specification for this hardware.

1.1 An Appetizer

Let's have a look at an example to see what FoREnSiC is all about. Consider the following function (see `examples/max.c`).

```
4  int max(int x, int y) {
5    int res = x;
6    if(y > x)
7      res = x;
8    assert(res >= x && res >= y);
9    return res;
10 }
```

It is supposed to compute the maximum of two integer numbers, but it contains a bug in Line 7. Instead of `x`, the programmer should have written `y`. The

variables `x` and `y` are inputs to the function. A specification of the function is given in form of an assertion. For some input values the specification is satisfied but for others it is violated, so the program is not fully correct.

Instead of thinking hard what the problem in this program might be and how it can be fixed, we can simply run FoREnSiC to analyze the program for us. We get the following output[1]:

```
[RES] Diagnoses:
[RES] Line 7: "x"
[RES] Line 5: "x"   and   Line 6: "y > x"
[RES] Repairs:
[RES] Replace   Line 7: "x"   by   "y"
[RES] Replace   Line 7: "x"   by   "y + 1"
[RES] Replace   Line 7: "x"   by   "y + 2"
[RES] Replace   Line 5: "x"   by   "y + 1000"
          and   Line 6: "y > x"   by   "-x + y <= 0"
[RES] Replace   Line 5: "x"   by   "y + 999"
          and   Line 6: "y > x"   by   "x - y > 0"
[RES] Replace   Line 5: "x"   by   "y + 999"
          and   Line 6: "y > x"   by   "x - y >= 0"
```

The reported diagnoses express that FoREnSiC suspects either the `x` in Line 6 or the expressions in both the lines 4 and 5. It also suggests replacements for these expressions. Not all of them are what one would expect for a function which computes the maximum. The reason is that the specification is not complete. It does not require that `res` is either `x` or `y`. If the specification is refined in this respect (`examples/max2.c`), FoREnSiC suggests the following fixes.

```
[RES] Replace   Line 7: "x"   by   "y"
[RES] Replace   Line 7: "x"   by   "res - x + y"
[RES] Replace   Line 7: "x"   by   "-res + x + y"
[RES] Replace   Line 5: "x"   by   "y"
          and   Line 6: "y > x"   by   "x - y >= 0"
[RES] Replace   Line 5: "x"   by   "y"
          and   Line 6: "y > x"   by   "-x + y <= 0"
[RES] Replace   Line 5: "x"   by   "y"
          and   Line 6: "y > x"   by   "-x + y < 0"
```

For this simple program, FoREnSiC was able to compute error locations and corrections in a fraction of a second.

1.2 Features and Benefits

Automatic error localisation and correction are very challenging tasks, especially for software. The ideal automatic debugging method can handle all features provided by the programming language, it is able to diagnose and fix any bug, it pinpoints potential error locations precisely and without false-positives, it suggests fixes which are not only correct but also easily understandable, and it

[1] To reproduce the output execute `build/src/forensic-bin -i ../examples/max.c -b syb --syb_cond=True --syb_rep_mrpd=3 --print=RWE` from a shell in the directory `tool`.

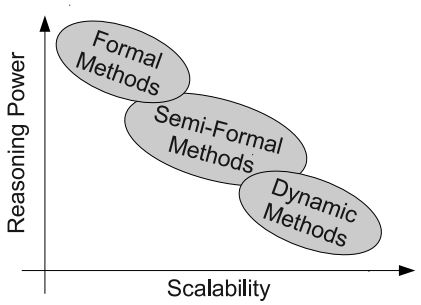

Figure 1.1: Different characteristics of different debugging methods.

scales to huge programs. Clearly, these goals cannot all be maximized at the same time; different techniques provide different trade-offs.

Figure 1.1 depicts a rough classification of debugging methods together with their main characteristics. Formal methods typically transform the program into logic formulas and use logic solving to find diagnoses and repairs. They provide high reasoning power but are computationally expensive. Dynamic methods execute the model with a given set of inputs. They are very scalable but notoriously incomplete. Semi-formal methods provide a compromise between these two extremes. They often execute the program but provide additional information about execution paths.

A key feature of FoREnSiC is that it does not only implement one but several debugging methods in different back-ends. The different back-ends and methods can be accessed in a unified way. This allows the user to easily switch from one method to another. It also allows to compare or combine methods.

Table 1.1 gives a rough overview over the back-ends currently available in FoREnSiC. The symbolic back-end uses symbolic execution and SMT-solving to identify potential error locations and corrections. The specification has to be given in form of assertions in the code. The simulation-based back-end searches for diagnoses and repairs by repeatedly executing the program on a given set of input vectors. Dynamic slicing is used to improve the accuracy of the diagnoses. Repairs are computed by mutating the faulty components. The specification can be given in form of expected outputs for the input vectors and using assertions. Finally, the WoLFram back-end performs error detection and localisation in hardware designs using a C program as a specification. This is done by checking functional equivalence using SAT/SMT-solving techniques.

The value of FoREnSiC is not only in its back-ends. FoREnSiC is also an environment for implementing new program analysis, verification, and debugging techniques. It provides a lot of infrastructure which can be used. It contains a front-end which is able to transform a C program into a very simple graph-based representation. It also provides data structures to represent logic formulas. Logic solvers such as SMT-solvers are accessible via simple interfaces to solve these formulas.

Table 1.1: Overview and classification of the back-ends.

	Symbolic Back-End	Simulation-Based Back-End	WoLFram Back-End
Debugging subject	C program	C program	RTL-Circuit
Specification	Assertions	Expected outputs and assertions	C program
Method	Semi-formal	Dynamic	Formal
Error localisation	✔	✔	✔
Error correction	✔	✔	✘

1.3 Limitations

FoREnSiC cannot handle concurrent programs. It is not able to fix deadlocks and race conditions. FoREnSiC is a research prototype and it has not been extensively tested. Bug reports as well as other feedback, suggestions for improvements, and personal experiences with the tool are of course always very welcome. Use the address `forensic@lists.iaik.tugraz.at`. FoREnSiC has no graphical user interface yet. Usability will be improved in future releases.

1.4 License

FoREnSiC is an open-source software. It is distributed under the GNU Lesser General Public License (LGPL), Version 2.1, with the copyright held by the University of Bremen, Graz University of Technology, and Tallinn University of Technology. A copy of the license can be found in the distribution and under `http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`.

1.5 Structure of this Manual

This manual is structured as follows. Section 2 **(Installing FoREnSiC)** explains how to install FoREnSiC. Section 3 **(Using FoREnSiC)** contains all information which is needed to use FoREnSiC, but it does not explain how it actually works. More detailed information about FoREnSiC, its architecture, and how it works can be found in Section 4 **(Understanding FoREnSiC)**. A deeper understanding of FoREnSiC may be useful if the obtained debugging results are not satisfying. Often, changing parameters helps, but finding appropriate parameter values often requires some deeper understanding of the implemented methods. Another scenario is that you want to improve FoREnSiC or extend it with a new back-end, for which you need a deeper understanding as well.

Chapter 2

Installing FoREnSiC

FoREnSiC runs under Linux operating systems only. In order to install it, perform the following steps.

1. Make sure that your system has the following tools and libraries installed. Required are
   - a GNU C and C++ compiler[1] (`gcc` and `g++`),
   - the GNU `make`[2] tool,
   - the macro processor `m4`[3],
   - the compression libraries `zlib` and `libbz2` with development support,
   - the tool `cmake`[4],
   - an Objective CAML[5] interpreter and compiler (`ocaml` and `ocamlc`), and
   - the `doxygen`[6] documentation system.

   On Debian-based Linux distributions (such as Ubuntu) the tools can be obtained with the commands

```
sudo apt-get install build-essential
sudo apt-get install m4
sudo apt-get install zlib1g-dev
sudo apt-get install libbz2-dev
sudo apt-get install cmake
sudo apt-get install ocaml
sudo apt-get install doxygen
```

[1] http://gcc.gnu.org/
[2] http://www.gnu.org/s/make/
[3] http://www.gnu.org/s/m4/
[4] http://www.cmake.org/
[5] http://caml.inria.fr/
[6] www.doxygen.org/

or by installing the respective packages using the packet manager. For other Linux distributions, similar packages exist. If not, follow the links to download and install the tools and libraries.

2. Create a directory where all the third-party tools on which FoREnSiC relies should be installed. We will refer to the absolute path to this directory as `<TP_INSTALL>` in the following.

3. Set the environment variable `FORENSICTP` to `<TP_INSTALL>`. In `bash` this works with the command

   ```
   export FORENSICTP=<TP_INSTALL>
   ```

   This environment variable must be set both during the installation process and when running FoREnSiC. To avoid having to set this variable whenever a new shell is opened, the above line can simply be included into the configuration file of the `bash`, which is usually `~/.bashrc`.

4. Add to the environment variable `LD_LIBRARY_PATH` the two directories `<TP_INSTALL>/mpfr-2.4.2/lib` and `<TP_INSTALL>/gmp-4.3.2/lib`. This is necessary for running FoREnSiC. The most convenient way to do so is by adding the line

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$FORENSICTP/
       mpfr-2.4.2/lib/:$FORENSICTP/gmp-4.3.2/lib/
   ```

   to the `~/.bashrc` configuration file.

5. Install all third-party tools and libraries required by FoREnSiC. This includes

   - Boost 1.45.0, see `http://www.boost.org/`,
   - CBMC 3.9, see `http://www.cprover.org/cbmc/`,
   - CREST's extension of CIL, see `http://code.google.com/p/crest/`,
   - CppUnit 1.12.1, see `http://sourceforge.net/projects/cppunit/`,
   - GMP 4.3.2, see `http://gmplib.org/`,
   - MPFR 2.4.2, see `http://www.mpfr.org/`,
   - MPC 0.8.1, see `http://www.multiprecision.org/`,
   - GCC 4.5.0, see `http://gcc.gnu.org/`,
   - Yices 1.0.28, see `http://yices.csl.sri.com/`, and
   - Z3 3.1, see `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

   For your convenience, we have provided an installation script which downloads and installs all these tools and libraries. Before you execute it, check out the licenses of the listed tools. To execute the script, open a `bash` in the directory `thirdparty/` and execute `install-all.sh`. This may take some time (some hours, even). The main reason is that Boost and GCC are compiled on your machine. Go and drink some coffee meanwhile.

6. Check if all third-party tools have been installed successfully. For every tool listed above there should be now a corresponding directory in `<TP_INSTALL>`. If not, search for error messages (i.e., for the keyword "error") in the output of the install script, try to resolve the problems and re-execute the installation script. Instead of re-executing `install-all.sh`, you can execute the install script for the tool which failed only. This saves time.

7. Compile FoREnSiC itself. This is done by opening a shell in the directory `tool/` and typing "`make`". An executable of FoREnSiC will be created in `tool/build/src/forensic-bin`.

8. Test FoREnSiC by typing "`make check`". If all tests pass, then the installation should have been successful.

9. Type "`make doc`" to create the Doxygen documentation of the source code. The documentation is then accessible from `tool/doc/doxygen.html`.

Chapter 3

Using FoREnSiC

This section explains briefly how to use FoREnSiC. It does so without explaining how FoREnSiC works internally. Hence, this Section has to be understood as a quick-start guide. The question how FoREnSiC is used in the best way for a particular purpose can only be answered with more back-ground information, which is given in Section 4.

3.1 Starting FoREnSiC

After building FoREnSiC by typing `make` into a shell opened in the directory `tool`, an executable is created in the file `tool/build/src/forensic-bin`. This executable is the command-line application FoREnSiC. It is configured via command-line options. To get a list and a short description of all options, execute FoREnSiC with the `-h` or `--help` option. That is, type

