

Marcel Gehrke

Bidirectional Predicate Propagation in Frama-C and its Application to Warning Removal

September 7, 2014

supervised by:

Prof. Dr. S. Schupp

Prof. Dr. F. Mayer-Lindenberg

Dipl.-Ing. S. Mattsen

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, 7. September 2014

Marcel Gehrke

Contents

1	Introduction	1
2	Introduction to Data Flow Analyses	5
2.1	Control Flow Graph	5
2.2	Forward Data Flow Analysis	6
2.2.1	If Then Else Block	9
2.2.2	While Loops	12
2.2.3	Formal Definition of Forward Data Flow Analysis	15
2.2.4	Definition of our Forward Analysis	16
2.3	Backward Data Flow Analysis	18
2.3.1	Formal Definition	19
2.4	Worklist Algorithm	21
2.4.1	Calculation using the Worklist Algorithm	24
3	Data Flow Analyses for Bidirectional Predicate Propagation	27
3.1	Forward Data Flow Analysis to Track Predicates	27
3.1.1	Example	27
3.1.2	Formal Definition	30
3.2	Warning Removal	33
3.3	Backward Data Flow Analysis to Track Predicates	33
3.3.1	Example	35
3.3.2	Definition	36
3.3.3	Drawbacks	38
3.4	Warning Insertion	40
3.5	Correctness	41
3.5.1	Must Analysis	41
3.5.2	Correctness of the Backward Analysis	42
3.5.3	Correctness of the Forward Analysis	44
4	Plug-In to Remove Redundant Warnings	45
4.1	Frama-C	45
4.2	Plug-In	46
4.2.1	Warning Collection Module	46
4.2.2	Forward Analysis Module	46
4.2.3	Warning Removal Module	47
4.2.4	Backward Analysis Module	47
4.2.5	Warning Insertion Module	48
4.2.6	Combination of the Modules	49
4.3	Problem with C Pointers	49
4.4	Restrictions in the Plug-In	50

4.5	Improvements to the Plug-In	50
4.5.1	Path Sensitivity	50
4.5.2	Predicate Implication	52
4.5.3	Splitting up Predicates	54
4.5.4	Transfer of Warnings	54
5	Evaluation	57
5.1	Testing of the Implementation	57
5.2	Evaluation with the Value Analysis	58
5.2.1	Warning Removal Feature of the Value Analysis	59
5.2.2	Bidirectional Predicate Propagation in Combination with the Value Analysis	60
5.3	Benchmark	62
5.3.1	IARPA STONESOUP Phase 1 - Null Pointer Dereference for C Version 1.0 Test Suite from NIST	63
5.3.2	IARPA STONESOUP Phase 1 - Memory Corruption for C Version 1.0 Test Suite from NIST	65
5.3.3	Insights of the Benchmark	67
6	Related Work	69
7	Future Work & Conclusion	71
	Bibliography	73
	Appendix	75

List of Figures

2.1	CFG of the Straightforward Example from Chapter 1	5
2.2	CFG with Warnings	6
2.3	Step 1	7
2.4	Step 2	7
2.5	Step 3	7
2.6	Step 4	8
2.7	Step 5	8
2.8	Final Result	8
2.9	CFG of <code>If Then Else</code> Example with Warnings	9
2.10	Branching of <code>If</code> Example	10
2.11	Branch Results of <code>If</code> Example	10
2.12	Combination of <code>If</code> Example	11
2.13	Result of <code>If</code> Example	11
2.14	CFG of <code>While</code> Loop Example with Warnings	12
2.15	First Steps of <code>While</code> Loop Example	13
2.16	First Iteration of <code>While</code> Loop Example	13
2.17	Second Iteration of <code>While</code> Loop Example	14
2.18	Result of <code>While</code> Loop Example	14
2.19	General Forward Data Flow Equations	15
2.20	Data Flow Equations of our Example Forward Analysis	16
2.21	Initial	19
2.22	Intermediate	19
2.23	Result	19
2.24	General Backward Data Flow Equations	19
2.25	Data Flow Equations of our Example Backward Analysis	20
2.26	Lattice	22
2.27	Worklist Algorithm [14]	23
2.28	Lattice of <code>While</code> Loop Example from Subsection 2.2.2	24
3.1	CFG of the Example with Different Kinds of Warnings	28
3.2	Application of the Forward Data Flow Analysis to the Example with Different Kinds of Warnings	29
3.3	Final Result of the Example with Different Kinds of Warnings	29
3.4	Data Flow Equations of our Forward Analysis	31
3.5	Algorithm to Select Warnings We can Remove	33
3.6	CFG of Backwards Analysis Example	34
3.7	CFG with Complete Flow Data	34
3.8	CFG with Removed Warnings	34
3.9	Backwards Analysis Applied to CFG from Figure 3.6	35
3.10	CFG with Inserted Warnings	35

3.11	Forwards Analysis Applied to the Result of the Backwards Analysis Applied to CFG from Figure 3.6	36
3.12	CFG with Removed Warnings	36
3.13	Data Flow Equations of our Backward Analysis	37
3.14	Backwards Analysis Applied to the CFG of <code>If Then Else</code> Example from Subsection 2.2.1	38
3.15	<code>If Then Else</code> Example from Subsection 2.2.1 with Inserted Warning	38
3.16	Example with a Drawback when Applying the Backward Analysis	39
3.17	Drawback Example with Inserted Warning	39
3.18	Algorithm to Select Warnings We Insert	40
3.19	Example of <i>Must</i> Analysis without Reassignment	41
3.20	Example of <i>Must</i> Analysis with Reassignment	41
3.21	Example of <i>May</i> Analysis without Reassignment	42
3.22	Example of <i>May</i> Analysis with Reassignment	42
3.23	Example of Computing Difference Sets for all Predecessors in Insertion Algorithm	43
3.24	Example from Figure 3.23 with Inserted Warnings	43
4.1	CFG with Predicates from Path Sensitivity Included in Flow Data	51
4.2	CFG with Removed Warnings	51
4.3	CFG of the Code from Listing 4.4	55
4.4	CFG of the Code from Listing 4.4	55
5.1	A Test Case for the <i>Confluence</i> Operator	57
5.2	Percentage of Removed Warnings (Y-Axis) for each Program (X-Axis) using only the Forward Analysis	64
5.3	Percentage of Removed Warnings (Y-Axis) for each Program (X-Axis) using the Forward Analysis with <code>-path</code> , <code>-sat</code> , and <code>-sub</code>	64
5.4	Percentage of Removed Warnings (Y-Axis) for each Program (X-Axis) using the Backward Analysis without any Additional Options	67
1	Percentage of Removed Warnings (Y-Axis) for each Program (X-Axis) using only the Forward Analysis of the IARPA STONESOUP Phase 1 - Memory Corruption for C Version 1.0 Test Suite from NIST	76
2	Percentage of Removed Warnings (Y-Axis) for each Program (X-Axis) using the Forward Analysis with <code>-path</code> , <code>-sat</code> , and <code>-sub</code> of the IARPA STONESOUP Phase 1 - Memory Corruption for C Version 1.0 Test Suite from NIST	77

List of Tables

2.1	Warning Removal of the If Then Else Example	12
2.2	Result of Applying the Data Flow Equations to the If Then Else Example	18
2.3	Result of Applying the Backward Data Flow Equations to the Straight- forward Example from Chapter 1	20
2.4	Applying the Worklist Algorithm to the While Loop Example from Sub- section 2.2.2	25
2.5	Result of Applying the Worklist Algorithm to the While Loop Example from Subsection 2.2.2	26
3.1	Result of Applying the Data Flow Equations to Different Kinds of Warn- ings Example from Subsection 3.1.1	32
3.2	Result of Applying the Backward Data Flow Equations to the Example from Section 3.3	37
5.1	Benchmark of the IARPA STONESOUP Phase 1 - Null Pointer Dereference for C Version 1.0 Test Suite from NIST without the Backward Analysis	63
5.2	Benchmark of the IARPA STONESOUP Phase 1 - Null Pointer Dereference for C Version 1.0 Test Suite from NIST with the Backward Analysis	65
5.3	Benchmark of the IARPA STONESOUP Phase 1 - Memory Cor- ruption for C Version 1.0 Test Suite from NIST without the Backward Analysis	65
5.4	Benchmark of the IARPA STONESOUP Phase 1 - Memory Corrup- tion for C Version 1.0 Test Suite from NIST with the Backward Analysis .	66
1	Predicates of Frama-C	75