```
build/src/forensic-bin -h
```

in your shell. Don't be frightened by the high number of options. The most important ones are the `-i` option to specify an input file and the `-b` option to specify a back-end. There are different back-ends implementing different error localisation and correction methods. See Section 3.3 for an overview and Section 4 for more information about the working principle of the different back-ends. Most options only fine-tune the behavior of the back-ends. They are mainly meant for advanced users. Non-experts can go with the default values.

3.2 Annotating Programs

In order to make FoREnSiC understand what the inputs and outputs of the program under analysis are, the program can be annotated with special functions. All these special functions are declared in the header file `forensic_instr.h` in `tool/src/library/`. If this header file is included, all calls to the special functions will be ignored by the compiler. This means that the annotations for

FoREnSiC can be kept in the program without any harm for compiling and executing the program. The functions only have a special meaning inside FoREnSiC.

Not every back-end supports every annotation. Confer to the description of the back-end to see which annotations are supported.

Modeling Input

The function `FORENSIC_input_`*some_type*`(x)` indicates that the variable `x` is an input. There exist different versions of these functions for different types of the variable `x`. Depending on the type of `x`, *some_type* may be substituted by `char`, `short`, `int`, `long`, `long_long`, `unsigned_char`, `unsigned_short`, `unsigned_int`, `unsigned_long`, `unsigned_long_long`, `float`, `double`, or `long_double`. There is also a function `FORENSIC_input_string(char** x, int MAXLEN)` which can be used to indicate that a string (a `char`-array) is an input. The second argument of the function specifies the maximum length or the string. There are no dedicated functions for compound types such as structs or unions. In order to declare a struct or union to be an input, its fields have to be declared as inputs manually. Moreover, there are no functions which can express that a pointer is an input.

Different back-ends may treat variables as inputs also if they are not explicitly marked with `FORENSIC_input_`*some_type*`(x)`. Examples are the parameters of the entry function or uninitialized local variables.

Modeling Output

The counter-part to the functions `FORENSIC_input_`*some_type*`()` are the functions `FORENSIC_output_`*some_type*`()`, using the same syntax. These functions are used to indicate that a certain variable at a certain location in the program is somehow communicated to the environment of the program. They are useful for specifying the desired behavior of the program using expected output values for given input values.

Specifying Desired Behavior

Many back-ends support the `assert`-macro to specify safety properties of the program. In addition to that, FoREnSiC provides a macro `FORENSIC_assume(x)`, where `x` is is some predicate. This function expresses the assumption that the predicate `x` holds at the current location in the program. If the assumption does not hold, the program is allowed to behave arbitrarily. The macro is defined as `if(!(x)) {exit(0);}`.

Example

The following example program illustrates the usage of the FoREnSiC annotations. It computes the sum of all integers from `0` to `max` (including `max`).

```
1 void main() {
2   int max, count, sum = 0;
3   FORENSIC_input_int(max);
4   FORENSIC_assume(max >= 0);
5   for(count = 0; count <= max; count++)
6     sum += count;
7   assert(sum == (max * (max + 1)) / 2);
8   FORENSIC_output_int(sum);
9 }
```

The variable `max` is marked as an input of this program, the variable `sum` is declared to be the output. Line 4 expresses an assumption on the input: its value is assumed to be greater or equal to zero. The program works correctly only if this assumption is fulfilled. In Line 7, an assertion is used to check whether the computation was done correctly using Gauss' formula for the sum of integers.

If a program should model that an input is read repeatedly from some source, then the `FORENSIC_input_`*some_type*`()` function can also be used in a loop. The same holds for outputs.

3.3 Selecting a Back-End

Consider again Table 1.1, which gives an overview and a rough classification of FoREnSiC's back-ends.

The **symbolic back-end** performs automatic error localisation and correction in incorrect C-programs using semi-formal methods. The specification must be given using assertions in the code. Note that this also allows to use reference implementations as a specification: simply write a wrapper which executes the program and the reference implementation on the same inputs and then compare the outcomes using suitable assertions. This has the nice effect that the notion of equivalence can be defined freely by the user. The program is analyzed using symbolic or concolic execution. Symbolic execution does not support arrays, pointers, structs, unions, and floating-point computations at the moment. When using concolic execution, these constructs are allowed in the program. However, they are handled only approximatively at the moment. Arrays are handled like sets of variables. The symbolic back-end does not realize that `arr[i]` follows `arr[i-1]`. Consequently, it cannot locate or repair bugs which result from erroneous array indexing. The situation is similar for structs, unions and pointers. The back-end supports all annotations of the form `FORENSIC_input_`*some_type*`()` except for `FORENSIC_input_string()`. Calls to `FORENSIC_output_`*some_type*`()` are ignored. The user can also encapsulate parts of the program by `<ASSUME_CORRECT>` and `</ASSUME_CORRECT>` tags. The code between these tags is assumed to be correct, it is not diagnosed or repaired. This back-end is still under active development.

The **simulation-based back-end** has the same goal, namely to debug incorrect C programs. As a specification, the user can provide input vectors with corre-

sponding expected outputs. In addition to that, assertions in the code can be used. The back-end implements a simulation-based method. It does not impose serious restrictions on the subset of C that can be handled, except for the restrictions of the front-end (see Section 4.2.1). The back-end supports all annotations of the form `FORENSIC_input_`*some_type*`()` and `FORENSIC_output_`*some_type*`()`.

The **WoLFram back-end** implements an equivalence check between a hardware design and a C program. That is, in contrast to the other back-ends, the C program serves as a specification (for the hardware design) here. The hardware design may be given in Verilog or VHDL at the register transfer level. In the C program this back-end does not support compound types such as structs and unions, pointers, and multidimensional arrays. It supports the `FORENSIC_input_`*some_type*`()` annotations except for `FORENSIC_input_string()`. The annotations `FORENSIC_output_`*some_type*`()` are not supported. The WoLFram back-end uses the tool WoLFram [16] as an underlying reasoning framework. This tool is not included in this open-source release. Contact the FoREnSiC team to obtain a copy of this tool. Since the WoLFram back-end is not fully included in this release, it will also not be described in detail in Section 4.

Chapter 4

Understanding FoREnSiC

This section gives insight into the working principle of FoREnSiC and its back-ends. This additional information about the working principle can be useful for fine-tuning the behavior of a back-end to obtain better results for a particular debugging problem. The information provided in this section is also vital if you want to extend FoREnSiC, e.g., by improving a back-end or adding a new one.

4.1 The Architecture

Figure 4.1 depicts the architecture of FoREnSiC in a simplified form. A (potentially faulty) C program is the main input for the tool. The front-end of FoREnSiC parses this C program and produces an internal model of the program. FoREnSiC has several back-ends operating on the model of the program. They implement different error localisation and correction methods. The user selects the back-end which is best suitable for her problem.

Certain back-ends require additional inputs. This may include test vectors with expected outputs or other forms of specifications, hardware implementations to check equivalence with, or the string representation of the C program. These additional needs are not drawn in the figure for simplicity. FoREnSiC does not

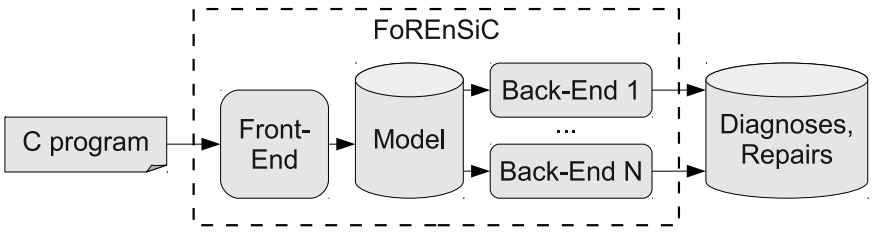

Figure 4.1: The architecture of FoREnSiC.

really impose restrictions on back-ends. It only provides an infrastructure that can be used. The format of the diagnostic information produced by the back-ends is not (yet) unified. Every back-end presents diagnostic information in its own way to the user.

The following sections explain the different parts of the architecture and the different back-ends in more detail.

4.2 The Front-End

The front-end is based on the GCC plug-in API version 4.5.0 and hooks into the GIMPLE pass [8]. The front-end processes the input program as a tree in low level GIMPLE SSA (single static assignment) form. The compiler is invoked without the linking stage, so it is possible to process programs without a main function or with undefined functions. To get the SSA form, one can invoke *gcc* with the option `-fdump-tree-ssa`.

The front-end should work on 32-bit and 64-bit platforms, but it is required that every part of the toolchain (forensic-bin, gcc plugin, gcc) is compiled on the same host. The input program has to be a correct ANSI C (C90) program, including the GNU dialect of ISO C90. If the compiler finds an error, no model is built. Warnings are ignored by the plug-in. Currently only one input file is supported.

4.2.1 Unsupported C Language Elements

Some C features cannot be represented in the FoREnSiC model:

- keyword `volatile` as type modifier
- Bit-fields
- storage duration specifiers (`auto`, `register`, `extern`) are ignored; the keyword `static` is an exception
- empty infinite loops like: `while(1){}`

Moreover, `switch` statements are not supported in the model, but they are transformed into a series of nested `if-else` statements.

4.2.2 Translating into the Internal Model

The translation to the internal model (see 4.3) does not any longer represents a valid ANSI C program. Mainly because the auto-generated identifiers for the SSA temporary variables like `D.1234` are not allowed in C. Some operators and control structures are not used in the SSA form, like the increment (`++`) operator or `for` and `while` loops.

Furthermore, because the input file is completely preprocessed by the compiler at the stage of our plug-in, there is no direct information in the model for compiler directives like `#include`, `#pragma` etc. Also unused elements like variables or structs are omitted.

Arrays are often represented with pointers, indexing is done with pointer arithmetic, but not in all cases. The decision is done by the GCC, further information can be found in the GCC internal documentation.

The model is an exact representation of the GIMPLE SSA tree, except for `switch` statements as described above.

Strings are represented like arrays, e.g.: `&"Message"[0]`.

4.2.3 Special Cases

Because the low level GIMPLE SSA tree is only an intermediate representation for internal use by the GCC, there are cases in which the plug-in is unable to find a correct translation in the model or does not understand a specific GIMPLE tree. In these cases, there is an error message thrown of some type, but due to the multitude of possible GIMPLE statements, trees and nodes, some cases are not covered yet. An example for such a special case is the internal use of ranged array indices:

```
int list[0:D.3136] [value-expr: *list.4];
  int[0:D.3136] * list.4;
```

4.2.4 Code Location Information

Many elements in the model have fields for location information such as filename, line and column number. Available information is filled into the model. However, GCC gives only "points to" information and not range information. Thus, the location of a statement is determined by the first char of an identifier (declaration), the position of the equal sign (assignment), the left parenthesis (method call) or the first char of a method name (method declaration). The last known position is assigned to generated statements and identifiers, because they are "related" to this one.

4.3 The Internal Model

This section explains the internal data structures which are used to represent a C program. The front-end transforms the program into this internal representation. The back-ends then use these data structures for automated debugging. The information given about the model in this section is rather detailed. The reason is that, if you want to work with the model, improve a back-end, or even create a new back-end, detailed information about the model will be needed.

4.3.1 Structure of the Model

Figure 4.2 illustrates the structure of the internal model in form of a UML class diagram. It focuses on which data is stored where. For information about the methods to traverse the data structure, read out the information, or to manipulate the data structure, have a look at the Doxygen documentation in `tool/doc/doxygen.html`.

ProgramData and FunctionData

The result produced by the front-end is an object of type `ProgramData`. Via this object, all information about the program can be accessed. A `ProgramData` object stores the global variables of the program and their type. Furthermore, it stores information about the functions of the program in `FunctionData` objects. The information about a function of the program includes, in essence, the parameters and their type, the local variables and their type, and the return type of the function. The actual body of the function, i.e., the executable code of the function, is represented as a flowchart.

Flowcharts

Every node of the flowchart is represented by a `FlowChartNode` object. An object of type `FunctionData` knows all `FlowChartNode`s which belong to it. It also has a reference to a special `FlowChartNode` which represents the root node of the flowchart. The `FlowChartNode` itself does not directly contain information about the statement it represents. This information is outsourced into a `Statement` object. Every `FlowChartNode` has exactly one `Statement` object. The `FlowChartNode` itself only contains information about the interlinking with other `FlowChartNode`s. That is, it knows its predecessors and successors. Every `FlowChartNode` except for the root node of the flowchart has at least one predecessor.

The model distinguishes three types of `FlowChartNode`s. `OperationNode`s represent operational statements of the code. For instance, the statement `x = a + 4;` would be represented as an operation node. Other examples are function calls or `return` statements. Most `OperationNode`s have exactly one successor. However, if the `OperationNode` represents the last node of a function (such as a `return` statement) it may also have no successor.

`ConditionNode`s model conditions which evaluate to `true` or `false`. They represent branching points in the program. They always have exactly two successors. One is taken when the condition evaluates to `true`, the other one when it evaluates to `false`. For instance, expressions like `a > b + 1` occurring as conditions in `if`-statements of loops are modeled using `ConditionNode`s.

Finally, `PhiNode`s are special nodes that do not directly correspond to a code fragment of the C program. They store information saying how to unify variable names on merging paths when the program is represented in static single assignment (SSA) form. `PhiNode`s have only one successor.

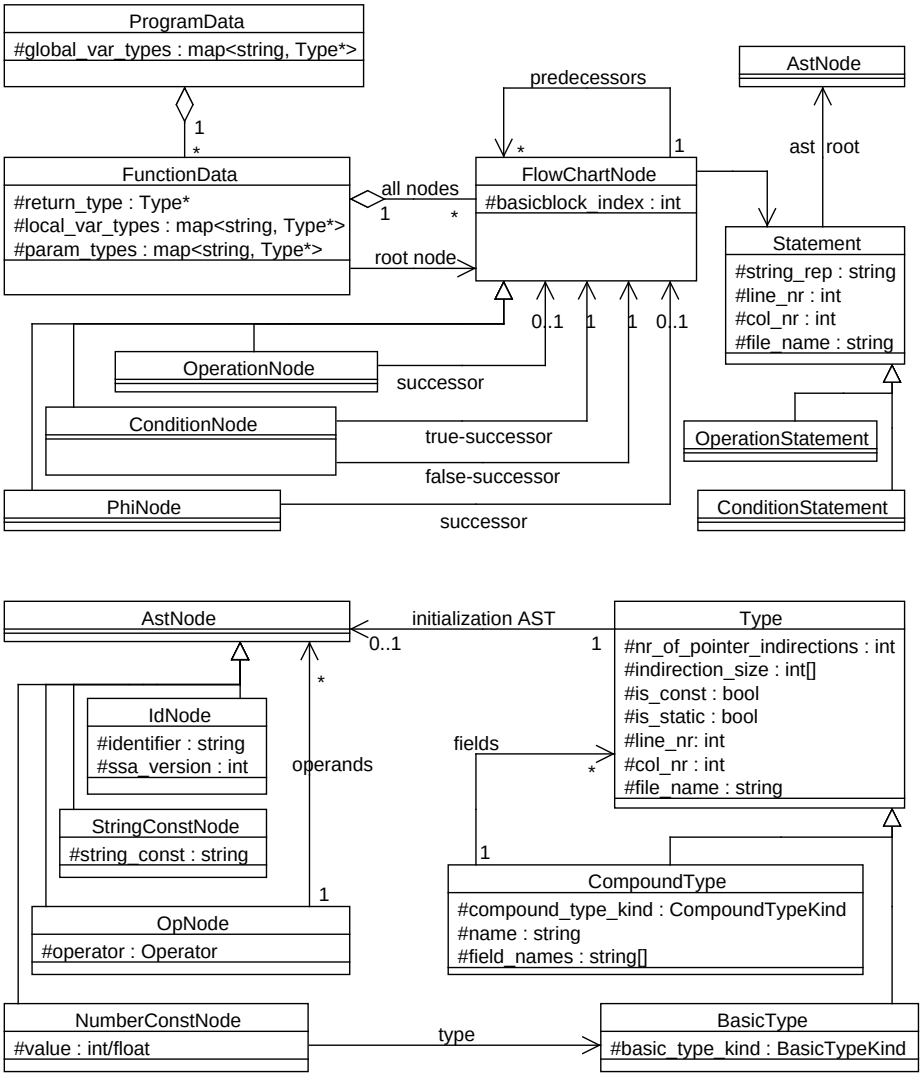

Figure 4.2: A UML class diagram illustrating the structure of the model.

Statements and their Abstract Syntax Tree

While `ConditionNode`s maintain the control flow, the data flow of a program is modeled with `Statement`s and their abstract syntax tree (AST). `Statement`s store their string representation as well as their location in the C program. The AST of a statement is a tree-like decomposition of the statement into its operators and operands. The AST is represented by a tree of `AstNode` objects and the `Statement` holds a reference to the root of the tree.

There are four kinds of `AstNode`s. `IdNode`s represent identifiers in the statement. Most importantly, these are variable names and function names. Identifiers are represented as string. A unique integer `ssa_version` is used to create unique variable names when using the SSA form. `IdNode`s are always leaves of the AST.

`StringConstNode`s represent string constants. For instance, the string in the statement `printf("abd");` would be represented as a `StringConstNode`. These objects are leaves in the AST as well.

An `OpNode` represents an operation applied to other `AstNode`s. Such nodes form the inner nodes of the AST. `OpNode`s store the kind of operation applied. Possible operations are arithmetical operations (`+`, `-`, ...), logical operations (`&&`, `||`), bitwise operations (`&`, ...), comparison operations (`>`, ...), but also casts, pointer dereferencing, array index operations, and function calls. A function call is handled as an `OpNode` where the first child is an `IdNode` containing the name of the function. The remaining children represent the parameters of the function.

Finally, there is the `NumberConstNode` which represents a constant number in the program. Besides the value it also stores the type of the constant (e.g., `int`, `unsigned long`, etc.; recall that you can write `16` or `16UL` in C). `NumberConstNode` are leaves in the AST as well.

Data Types

Data types are represented as instances of class `Type`. We distinguish between `BasicType`s and `CompoundType`s. A `BasicType` represents a primitive type such as `void`, `char`, `unsigned char`, `short`, ..., `unsigned long long`, `float`, `double`, or `long double`. The `BasicTypeKind` is basically an enumeration of constants for all these primitive types.

`CompoundType`s are types that consist of other types, i.e., structs or unions. The `CompoundTypeKind` can take on exactly these two values. A `CompoundType` has a name and a list of fields and their names. Each field in the `CompoundType` is a `Type`, so either a `BasicType` or again a `CompoundType`.

Pointers and arrays are modeled in the following way. Since every type can occur as a pointer (or a pointer to a pointer, etc.), we store the number of pointer indirections to every type (in the field `nr_of_pointer_indirections` of `Type`). If it is zero, this means that we have a normal type. If it is one, we have a pointer of this type. If it is two, we have a pointer to a pointer of this type, and so on. If we have a multidimensional array where the size is statically

known, this size is stored for each dimension in `indirection_size`. This reflects the fact that arrays and pointers are not fundamentally different — an array is just a pointer to its start and a size. Besides this information for pointers and arrays, the `Type` stores the location in the C-file where the type is declared, flags indicating whether the type is static or constant, and an AST representing initializing code for the type.

4.3.2 Example

This section illustrates how the model is structured using a simple example. We use our front-end to convert the following function (`examples/max_only.c`) into a model.