Listings

1.1	C Code with Possible Run-Time Errors	1
1.2	C Code Annotated by Frama-C with Possible Run-Time Warnings	2
1.3	C Code Annotated by Frama-C with Needed Run-Time Warnings	2
2.1	If Then Else Example	9
2.2	While Loop Example	12
3.1	Example with Different Kinds of Warnings	28
3.2	Backward Analysis Example	34
4.1	Annotated C Code with Different Restrictions	52
4.2	Annotated C Code with Different Restrictions	52
4.3	Annotated C Code with Different Restrictions	53
4.4	Annotated C Code with Restriction Transfer	54
5.1	-remove-redundant-alarms Applied to If Then Else Example	59
5.2	Example Containing Two Division-By-Zero Warnings	60
5.3	Example with one Redundant Warning	61
5.4	Example with Array-Index-Out-Of-Bound Warning	62
1	Excerpt of C Code from ./TC_C_785_v934/src/desaturate.c of the IARPA STONESOUP Phase 1 - Memory Corruption for C Version 1.0 Test Suite from NIST	78

1 Introduction

Frama-C is a framework to statically analyse C code. The framework can be used to verify a program written in C. During the verification, the framework tries to find possible faults in a program. In case the framework cannot prove that a certain fault will not occur, it throws a warning and annotates the C code. These faults can be run-time errors. Examples of run-time errors are division-by-zero, array-index-out-of-bound, signed-overflow, unsigned-overflow, or memory access errors. In C code, there can be a lot of such run-time errors. Therefore, Frama-C can throw many warnings. To determine that the code works correctly, the reviewer has to go through all the warnings and determine if the corresponding fault can occur. Going through many warnings, however, can be really cumbersome and tiresome. Thus, we want to reduce the number of warnings displayed.

Let us first have a look at a case in which we can remove a warning without losing information. Listing 1.1 shows a straightforward C function that contains possible run-time errors.

```
1  void main(int x, int y) {
2      int h = 1/x;
3      h = 1/x;
4      x = y;
5      h = 1/x;
6  }
```

Listing 1.1: C Code with Possible Run-Time Errors

First of all, we have a look at the function and determine all the possible run-time errors. The function has two variables as input, i.e., `x` and `y`. These two variables can have any value of the `int` range, which includes 0. Therefore, when the program calculates the first statement `h = 1/x` in line 2, it could be a division-by-zero error. Unfortunately, a division-by-zero results in an undefined behaviour. Frama-C throws a run-time warning due to the undefined behaviour. The same holds for the second statement in line 3. Here, we could also have a division-by-zero error if `x` has the value zero. In the next line, `x` is reassigned to `y`. However, `y` can also have all values from the `int` range. Hence, the fourth statement, which is in line 5, could be another division-by-zero error if `y` and therefore `x` have the value zero. After we know where a fault can occur in this program, we can compare it against the possible bugs Frama-C finds in the function.

The value analysis [11, 6] is a plug-in of Frama-C with the aim to verify C code. Further, the value analysis throws warnings for possible bugs and annotates the C code with the warnings. Listing 1.2 depicts how the value analysis annotates the C code with the possible faults it can find. Frama-C inserts the annotations as a C comment. The annotations are an assertion and they tell us what kind of fault can occur as well as what has to hold for the fault to not occur. Going through the annotations, we see that

it warns in statement 1,2, and 4 about a division-by-zero error. That is exactly the same result that we obtained by going through the code by hand.

```
1 void main(int x, int y) {
2     /*@ assert Value: division-by-zero: x ≠ 0; */
3     int h = 1/x;
4     /*@ assert Value: division-by-zero: x ≠ 0; */
5     h = 1/x;
6     x = y;
7     /*@ assert Value: division-by-zero: x ≠ 0; */
8     h = 1/x;
9 }
```

Listing 1.2: C Code Annotated by Frama-C with Possible Run-Time Warnings

Having the warnings, we can now have a look if we need all three warnings or if fewer are sufficient. Frama-C annotated the first statement with `/*@ assert Value: division-by-zero: x ≠ 0; */`. The annotation tells us that the first statement could induce a division-by-zero error. For the fault to not occur, $x \neq 0$ has to hold. Further, the annotation shows us that it was annotated by the value analysis and that the analysis asserts that after the first statement $x \neq 0$ holds. The first statement `h = 1/x` is annotated with the same annotation, namely that there could be a division-by-zero error unless $x \neq 0$ holds.

Between the first and second statement we did not reassign the variable `x`. Therefore, `x` evaluates to the same value in both statements 1 and 2. In case `x` evaluates to zero, two faults will occur, first in statement 1 and then in statement 2. However, if `x` could be zero, we would need to ensure that $x \neq 0$ holds, due to either of the warnings corresponding to the first and second statement. The other case is that $x \neq 0$ holds and then both warnings are false positives. A warning is a false positive if the corresponding fault cannot occur. Here, one warning would be sufficient. In case `x` could be zero, either of the warnings warn about the possible fault. Otherwise, if $x \neq 0$ holds, neither of the warnings is needed. Hence, having just the annotation attached to the first statement would be sufficient. The annotation attached to the second statement could be removed.

Another way to see that only the first annotation is sufficient is to have a closer look at the annotation again. In the annotation we assert $x \neq 0$. Hence, reaching the second statement the assertion still has to hold because we did not reassign the variable `x`. Thus, $x \neq 0$ holds in the second statement and we do not need the annotation in the second statement.

In the third statement, the variable `x` is reassigned. After the reassignment, we cannot be certain that $x \neq 0$ still holds. Thus, the annotation corresponding to the fourth statement is needed, because `x` could be zero after the reassignment. In conclusion we can remove one annotation. Listing 1.3 displays the C code with the annotation that a reviewer needs to have a look at. The first annotation covers the possible faults of the first and second statement and the other annotation covers the possible fault of the fourth statement.

```
1 void main(int x, int y) {
2     /*@ assert Value: division_by_zero: x ≠ 0; */
3     int h = 1/x;
4     h = 1/x;
5     x = y;
6     /*@ assert Value: division_by_zero: x ≠ 0; */
7     h=1/x;
8 }
```

Listing 1.3: C Code Annotated by Frama-C with Needed Run-Time Warnings

In the example, we are able to reduce the number of warnings needed from three to two. Here, the additional warning does not increase the review time a lot. However, in a larger C project with thousands of lines of code, the additional warnings increase the review time. Thus, we want to review as few warnings as possible. Further, the analysis of the program can take quite some time. In that case, one wants to run the analysis as few times as possible while fixing bugs. Knowing only the important warnings is of great help then.

In this thesis, we present a way to automatically reduce the overall number of warnings without losing information. To that end, we apply a data flow analysis. Using a data flow analysis, we can track the annotated warnings and remove not needed warnings after the analysis. T. Muske, A. Baid and T. Sanas present the main idea of using a data flow analysis to remove warnings in their paper “Review Efforts Reduction by Partitioning of Static Analysis Warnings” [20]. Unlike them, we do not track expressions in the data flow analysis but use the predicates that Frama-C generates in the annotations. In our example, the predicate of the Frama-C warnings is $x \neq 0$. Besides other papers, we will discuss their paper and why we are using the annotations and not expressions in the related work Chapter 6.

Using a data flow analysis on predicates does not only help with removing warnings but it also provides an insight into what predicates have to hold at a certain point at a program. These predicates allow us to make statements about possible valuations of the variables. Therefore, applying our bidirectional predicate propagation, which we discuss in depth in Chapter 3, we get a better insight into the possible valuations of variables. One possible use of that information is to remove warnings. Further, we show improvements that we can apply due to the fact that we track predicates as well as the plug-in in Chapter 4. Lastly, we still have to evaluate our analysis. In Chapter 5, we show that our implementation works and demonstrate using little example where it can reduce the review efforts. However, to really evaluate our analysis, we present benchmarking results using over 300 C programs. We give example data flow analyses and explain the concept of data flow analyses in the next chapter.