```
1 int max(int x, int y) {
2   int res = x;
3   if(y > x)
4     res = x;
5   assert(res >= x && res >= y );
6   return res;
7 }
```

Using the command-line option `--dotmv`, a picture of the model can be exported in DOT format. Figure 4.3 contains such a picture for the above function. It has been produced by typing[1]

```
build/src/forensic-bin -f gcc -i ../examples/max_only.c
    --dotmv=4
dot -Tpdf ./model.dot -o ./model.pdf
```

in a shell opened in the directory `tool/`.

Let us discuss Figure 4.3 now in more detail. Normally, the global variables would be listed in the box titled "Globals", but our program does not have any global variables. The blue box contains all information to the function `max`. The parameters and local variables of the function are listed in the respective boxes, together with their type. Note that the local variables `D.2728` and `iftmp` have been introduced by the front-end to simplify the code. The root of the flowchart is marked with "Start", the terminal node is marked with "End". In general, there can by many terminal nodes in one flowchart.

Black boxes in the flowchart represent `OperationNode`s, green boxes depict `ConditionNode`s, and brown boxes contain `PhiNode`s. Arrows between the boxes represent successor relations, i.e., they express the control flow. The golden boxes contain the `AstNode`s. The structure of the `AstNode` and their content should be rather self explaining.

This example should have given a first idea about how the model looks like. In order to get a deeper understanding, print the model for more complex programs and study it. Recall that the `--dotmv` argument allows you to define the level

[1] `dot` is a program to visualize graphs. On Debian-based systems you can install it with the command `sudo apt-get install graphviz`. Otherwise, download it from `http://www.graphviz.org/`.

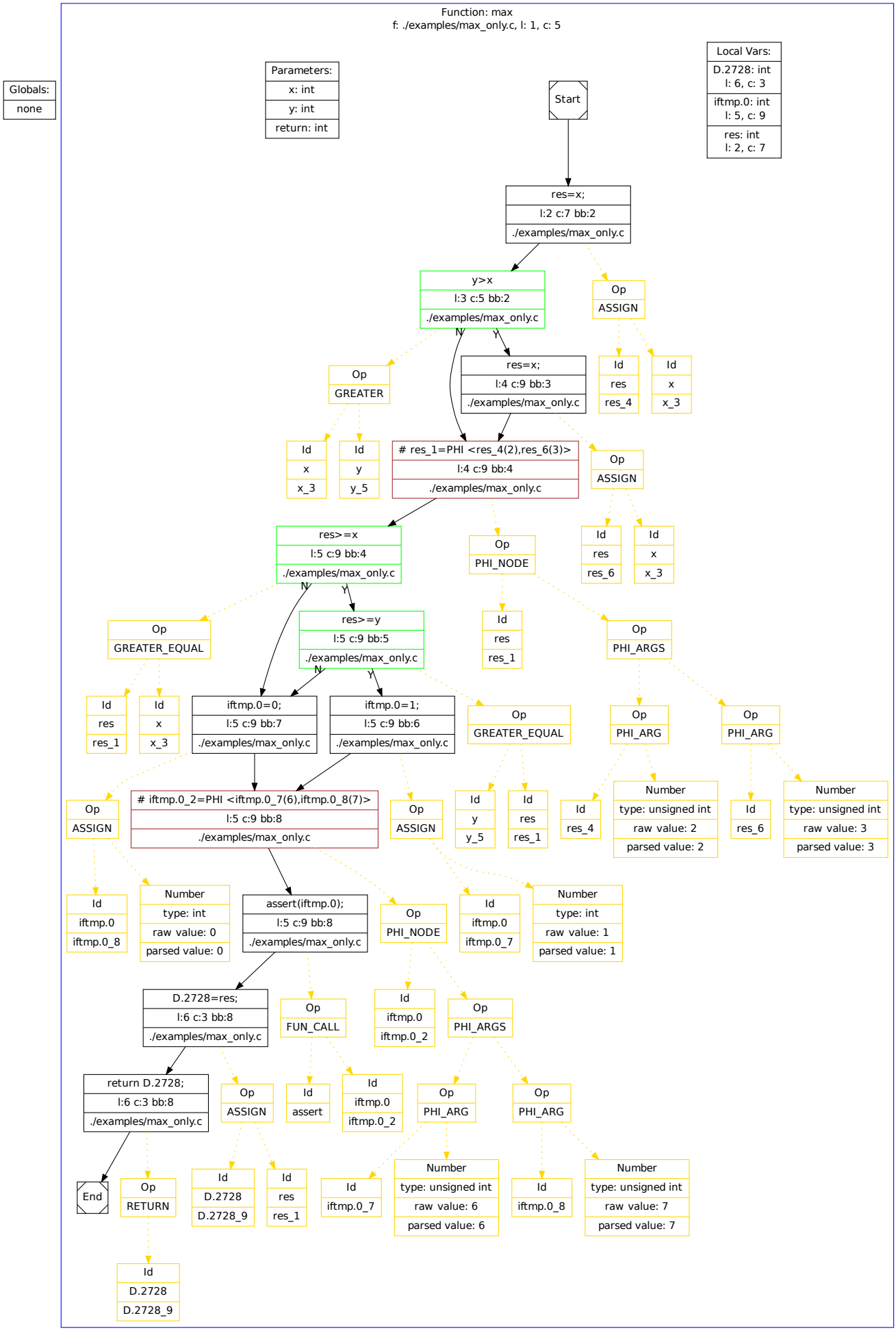

Figure 4.3: An illustration of the model for a small example program.

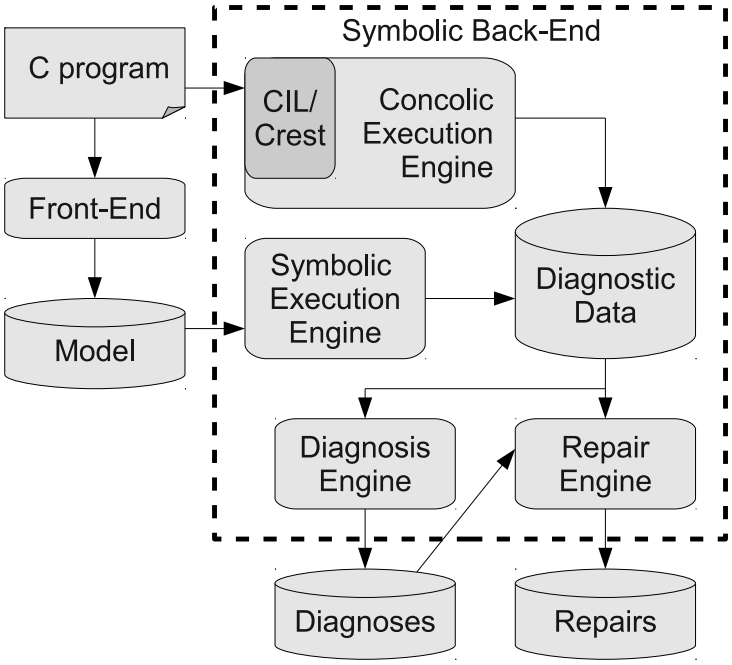

Figure 4.4: The architecture of the symbolic back-end.

of details that should be included in the illustration of the model.

4.4 The Symbolic Back-End

This back-end performs error localisation and error correction in incorrect C programs. The specification must be given in form of assertions in the code. The symbolic back-end is categorized as a semi-formal method, because the symbolic execution step on which it relies is semi-formal. It is able to detect, locate, and correct errors in the program. As a fault model, the back-end assumes that the error is on the right-hand side of an assignment or in a condition. The front-end often assign expressions (e.g., arguments of function calls) to temporary variables, so the back-end is able to locate and correct bugs in many expressions. However, it cannot handle, e.g., missing statements, missing function calls, etc. This back-end is neither complete nor sound. That is, it does not guarantee that a diagnosis or repair is found, even if one exists. It also does not guarantee that all the found diagnoses and repairs are correct.

The symbolic back-end is outlined in Figure 4.4. Basically, this back-end works in three steps. The first step is a program analysis step which results in diagnostic information about the program under consideration. This program analysis step can be done in two different ways: using symbolic execution or using concolic execution. Both ways lead to the same information about the

program. In essence, the resulting *Diagnostic Data* contains formulae in some logic that state under which circumstances the program conforms to the specification. The diagnosis engine attempts to identify sets of components of the program which can be modified in such a way that the program becomes correct. Finally, the repair engine attempts to synthesize new implementations for the incorrect components. Both the diagnosis engine and the repair engine use SMT-solving to accomplish their task. In the following, the debugging method is explained in more detail. Even more background information can be found in [11].

4.4.1 The Symbolic Execution Engine

Symbolic execution [5, 10] is a technique which allows to reason about the correctness of a program automatically. This forms a basic building block for answering questions like

- Can a certain component of the program be replaced in such a way that the program is correct afterwards? (Diagnosis)
- How would such a replacement have to look like? (Repair)

In order to reason about the correctness of a program one has to analyze its behavior. The most straightforward way to do so is to execute it. The problem with this approach is that the program behavior depends on inputs, and typically one cannot execute a program for all possible inputs. Symbolic execution provides a solution for this dilemma. Instead of executing the program with concrete values for the inputs, it is executed using symbols as inputs. Symbols are like variables: they can represent any possible value. Symbolic execution keeps track of the symbolic values of all variables, where a symbolic value can be an arbitrary expression over symbols and constants. Whenever a branching point in the program is encountered, the execution forks. For every branch, a condition, formulated over the input symbols and stating when the branch is taken, is computed. Along an execution path, the branch conditions are accumulated to a path condition. A path condition states when a certain path through the program is taken. The assertions in the code (our specification) allow to classify execution paths into correct and incorrect executions: an execution ending in an assertion violation is then an incorrect one. Reasoning about the correctness of the program reduces then to the reasoning about the feasibility of the path conditions associated with incorrect execution paths. This can in turn be automated using SAT/SMT solving techniques.

Example

Figure 4.5 illustrates symbolic execution on an example. Two symbols X and Y are used for the unknown values of `x` and `y`. Boxes contain execution states, dashed lines link them to the program, and arrows indicate the execution flow. In Line 3, the execution forks since both branches are feasible. The condition which has to be fulfilled for the program to reach a certain state is denoted as

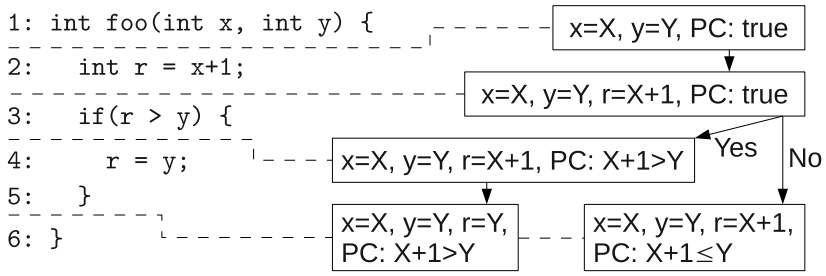

Figure 4.5: Example: Symbolic execution.

PC. The path conditions can be read from the PC-fields in the leaves of the tree. This program has two paths with conditions $X + 1 > Y$ and $X + 1 \leq Y$.

Symbolic Execution for Error Localisation and Correction

Standard symbolic execution performed on the program under analysis only allows to check whether the program is correct, i.e., whether there exist input values for which an assertion is violated. In order to allow for error localisation and correction, the program is transformed in the following way before it is executed symbolically. Each assignment `LHS = RHS` is replaced by