2 Introduction to Data Flow Analyses

In this chapter, we introduce the concept of data flow analyses. The aim of a data flow analysis is to gain information about the possible states of the program at certain points in that program before executing it. One information of interest could be at which points a variable cannot be zero, ruling out a division-by-zero error. However, before we define such a data flow analysis, we need to know how the data can travel the code. To that end, we have a look at what control flow graphs (CFG) are.

2.1 Control Flow Graph

To know how data can traverse through the program, we use a control flow graph to see what paths the data can take.

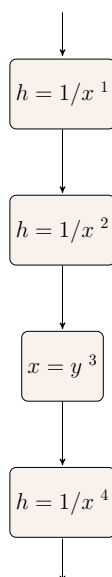


Figure 2.1: CFG of the Straightforward Example from Chapter 1

In Figure 2.1, we see the CFG corresponding to the code of Listing 1.1. A CFG is a directed graph. Each block of the CFG corresponds to one statement of the code. Further, each block is labeled with a number to identify it and distinguish different blocks. The edges show how the program can be traversed and thereby how statements are connected to each other. In the C language, there are constructs like `if then else` and `while` loops. These constructs have two possible successors for one block, leading to branching in the CFG. However, in this example we do not have any branches. Therefore, in our example every block, except for the exit block, has one successor. We can now use the CFG to see how certain data can flow through the program.

2.2 Forward Data Flow Analysis

Picking up the example from the introduction, let us say we are interested in a data flow analysis that tells us that after a certain statement, a variable cannot be zero for the program to run error-free. For that analysis, we need to track variables that cannot be zero after a statement.

To track the variable, we add it to our flow data, which is a set containing all the variables that cannot be zero. These flow data are attached to the edges. The flow data are calculated in each block and the result of that calculation is passed on using the outgoing edges. To calculate the flow data, we need to check if Frama-C threw a division-by-zero warning. In case there is a warning, we need to add the corresponding variable to the flow data.

However, we also have to remove variables from the flow data. We want to know if a variable cannot be zero. In case a variable is reassigned, we remove that variable from the flow data if it was in the flow data. After a reassignment, the variable is again allowed to be zero and we remove that variable. The variable is allowed to be zero until we see a division-by-zero warning with that variable again. We go through the CFG from the entry block to the possible exit blocks of the function. Such a data flow analysis is called forward data flow analysis.

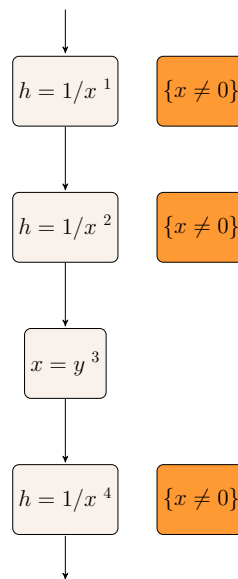


Figure 2.2: CFG with Warnings

For our analysis, we do not only need the CFG of the program but also the statements that threw a division-by-zero warning and the affected variable. Figure 2.2 displays the combination of the CFG and the attachment of warnings to statements. In the figure, we have the CFG and on the right side of the blocks, coloured in orange, the division-by-zero warnings of Frama-C that correspond to that block.