```
1 old_value = LHS;
2 LHS = RHS;
3 FORENSIC_repair(c, LHS, v1, v2, ..., vn);
```

where `FORENSIC_repair` is a special function. Its first argument is a unique identifier of the component (the assignment statement) in the program. The second argument contains the variable to which the RHS is assigned. The remaining arguments `v1...vn` are the names of the variables which are in scope when the statement is executed.

Each condition `if(A OP B)`, where $OP \in \{$`>`, `<`, `>=`, `<=`, `==`, `!=`$\}$, is instrumented in a similar way:

```
1 op1 = A;
2 op2 = B;
3 FORENSIC_repair_cond(c, cond, op1, op2, OP, v1, ..., vn);
4 if(cond != 0)
```

This is not only done for conditions in an `if`-statement but also for conditions of loops. More complex conditions are decomposed by the front-end. All this instrumentation is done directly on the model.

The functions `FORENSIC_repair` and `FORENSIC_repair_cond` express that the corresponding components of the program may be faulty. The components can behave arbitrarily in the transformed program. The functions are handled in a special way during symbolic execution. When executing `FORENSIC_repair`, a new symbol r is created. This new symbol is not an input symbol but what will

be called a *repair symbol* in the following. The repair symbol is associated with some additional information. This includes the symbolic value of the second argument of the function. This is the value the symbol gets if the component is assumed to be correct and, hence, remains unchanged. This value will be denoted Orig(r) in the following. The component `c` that produced symbol r is denoted as CmpOf(r). Furthermore, the symbolic values of all variables `v1`...`vn` are stored. These values are denoted as Vals(r) in the following. They are needed to synthesize new expressions involving the variables in the error correction step. The newly created symbol r forms the new value of `LHS` when `FORENSIC_repair` is finished. `FORENSIC_repair_cond` is handled similarly. The main difference is that the new symbol which is created is Boolean and that its value if it is assumed to be correct is Orig(r) = `op1 OP op2`. The newly created symbol r is assigned to the variable `cond`.

Configuration of the Symbolic Execution Engine

The program analysis using symbolic execution is activated with the option `--syb_conc=False`. Otherwise, program analysis is done using concolic execution (see Section 4.4.2). Whether or not conditions should be treated as potentially faulty components can be configured with the option `--syb_conc`.

The semantics of the program has to be mapped to formulas. The expressiveness of these formulas can be configured with the `--syb_th` option. `--syb_th=lin` means that only linear expressions are allowed in the formula. Linear expressions are of the form $k + \sum k_i \cdot X_i$, where all k_i are integer constants and all X_i are variables. The comparison operators {`>`, `<`, `>=`, `<=`, `==`, `!=`} are allowed to relate expressions, Boolean connectivities are allowed to combine predicates. In SMT terminology this is called (unquantified) Linear Integer Arithmetic and is abbreviated as `QF_LIA` [1]. Consequently, when using `--syb_th=lin`, the semantics of bitwise operations, divisions, etc. cannot be captured exactly. As an approximation in such a case, the symbolic execution engine creates a new input symbol to represent the result of an operation which is not supported. With `--syb_th=bit` bitvector arithmetic (`QF_BV` [1]) is used to express the semantics of the program. This logic allows to express the semantics of all operations on integer variables that are allowed in C. The `--syb_th` option does not only specify the logic used in the symbolic execution engine but also in the diagnosis engine and in the repair engine.

The option `--syb_sv` specifies the solver to be used for SMT solving in the symbolic back-end. In principle, the solvers are functionally equivalent, they only differ in performance. One exception is the Yices solver (`--syb_sv=yices_api`) which does not support the full bitvector arithmetic as defined in [1]. Thus, if `--syb_sv=yices_api` is used in combination with `--syb_th=bit`, approximations are done in some cases (divisions, modulo operations, and bit-shifts by a variable number of positions).

The symbolic execution engine is able to output a symbolic execution tree (a tree as shown in Figure 4.5 but with some more details) in DOT format. This tree can be used to understand in detail how the program was analyzed using symbolic execution. However, since these trees can become quite large, a visualization

using the `dot` program only makes sense for small programs. The `--syb_se_df` option specifies the name of the file into which the tree is written.

The symbolic back-end is designed to perform only incomplete program analysis. This allows for higher scalability, but the computed diagnosis and repairs are not guaranteed to be correct or fully accurate. The thoroughness of the program analysis done by the symbolic execution engine can be configured with the options `--syb_se_ml` and `--syb_se_mf`. With `--syb_se_ml` a maximum number of iterations through any loop of the program can be configured. Putting a bound on this number is necessary because without such a bound symbolic execution may take too long. Often the number of loop iterations is not fixed but depends on the inputs, and any number of loop iterations is possible with some input. If the maximum number of loop iterations is set to, e.g., four, then symbolic execution does not produce any information about the behavior of the program for inputs which cause more than four iterations. Consequently, the diagnoses and repairs found subsequently may not be fully accurate for such inputs. However, there is the hope that if the program works for at most four loop iterations, it would also work for more. The `--syb_se_mf` option serves a similar purpose. It limits the number of functions on the call stack, i.e., the total call depth. This is especially important for programs which contain recursive function calls.

4.4.2 The Concolic Execution Engine

Concolic execution [9, 14] is a variant of symbolic execution. The main difference to symbolic execution is that the program is executed on both concrete and symbolic inputs simultaneously. The path through the program is determined by the concrete inputs. A path condition is computed for this execution path. In order to achieve this, the program is instrumented in such a way that every operation is not only executed normally but also symbolically. Hence the name "concolic" execution, which is an artificial word composed of "**conc**rete" and "symb**olic**". The path condition is not computed as one monolithic condition over the input symbols. It is computed as a conjunction of conditions, where every conjunct corresponds to the condition of a branching point along the execution path. This allows the creation of inputs that will cause the execution to follow a different path through the program in the next iteration: If one of the conjuncts is negated, conjuncts corresponding to later branches are discarded, and the resulting condition is solved for concrete inputs using an SMT solver. This gives inputs that trigger the same execution path until the branching point where the corresponding condition has been negated. At this point, the other branch will be taken. Different search strategies for different paths through the program can be implemented [2]. The resulting path conditions enable automated reasoning about program correctness, just as for symbolic execution.

Example

Consider the program in Figure 4.5. It may be executed concolically with the inputs `x=0` and `y=0` in the first iteration. These inputs cause the execution to take the path via Line 4. Along this path, the path condition $X + 1 >$

Y is computed. Here, X and Y denote the values of the variables `x` and `y`, respectively. One conjunct of the path condition now is negated. There is only one conjunct in this example, so $X + 1 \leq Y$ is obtained. This formula is solved for a satisfying assignment to get inputs which trigger a different execution path. A possible solution is $X = 0$ and $Y = 1$. The program is executed using these inputs in the next iteration. The concolic execution takes the other path and computes the path condition $X + 1 \leq Y$.

Concolic Execution for Error Localisation and Correction

Similar to symbolic execution, the program is transformed before it is executed concolically. This is done to allow for error localisation and correction. This transformation works as described in Section 4.4.1. The difference is that the transformation is done textually on the input program and not on the model. In detail, the following steps are performed.

1. The program is simplified using CIL [12]. The simplified file is written to `instr_tmp/input.cil.c`. If FoREnSiC reports diagnoses and repairs which are difficult to link to the original program, consult this file to see how it was simplified.
2. The simplified program is transformed as explained in Section 4.4.1. The result is written to the file `instr_tmp/input.rep.c`. Have a look at this file if you want to find out which statements of the program are considered as potentially faulty components.
3. The transformed program is instrumented for concolic execution. That is, function calls are inserted which allow to keep track of the symbolic values of all program variables during the execution. This instrumentation is done using CREST's extension of CIL [2]. The instrumented file can be found in `instr_tmp/input.rep.cil.c`.
4. The instrumented file is compiled into an executable. It can be found in `instr_tmp/input.rep`.

The created executable is now executed repeatedly using different inputs. Every execution yields a path condition for the program path that has been executed. The path condition is used to compute inputs that activate a different path in the next iteration.

Configuration of the Concolic Execution Engine

Concolic execution is activated with the option `--syb_conc=True`. Since the concolic execution engine is an extension of CREST it also inherits the different search strategies for execution paths [2]. The search strategy can be set with the `--syb_ce_se` option. Some strategies can be configured with a search depth using the `--syb_ce_de` option. The option `--syb_ce_it` defines the maximum number of concolic execution runs. That is, it defines the number of paths through the program that should be analyzed at maximum. The higher this

number, the more precise will be the obtained information about the program behavior. Of course, a higher number will also require more computational resources. The `--syb_ce_ab` argument specifies the maximum length of a path to analyze. The length is defined via the maximum number of elements that are pushed to the operand stack during concolic execution. It corresponds to the number of executed statements only very roughly. Just like the `--syb_se_ml` and the `--syb_se_mf` argument for symbolic execution, this parameter limits the depths in which the program is analyzed and avoids that concolic execution hangs in an endless loop or infinite recursion. The parameters `--syb_cond`, `--syb_sv`, `--syb_th` affect the concolic execution engine as already explained in Section 4.4.1.

4.4.3 The Diagnostic Data

The diagnostic data consists of

- the set CMP of all components c identified in the program,
- a function Vars mapping each component c to the names of the variables in scope when c is executed,
- the input symbols $\bar{i}$ representing unknown input values,
- the repair symbols $\bar{r}$ representing unknown behavior of the components of the program,
- the function CmpOf mapping each repair symbol to the component which produced it,
- the function Orig mapping each repair symbols to the value that is returned by the original version of the component in the same situation,
- the function Vals mapping each repair symbol r to the symbolic values of all variables in scope when component CmpOf(r) was executed to produce the symbol r,
- the disjunction $\pi_F(\bar{i}, \bar{r})$ of all path conditions corresponding to paths which ended in an assertion violation, and
- the disjunction $\pi_P(\bar{i}, \bar{r})$ of all path conditions corresponding to paths which ended in a normal termination of the program.

All this information is collected during symbolic (or concolic) execution, as explained in Section 4.4.1. The brackets in $\pi_F(\bar{i}, \bar{r})$ and $\pi_P(\bar{i}, \bar{r})$ indicate that these predicates are formulated over the input symbols $\bar{i}$ and the repair symbols $\bar{r}$. Note that $\pi_P(\bar{i}, \bar{r})$ is not necessarily equal to $\neg\pi_F(\bar{i}, \bar{r})$. One possible reason for a difference is that not all paths through the program may have been analyzed using symbolic (or concolic) execution. In the following, $\pi_P(\bar{i}, \bar{r})$ and $\pi_F(\bar{i}, \bar{r})$ will be used to establish different notions of correctness.

4.4.4 The Diagnosis Engine

We say that a diagnosis is a set of components that can be modified in such a way that the program becomes correct. This definition makes sense because components that can be modified in such a way that the program becomes correct may be responsible for the incorrectness. The diagnosis engine computes minimal diagnoses, i.e., diagnoses for which no subset of components is a proper diagnosis.

A given set Δ of components is a diagnosis iff the formula

$$\forall \bar{i} \,.\, \exists \bar{r} \,.\, \pi_P(\bar{i}, \bar{r}) \wedge \bigwedge_{\{r \in \bar{r} \mid \mathrm{CmpOf}(r) \notin \Delta\}} r = \mathrm{Orig}(r) \tag{4.1}$$

is satisfied. That is, for all inputs $\bar{i}$, there must exist some values that can be returned by the components (some values for the symbols $\bar{r}$) such that the program behaves conforming to the specification. Components which are not part of the diagnosis must return the value that is returned by the original implementation of the component (stored as $\mathrm{Orig}(r)$). If the above formula is satisfied, then this means that there exist some values that can be returned by the components in Δ such that the program becomes correct. This means that the components in Δ can, in principle, be modified in such a way that the program becomes correct.

Computation of diagnoses

Equation 4.1 contains a quantifier alternation which makes it computationally hard to solve. Therefore, in the implementation, Equation 4.1 is not checked for all inputs but only for a given set of concrete input values. Only inputs for which the original program fails are used because inputs for which the original program conforms to the specification make Equation 4.1 vacuously satisfiable. When using concolic execution, the concrete input values that are used to activate certain paths of the program are used. When using symbolic execution, satisfying assignments of the path conditions are computed to obtain concrete values for the inputs. When using a finite set of concrete input values, the universal quantification turns into a finite conjunction. The resulting formula can be solved with an SMT-solver directly. Using concrete values for the inputs can lead to false positives in diagnosis computation, but it increases the efficiency.

Equation 4.1 can be used to compute diagnoses in a naive way: every set of components can be check if it is a diagnosis. FoREnSiC uses a more advanced algorithm to compute minimal diagnoses more quickly. It is based on the theory of model-based diagnosis [13, 6]. It computes minimal conflicts, which are minimal sets of components from which at least one element has to be modified to obtain a correct program. In principle, such sets can be computed by repeatedly solving Equation 4.1. FoREnSiC uses a faster method which exploits the fact that such a minimal conflict corresponds to a minimal unsatisfiable core of the formula. The SMT-solver is used to compute an unsatisfiable core directly. The details are described in [11] and in the Doxygen documentation of the class `DiagnosisConstraintSystem`. Finally, all minimal hitting sets for the

collection of all minimal conflict sets are computed. This is done by building a hitting set tree as explained in [13]. The hitting set tree is exported in DOT format in the file `instr_tmp/hs_tree.dot` and can be visualized using `dot`.

Configuration of the Diagnosis Engine

The diagnosis engine can be operated in two different modes, a conservative mode (`--syb_dia_c=True`) and a progressive mode (`--syb_dia_c=False`). The two modes differ in how they treat the incompleteness in program analysis. In the conservative mode, the program is only considered correct if a termination of the program can be enforced without an assertion violation. This is achieved using Equation 4.1 directly. In the progressive mode, the program is deemed correct if all known assertion violations can be avoided. This is achieved by replacing $\pi_P(\bar{i}, \bar{r})$ by $\neg\pi_F(\bar{i}, \bar{r})$ in Equation 4.1. Note that this has the effect that also endless loops are considered as correct behavior. The conservative method may miss diagnoses, the progressive mode may find too many diagnoses. Both have their merits.

The parameter `--syb_dia_md` specifies the maximum number of diagnosis to compute and the parameter `--syb_dia_ms` specifies the maximum size (in terms of the number of faulty components) of a diagnosis to compute. Diagnoses are computed in order or increasing cardinality. Usually, diagnoses with lower cardinality are considered to be more likely and helpful. This means that, when aborting the computation before all diagnoses have been computed, only less likely diagnoses are missed. The parameter `--syb_dia_ni` defines the number of concrete input values to use for error localisation. The higher this number, the higher will be the accuracy of the diagnoses.

4.4.5 The Repair Engine

For every diagnosis, the repair engine attempts to synthesize new expressions for the faulty components such that the program becomes correct.

Templates for Expressions

The search for expressions is reduced to the search for constants by creating templates for unknown expressions. For instance, the template `k0 + k1*var1 + k2*var2`, where `k0,k1,k2` are unknown integer constants, can represent any linear expression over the program variables `var1` and `var2`. The unknown integer constants will be called template parameter in the following. If the theory of linear integer arithmetic is used for SMT-solving (`--syb_th=lin`), the engine searches for expressions using such linear templates only. The reason is that the semantics of more complex expressions cannot be captured in linear integer arithmetic. If bitvector arithmetic is used, the following sequence of templates is tried:

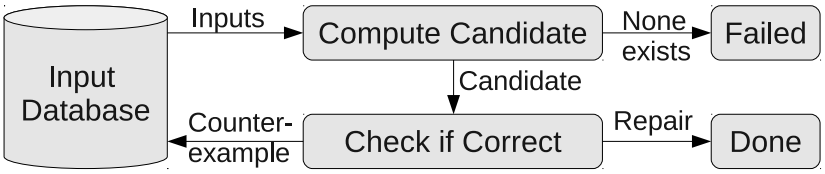

Figure 4.6: Counterexample-guided repair refinement.

Name	Structure for two variables
linear	`k0 + k1*var1 + k2*var2`
dnf	`k0 | (k1 & var1) | (k2 & var2)`
cnf	`k0 & (k1 | var1) & (k2 | var2)`
shift-dnf	`k0 | (k1 & ((var1<<k2)>>k3)) | (k4 & ((var2<<k5)>>k6))`
shift-cnf	`k0 & (k1 | ((var1<<k2)>>k3)) & (k4 | ((var2<<k5)>>k6))`

In the current version of FoREnSiC, adding new templates to this list requires programming. An extension which allows users to define new templates using a simple syntax is planned. At the moment, templates for conditions are always of the form `EXPR_TEMPL OP 0`, where `EXPR_TEMPL` is a template for a non-Boolean expression and $OP \in \{$`>`, `<`, `>=`, `<=`, `==`, `!=`$\}$.

Computation of Repairs

Let $\Delta \subseteq$ CMP be the diagnosis which is to be repaired. The starting point for computing repairs is the condition $\pi_P(\bar{i}, \bar{r})$. All repair symbols r that have been produced by a component $c \notin \Delta$ are replaced by the expression Orig(r). All repair symbols r that have been produced by components $c \in \Delta$ are replaced by a repair template. The variables of the template are Vars(CmpOf(r)). These variables are in turn replaced by their symbolic value, which is stored in Vals(r). The resulting condition contains only input symbols and template parameters. It will be denoted Correct($\bar{i}, \bar{k}$) in the following because it evaluates to true if the program conforms to the specification (when executed on certain inputs $\bar{i}$ and the faulty components have been replaced as defined via the template parameters $\bar{k}$).

The goal is now to compute template parameter values such that for all inputs the program conforms to its specification. That is, the formula

$$\exists \bar{k} : \forall \bar{i} : \text{Correct}(\bar{i}, \bar{k})$$

needs to be solved. Because this formula contains a quantifier alternation, many solvers cannot handle it directly. Therefore, FoREnSiC uses the idea of [15] and [3] to compute repairs by iterative refinements which are guided by counterexamples. This is illustrated in Figure 4.6.

There is a database I of input vectors which is initially empty. In every iteration, a repair candidate is computed in form of template parameter values such that the program becomes correct for the inputs in the database I. This is done by

computing a satisfying assignment $\overline{v_k}$ for the symbols $\overline{k}$ in condition

$$\bigwedge_{\bar{i} \in I} \text{Correct}(\bar{i}, \overline{k}). \tag{4.2}$$

If Equation 4.2 is unsatisfiable, the program cannot be repaired with the given templates and the procedure aborts. Otherwise, it is checked if $\overline{v_k}$ repairs the program for *all* inputs, i.e., if

$$\neg \, \text{Correct}(\bar{i}, \overline{v_k}) \tag{4.3}$$

is unsatisfiable. In this condition, the template parameters $\overline{k}$ have been replaced by their respective values $\overline{v_k}$. If Equation 4.3 is unsatisfiable then a correct repair has been found. The template parameter values $\overline{v_k}$ are mapped back to the expressions they represent, and the repair is presented to the user. Otherwise, a satisfying assignment for the input symbols is computed. These concrete input values are a counterexample for the correctness of the repair candidate. The counterexample is added to I and another iteration is started, which produces a better candidate. This is repeated. If further repairs should be computed, additional constraints are added to Equation 4.2 requiring that $\overline{k}$ is different to all previously computed repairs.

Configuration of the Repair Engine

Just like the diagnosis engine, the repair engine can also be operated in two modes. The conservative mode (`--syb_rep_c=True`) attempts to find a repair such that a successful termination (one that does not involve an assertion violation) of the program is enforced for all inputs. The progressive mode (`--syb_rep_c=False`) attempts to find a repair such that known assertion violations are avoided for any input. For the former, the correctness is formulated using $\pi_P(\bar{i}, \bar{r})$ as explained above. For the latter, the correctness is defined using $\neg\pi_F(\bar{i}, \bar{r})$ instead. The former treats program behavior that has not been analyzed as faulty, the latter as correct. The former may not find a repair even if one exists, the latter may find repairs which are actually incorrect.

When `--syb_rep_hq` is set to `True` a separate program analysis step (using symbolic or concolic execution) is triggered for every diagnosis before the repair process starts. The rationale behind this idea is as follows. In the transformed program (see Section 4.4.1), the number of feasible execution paths is typically way higher than in the original program. The reason is that the transformation allows all components to behave arbitrarily. For error correction, it suffices if only the components that are going to be repaired are left open. All other components can be set to their original implementation. Analysis of this program leads to higher coverage of the relevant behavior and thus, typically, to repairs of higher quality. The disadvantage is the additional time needed for the extra program analysis steps.

With `--syb_rep_maxsat=True`, FoREnSiC uses a performance optimization during the computation of repairs. It tries to find simple repair candidates and nasty counterexamples. A repair candidate is simple if many template parameters are zero or have a special value which makes terms in the template disappear. The search for simple candidates is encoded as a Maximum Satisfiability

(MAX-SAT) problem, as explained in [11]. Nasty counterexamples are ones that contain large, uncorrelated values for the inputs. The computation of such counterexamples is encoded as a MAX-SAT problem as well. Experiments indicate that this optimization leads to significantly faster convergence of the repair refinement loop.

The parameter `--syb_rep_ms` defines the maximum cardinality of a diagnosis to repair. Using this parameter one can, for instance, compute all diagnoses but repair only single-fault diagnoses. The parameter `--syb_rep_mrpd` defines the maximum number of repairs to compute per diagnosis. `--syb_rep_mr` allows to set a limit on the number of repair refinements. The option `--syb_rep_chk=True` makes FoREnSiC check if the computed repairs really make the program correct using the software model-checker CBMC [4]. Note that this option is still experimental.

4.4.6 Implementation

This section gives an overview on the implementation of the symbolic back-end. The classes can be found in `tool/src/back_end/symbolic/`. The entry point for the symbolic back-end is the class `SymbolicBackend`. It triggers program analysis, diagnosis and repair with the right parameters. Detailed information about all classes, their working principle and their interfaces can be found in the Doxygen documentation (in `tool/doc/doxygen.html`).

The transformation of the model for symbolic execution is implemented in the class `ModelInstrumenter`. The transformation for concolic execution is done in the `CILInstrumenter`. Symbolic execution itself is performed by the `SymbolicInterpreter` and its sub-classes. Concolic execution is done using the `ConcolicInterpreter`. The search for different execution paths in concolic execution is implemented in the class `CrestConcolicSearch`. Additional information (a type, the original value, values of variables in scope) to symbols is stored in `SymbolInfo` objects. Symbolic expressions are represented as `SymbolicExpr` objects. There are different variants of these objects depending on the SMT-theory which is used (`LinearExpr` for linear integer arithmetic, `BitvectorExpr` for bitvector arithmetic, `BitvectorExprYices` for the subset of bitvector arithmetic understood by the Yices solver). Symbolic predicates are represented as `SymbolicPredicate` objects. `SMTSolver` is an interface to an SMT solver. It allows to check `SymbolicPredicate`s for satisfiability, compute a satisfying assignment, unsatisfiable cores, or to solve Maximum Satisfiability (MAX-SAT) problems. Different implementations of the `SMTSolver` perform these operations with different solvers (e.g., the `YicesSolver`, the `Z3Solver`, or the `SMTLib2Solver`).

The `DiagnosisConstraintSystem` can check whether a given set of components forms a diagnosis. It builds up the corresponding formula (see Equation 4.1) and solves it using an SMT-solver. On top of that, the `ModelBasedDiagnoser` implements the model-based diagnosis algorithm as described in [13] to find minimal diagnoses quickly. The hitting set tree is built up using `HSTreeNode`s.

The repair engine is implemented in the class `TemplateRepairEngine`. It uses the class `RepTemplExpander` to create and instantiate templates. Repair tem-

plates are represented as a tree of `RepTemplNode` nodes. There are different kinds of nodes in this tree. A `RepTemplVar` represents a program variable in the template, a `RepTemplParam` represents a template parameter, and a `RepTemplOpNode` represents an operation on variables and parameters. The `LinRepairEngine` refines the `TemplateRepairEngine` with aspects that concern the theory of linear integer arithmetic. It performs some tricks to keep the predicates linear during repair computation. The `LinMaxSatRepairEngine` and the `BitMaxSatRepairEngine` implement the heuristics to speed up the repair refinement process.

4.4.7 Examples

This section shows how some versions of the TCAS program from the Siemens suite [7] can be debugged using the symbolic back-end with concolic execution. These benchmarks may also be a good starting point for playing with the symbolic back-end. The TCAS program implements a traffic collision avoidance system for aircrafts. It has 12 integer inputs and is implemented in approximately 180 lines of C code. The program comes in 41 faulty versions, which are contained in the directory `examples/tcas_concolic/`. The reference implementation is used as a specification. The script `execute_all.sh` in `examples/tcas_concolic/` runs FoREnSiC on all versions. Results of the debugging processes can be found in the sub-directory `results`.

Debugging `tcas_v1.c`

The original version of the TCAS program contains the line:

```
89  result = !(Own_Below_Threat()) || ((Own_Below_Threat())
        && (!(Down_Separation >= ALIM())));
```

In `tcas_v1.c` this line is changed to

```
89  result = !(Own_Below_Threat()) || ((Own_Below_Threat())
        && (!(Down_Separation > ALIM())));
```

That is, the faulty version contains a "`>`" instead of a "`>=`" in the last part of this line. Using the parameters as set by the `execute_all.sh` script, FoREnSiC produces the following diagnostic information:

```
[RES] Diagnoses:
[RES] Line 65: "500"
[RES] Line 89: "ALIM()"
[RES] Line 89: "Down_Separation > ALIM()"
[RES] Repairs:
[RES] Replace   Line 89: "Down_Separation > ALIM()"   by
      "Down_Separation - ALIM() >= 0"
[RES] Replace   Line 89: "Down_Separation > ALIM()"   by
      "-Down_Separation + ALIM() <= 0"
```

That is, FoREnSiC finds three single-fault diagnoses, the last one being the expected one. It attempts to repair every diagnosis but succeeds only for the last one. The two repairs which are found are obviously equivalent to the corresponding expression in the reference implementation.

Debugging `tcas_v28.c`

The original version of the TCAS program contains the line:

```
89 return (Climb_Inhibit ? Up_Separation + NOZCROSS :
       Up_Separation);
```

The constant `NOZCROSS` is equal to `100`. In `tcas_v28.c` this line is modified to