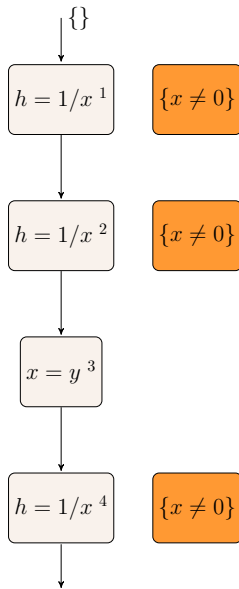


Figure 2.3: Step 1

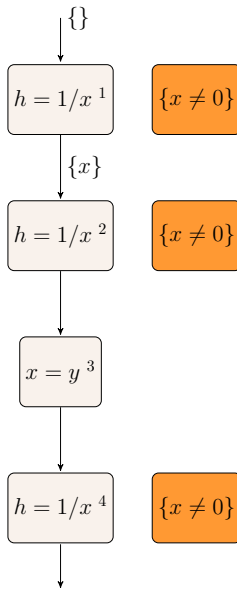


Figure 2.4: Step 2

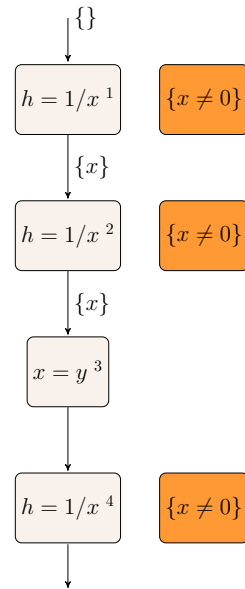


Figure 2.5: Step 3

Now, we can apply our analysis to the program. At the start of the analysis, there is no possibility that we already saw a warning. Therefore, the edge that enters the entry block is annotated with the empty set of flow data. In Figure 2.3, the first step with the initial information is displayed.

The second step is shown in Figure 2.4. In the second step, we encounter a block for the first time. For that block, we have to calculate the flow data. We want to add a variable to the flow data if it is the subject of a division-by-zero warning. Further, we remove variables if they are reassigned. In the first block, we encounter the statement $h = 1/x$. To the right of the block, we see that we have a division-by-zero warning concerning the variable x . Hence, we insert the variable x to our flow data.

Now, we still have to remove the variables that got reassigned in the statement. The variable h is reassigned in that statement. Thus, we have to remove it from our flow data. However, the variable is not in our flow data and removing an element that is not in the set does not change anything. So, our calculation result is the new flow data containing the variable x . We pass that flow data on to the next blocks, by attaching it to the outgoing edges. Here, we only have one outgoing edge with only one successor block.

We enter the second statement with the knowledge that x cannot be zero since the variable is in the flow data. In the third step, we calculate the flow data using the information we gain from the second statement in combination with the entering flow data. In the second block, we again have the statement $h = 1/x$. To the right of the statement we see that we have a division-by-zero warning concerning the variable x . Thus, we add the variable to our flow data. The flow data already contains x so its addition does not change the flow data. In the statement, we reassign h again. Therefore, we have to remove it from our flow data leaving us with the set containing x . We pass

that information on to the next block as displayed in the Figure 2.5.

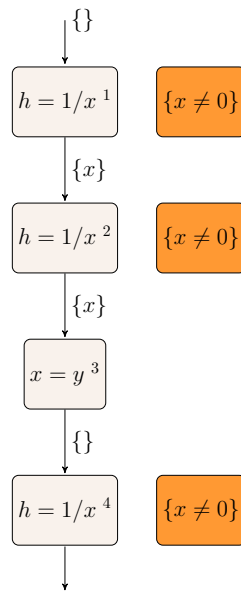


Figure 2.6: Step 4

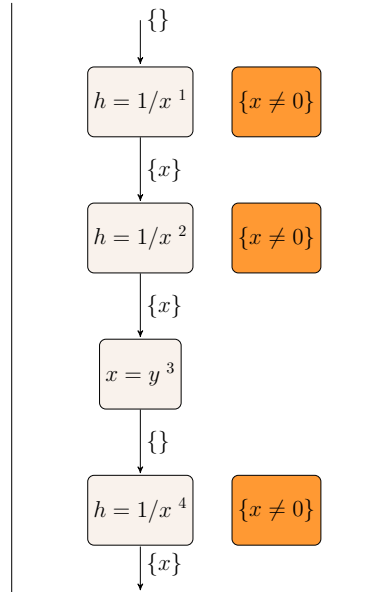


Figure 2.7: Step 5

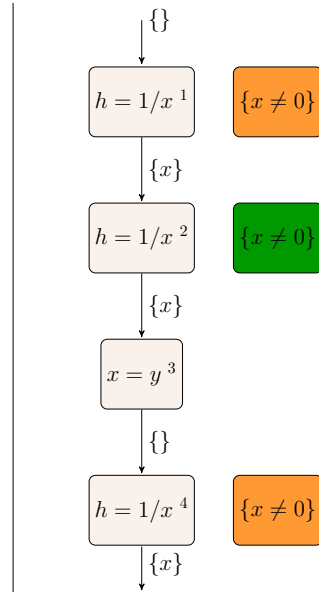


Figure 2.8: Final Result

The third block has the statement $x = y$. There is no division-by-zero warning attached to it. Therefore, we do not add any variables to the flow data. Here, we reassign the variable x . x is in our flow data since it got passed to the third block by the second block. Reassigning x means that we have to remove it from our flow data, resulting in the empty set as flow data at the end of the third block. Figure 2.6 shows us that we pass the empty flow data from the third to the fourth block.

The fourth and last block contains the statement $h = 1/x$. Further, the empty flow data is passed to it from the third block. To the statement belongs a corresponding division-by-zero warning. We add x to our flow data and remove h from it. Thus, we leave the block and the function with the flow data containing x , which can be seen in Figure 2.7.

Now, we can make use of the information we gained by applying the data flow analysis. Whenever we enter a block and have a non-empty flow data, we know that the variables in the flow data cannot be zero in the following block. We enter the first block with an empty flow data. Therefore, we did not gain any additional information there.

For the second block, we have x in our flow data. So, we know that x cannot be zero. The statement has a division-by-zero warning attached to it stating that if x is zero, a fault will occur. However, we gained the information that x cannot be zero in an error-free execution. Hence, we can say that the warning's fault cannot occur and remove the warning. In Figure 2.8, we changed the colour of the warning to green to indicate that the warning can be ignored. The third statement has no warning attached to it so there is nothing for us to do. We enter the fourth statement with an empty flow data. Having empty flow data, we did not gain any additional information about

this statement and we cannot make any further statements about whether the warning's fault can occur. Using the data flow analysis, we come to the same conclusion as in the introduction, i.e., that the second warning is not needed and can be removed.

In the examples so far, we had a straightforward CFG without any branches. However, in a normal C program, there normally are a number of branches. Therefore, let us have a look at how our data flow analysis works with an `if then else` block and a `while` loop.

2.2.1 If Then Else Block

In Listing 2.1, we see a little C function with the Frama-C annotations. The interesting part of the C function is that it contains an `if then else` block. In the function, four division-by-zero warnings appear. The expression `1/x` is calculated and assigned to `h`. At this point, the value analysis of Frama-C warns about a division-by-zero fault. Afterwards, we have the `then` branch. In that branch, we calculate `h = 1/x` and `h = 1/y`. In the `else` branch, we only calculate the statement `h = 1/x`.

```

1 void main(int x, int y) {
2   int h;
3   /*@ assert Value:
4     division-by-zero: x ≠
5     0; */
6   h = 1/x;
7   if (x != 0) {
8     /*@ assert Value:
9       division-by-zero: x
10      ≠ 0; */
11    h = 1/x;
12    /*@ assert Value:
13      division-by-zero: y
14      ≠ 0; */
15    h = 1/y;
16  }
17  else
18    /*@ assert Value:
19      division-by-zero: x
20      ≠ 0; */
21    h = 1/x;
22  }

```

Listing 2.1: If Then Else Example

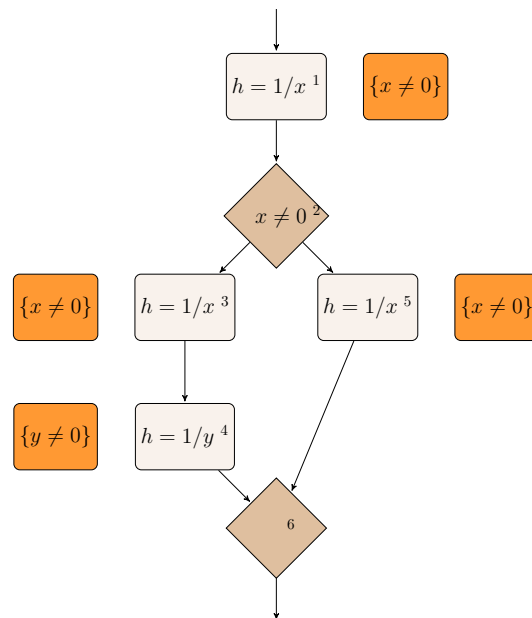


Figure 2.9: CFG of If Then Else Example with Warnings

In Figure 2.9, the graphical representation of the code from Listing 2.1 is depicted in the form of a CFG. In addition to the CFG, we again have the division-by-zero warnings that Frama-C threw analysing the piece of code. The second block of the CFG depicts

the condition for the **if then else** start. To the left of the decision block, we have the **then** branch and to the right the **else** branch. The sixth block combines the two branches again.