```
89 return ((Climb_Inhibit == 0) ? Up_Separation + NOZCROSS :
        Up_Separation);
```

That is, the condition in the ternary `if` is negated in the faulty version. FoREnSiC identifies three components in the above line: The condition (`Climb_Inhibit == 0`) is component c_4, the then-part (`Up_Separation + NOZCROS`) is component c_5, and the else-part (`Up_Separation + NOZCROS`) is c_6.

With the parameter configuration used in `execute_all.sh`, the diagnosis engine does not find out that c_4 may be wrong, it only finds out that c_5 and c_6 may both be wrong. Consequently, FoREnSiC starts to re-synthesize the components c_5 and c_6 simultaneously. Table 4.1 shows this synthesis process in detail. It contains the results of the different iterations of the counterexample-guided repair refinement process. Except for a short odyssey between iteration 8 and 15, the refinement process works quite focused towards its goal. A correct repair is found after 18 iterations already. Note that two components have been re-synthesized here simultaneously. For single faults, the repair process often succeeds even much faster. The computed repair is perfectly readable and useful.

4.5 The Simulation-Based Back-End

4.5.1 Simulation-Based Error Localisation and Repair

A simulation-based debug algorithm containing error localisation and correction has been implemented in FoREnSiC. A dynamic slicing based method for error localisation combined with a dedicated set of mutation operators for automated correction of errors has been implemented and is presented in this subsection.

Since the simulation functionality is the most frequent and time-consuming part of the simulation-based debug, it is best to implement it as reliable and fast as possible. Therefore the simulation-based back-end in FoREnSiC is implemented via simulating the C functionality. This guarantees short run-times and also matches the behavior of the original C code.

Table 4.1: Example: The repair process for `tcas_v28.c`.

Iteration	c_5	c_6	Correct
1	`0`	`0`	✘
2	`0`	`728`	✘
3	`0`	`1902`	✘
4	`0`	`3235`	✘
5	`0`	`Down_Separation + 1`	✘
6	`0`	`Up_Separation`	✘
7	`0`	`Up_Separation + 1000`	✘
8	`0`	`Up_Separation + 1`	✘
9	`0`	`Up_Separation + 2`	✘
10	`0`	`Up_Separation + 3`	✘
11	`0`	`Up_Separation + 4`	✘
12	`0`	`Up_Separation + 5`	✘
13	`0`	`Up_Separation + 6`	✘
14	`0`	`Up_Separation + 7`	✘
15	`0`	`Up_Separation + 8`	✘
16	`0`	`Up_Separation + 100`	✘
17	`Other_Tracked_Alt + 1000`	`Up_Separation + 100`	✘
18	`Up_Separation`	`Up_Separation + 100`	✔

4.5.2 The Observation Points

`Observation Points` define the output responses under analysis of the debugged design. These responses are compared to the ones of the specification, or reference design. The observation points can be seen as places within the design under debug, where the decision is made whether the design satisfies its specification or not. The `Observation Points` in FoREnSiC are defined using the `FORENSIC_output...(...)` functions. They are declared in the file `forensic_instr.h`.

`Observation Points` can alternatively be implemented using the `assert()` statement in the design. In that case there is no need of a specification or a reference design, no file output is performed, no reference outputs are generated. The decision whether the simulation is `passed` or `failed` is made during the simulation of assertions of the processed design.

For further details on observation points please refer to 3.2 Annotating C programs.

4.5.3 Statistical Error Localisation Using Dynamic Slicing

Error localisation is started if the outputs of the implementation do not match the reference outputs of the specification. In other words, when design verification fails. In this subsection the simulation-based error localisation algorithm based on dynamic slicing and statistical ranking of code statements are described.

The simulation-based localisation algorithm is based on storing the simulation

traces and calculating the dynamic slices out of them for all observable outputs of the system. Depending on whether an output response obtained by a given slice is correct or not, the slice is marked as a *passed* or a *failed* one, respectively. A statistical and coverage-based approach has been implemented to assign a score to flowchart nodes based on the number of times they are included into failed slices with respect to the number of times they occur within passed slices. Finally, the nodes of the model, i.e. `FlowChartNodes` (refer to 4.3 The Internal Model) are ranked according to this score, referred to as the *suspiciousness* score.

After the counters are set the decision is made what are the best candidates for repairs. If we have a single error in the model then every failed execution trace must include also the erroneous `FlowChartNode` and the number of *failed* slices can be used as the suspiciousness score for the candidate.

If we have two or more errors in the design then it is possible to apply the following equation to calculate the suspiciousness score for a node s:

$$\text{suspiciousness}(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}}$$

where $failed(s)$ is the number of failed slices for the diagnostic candidate s and $totalfailed$ is the total number of failed slices for the design. Similarly, $passed(s)$ is the number of passed slices for s and $totalpassed$ is the total number of passed slices for the design. The rank depends not only on *failed* information now, but on a *passed* counter for every candidate and on a total number of passed and failed executions. The candidates are sorted according to their *suspiciousness* and the list of candidates for repair is generated.

4.5.4 Mutation-Based Repair

After error localisation has been completed it is possible to try to fix the fault in the design. In the simulation-based back end, a mutation-based repair algorithm is used.

Mutation is a process, where syntactically-correct functional changes are inserted into the program. Traditionally, mutations are performed by perturbing the behavior of the program in order to see if the test suite is able to detect the difference between the original program and the mutated versions. The effectiveness of the test suite is then measured by computing the percentage of detected, or killed, mutations.

In FoREnSiC, we apply mutation operators for correcting erroneous circuits. The goal was to devise an error-matching based correction approach, which would be capable of modeling realistic design errors. In mutation-based repair it is crucial to select a limited number of mutation operators, because the perturbation and simulation of erroneous design implementations with a large number of error locations and mutant operators becomes prohibitively time-consuming.

The mutation operators include replacement of C language operators, which have been divided into several groups: arithmetic operators, relational oper-

ators, assignment operators, unary operators, etc. In addition, numeral mutations are performed by replacing each decimal digit in the numeric values one-by-one with other decimal values. This includes both, integer and floating point numbers and it covers also the array indexes. Also, constants are mutated by inserting unary operators `+` and `-` as well as replacing them by zero.

Since we target system-level hardware descriptions in C, we only focus on algorithmic aspects of the description and do not consider software-specific constructs and related errors, such as dynamic-memory allocation, pointer arithmetic, and file I/O.

In particular, the mutation-based repair algorithm fixes errors using the following mutation operators:

- AOR (arithmetic operator replacement) including `+`, `-`, `*`, `/`, `%`;
- ROR (relational operator replacement): `==`, `!=`, >, <, >=, <=
- LCR (logical connector replacement): `&&`, ||
- ASOR (assignment operator replacement): `+=`, `-=`, `*=`, `/=`, `%=`, `=`
- UOR (unary operator replacement): `+`, `-`, `~`, `!`
- Bitwise operator replacement: >>, <<, `&`, |, `^`
- Bitwise assignment operator replacement: >>=, <<=, `&=`, |=, `^=`
- Increment/decrement operator replacement: `x++`, `++x`, `x--`, `--x`
- Operator mutations which are adding one (C→C`+1`), subtracting one (C→C`-1`), setting to zero (C`=0`), and finally negating C→`-`C.
- Number mutation (decimal digit replacement in integers, floats, and array indexes): `0...9`.

Subsequent to the error location step described in Subsection 4.5.3, which ranks the statements of the program, the suspected error locations are iteratively tried according to their rank. The operators in the statements are, in turn, iteratively substituted by mutation operators, i.e., valid operators from the same group. That is replacing arithmetic operators by arithmetic operators, relational operators by relational ones etc. These iterations stop when the simulation result confirms that the mutated program provides output responses equal to the golden output responses, in other words, a correction has been found. Otherwise the process continues until there exist untried error locations and/or mutant operators, or when a user-specified time limit is reached.

The mutation-based correction method implemented in FoREnSiC is an error-matching approach. Error-matching is known to have the limitation that it is generally not capable of fixing errors that are not included to the model. On the other hand, the mutation-based error-matching provides easy-to-read corrections of system-level descriptions. In addition, the mutation-based approach can fix some of the not modeled errors by proposing alternative but equivalent fixes.

4.5.5 Execution of the Simulation-Based Back-End

The simulation-based back-end is executed using `-b BACKEND` or `--backend=BACKEND` command line option, where `BACKEND` is "*simmut*" or "*simslmut*". It has a number of additional options to control the execution. Those are:

- `-i INPUT_FILE` or `--in=INPUT_FILE` the location of the processed design.
- `-iv INPUTS_VALUES` or `--inval=INPUT_VALUES` inputs for the design and the specification. The format of the input files is defined by FORENSIC_input...(...) functions inside the design and inside the specification. FORENSIC_input...(...) executions will read the data separated by spaces. Data for consequential executions should be in separate lines. The actual location where the data will be readied from is INPUTS_VALUES_file, INPUTS_VALUES_arg has the arguments for the execution. One of these files can be empty if there is no corresponding data for execution.
- `-ovr REFERENCE_OUTPUT_VALUES` or `--outvalr=REFERENCE_OUTPUT_VALUES` is the location where the reference outputs will be written to if a specification is provided (reference input). If the specification is omitted, then `REFERENCE_OUTPUT_VALUES` will be used as reference outputs.
- `-ir REFERENCE_INPUT_FILE` or `--inr=REFERENCE_INPUT_FILE` defines the location of a reference specification.
- `--fasm=FAULT_ASSUMPTION` defines if single or multiple fault assumption is used.

  `1` single fault assumption

  `2` multiple fault assumption

  Default is `1`.

Full description of the command-line arguments can be obtained if using `--help` or `-h` option.

4.5.6 Example

In the following a hands-on tutorial presenting error localisation on a C code example is presented.

The Input Files

As an input file for the design error debug are the design under debug and the specification. Assume that we have one error inside the design, the misuse of an arithmetical operator for example. In the following listing, the code of the bubble sort algorithm example is presented where at Line 10, the condition `if(a[j]>a[j+1])` is replaced by `if(a[j]<a[j+1])`. The name of the erroneous input file is "`bubble_hiera_err.c`".

```
1  #include <stdio.h>
2
3  void bubble(int *a, int n)
4  {
5        int i,j,t;
6        for(i=n-2;i>=0;i--)
7        {
8          for(j=0;j<=i;j++)
9          {
10           if(a[j]<a[j+1])    /* must be a[j]>a[j+1]*/
11           {
12             t=a[j];
13             a[j]=a[j+1];
14             a[j+1]=t;
15           }
16         }
17       }
18 }//end function.
19
20 void main()
21 {
22     int a[6],n=6;
23     int i;
24     a[0]=89;
25     a[1]=-4;
26     a[2]=-67;
27     a[3]=5;
28     a[4]=78;
29     a[5]=11;
30     bubble(a,n);
31     for( i=0; i<=n-1; i++)  {
32      printf("%3d",a[i]);
33   }
34 } //end program.
35  /*  Finally sorted array is: -67 -4 5 11 78 89  */
```

Listing 4.1: The erroneous C program

The specification should not necessarily be in the same format as the program code. It can be for example the `Reference Outputs File` with correct values. In this example, however, we will use the correct implementation as the specification.

The Observation Points

Before starting processing the design with the FoREnSiC tool, it is necessary to set up `Observation Points`. The best practice is to define `Observation Points` in the same location where the `printf()` statements in the original code are located. This is usually best outputs locations, already defined by the designer (Refer to 4.5.2 The Observation Points).

Thus, after modifications the design looks like:

```
#include <stdio.h>
#include <forensic_instr.h>

void bubble(int *a, int n)
{
       int i,j,t;
       for(i=n-2;i>=0;i--)
       {
         for(j=0;j<=i;j++)
         {
           if(a[j]<a[j+1])    /* Fault Location */
           {
             t=a[j];
             a[j]=a[j+1];
             a[j+1]=t;
           }
         }
       }
}//end function.

void main()
{
    int a[6], n=6;
    int i;
    a[0]=89;
    a[1]=-4;
    a[2]=-67;
    a[3]=5;
    a[4]=78;
    a[5]=11;
    bubble(a,n);
    for( i=0; i<=n-1; i++)  {
     FORENSIC_output_int(a[i]);
//      printf("%3d",a[i]);
  }
} //end program.
```

Listing 4.2: The faulty C program with instrumentation.

The correct outputs are commented to avoid excessive outputs during processing. Inside the specification, the `Observation Points` should be inserted in the same place to produce valid outputs:

```
#include <stdio.h>
#include <forensic_instr.h>

void bubble(int *a, int n)
{
       int i,j,t;
       for(i=n-2;i>=0;i--)
       {
         for(j=0;j<=i;j++)
```

```
10          {
11            if(a[j]>a[j+1])
12            {
13              t=a[j];
14              a[j]=a[j+1];
15              a[j+1]=t;
16            }
17          }
18        }
19  }//end function.
20
21  void main()
22  {
23      int a[6], n=6;
24      int i;
25      a[0]=89;
26      a[1]=-4;
27      a[2]=-67;
28      a[3]=5;
29      a[4]=78;
30      a[5]=11;
31      bubble(a,n);
32      for( i=0; i<=n-1; i++)  {
33       FORENSIC_output_int(a[i]);
34  //      printf("%3d",a[i]);
35    }
36  } //end program.
37   /*  Finally sorted array is: -67 -4 5 11 78 89  */
```

Listing 4.3: The correct specification with instrumentation.

Launching the Tool

At the moment we have a specification, the design and the possibility to launch verification, error localisation and correction. As mentioned before, the executable `forensic-bin` of FoREnSiC is located in *tool/build/src*.

If we want to process the design with simulation-based diagnosis and mutation-based repair with dynamic slicing, then the parameters for the FoREnSiC should be:

`-b simslmut -i` *path_to*`bubble_hiera_err.c -ir` *path_to/*`bubble_hiera.c`

Execution Results

The execution should be processed with no errors. Compile errors that appear during the execution imply that the applied mutation for repair is not valid. In this case, correction is counted as invalid.

The last line of the correction is:

```
1 [INF] Execution passed. Valid Repair is '<=' -> '>='
      mutation in 'D.2658<=D.2663' (line 11, col 13).
```

This is a valid repair that fixes the fault. The location of the fault is given in `line 11 column 13` of the design, and the repair is a relational operator replacement.

In the internal representation of the design the `if(a[j]<a[j+1])` statement is divided into many smaller statements, and `D.2658` and `D.2663` are temporal variables, but the location of the error and error type is known. Locating and fixing the error is not a problem any more.

Bibliography

[1] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[2] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *International Conference on Automated Software Engineering*, pages 443–446, 2008.

[3] K.-H. Cfhang, I. L. Markov, and V. Bertacco. Fixing design error with counterexamples and resynthesis. In *Asia and South Pacific Design Automation Conference*, pages 944–949, 2007.

[4] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, 2004.

[5] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[6] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[7] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[8] Inc. Free Software Foundation. *GNU Compiler Collection (GCC) Internals*, 2010.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[10] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[11] R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *International Conference on Formal Methods in Computer Aided Design*, 2011. To appear.

[12] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, 2002.

[13] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[14] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference/International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[15] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Combinatorial sketching for finite programs. In *Proceedings on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006.

[16] A. Sülflow, U. Kühne, G. Fey, D. Große, and R. Drechsler. WoLFram – a word level framework for formal verification. In *International Workshop on Rapid System Prototyping*, pages 11–17, 2009.