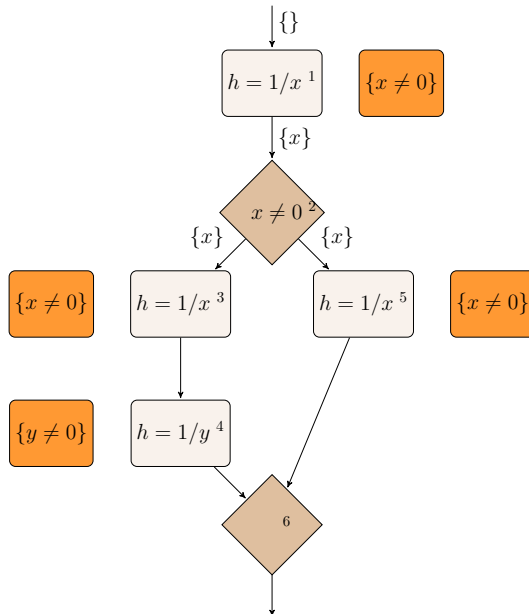


Figure 2.10: Branching of If Example

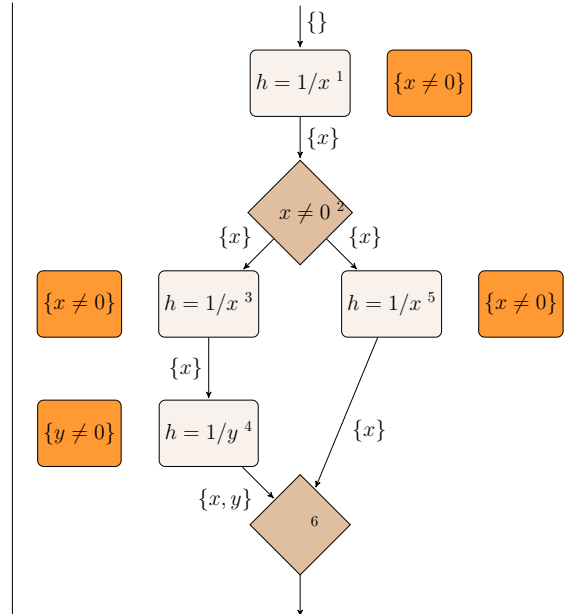


Figure 2.11: Branch Results of If Example

Let us now apply our data flow analysis to the CFG from Figure 2.9 to check whether a variable cannot be zero. The beginning is the same as in the straightforward example from Chapter 1. We enter the function with an empty set as flow data. The first block has a division-by-zero warning attached to it. Thus, we can generate new information and add it to our flow data. The warning contains the variable x , which we now insert into our flow data. Further, we have to remove the reassigned variable h from our flow data, which has no result.

Now, we have to pass the flow data on to the next block. The next block is the condition of the **if then else**, which has no warnings corresponding to it. Here, we do not do anything and only pass on the flow data to the successors of the second block. As we can see in Figure 2.10, we take the flow data of the incoming edge and attach it to the outgoing edges to the third and fifth block.

In Figure 2.11, the next steps are depicted. We calculate the flow data for each branch individually. To that end, we look at the third block in the **then** branch. The third block consists of the statement $h = 1/x$. For that statement, we have an attached warning. Our analysis yields that we now generate the subject variable of the warning and add it to our flow data. Adding x to the flow data does not result in any changes as x is already contained in the set. Additionally we again have to remove the variable h , which does not change the flow data either.

The next block is the fourth. In that block, we add the variable y to our flow data

and remove h . Therefore, we attach the flow data containing x and y to the outgoing edge of the fourth block. Having x and y in our flow data simply means that x cannot be zero and that y cannot be zero. The set actually is a conjunction of the elements in it.

We take a look at the `else` branch. Here, we only have the fifth block. In that block we calculate $h = 1/x$. Hence, we have to add the variable x to our flow data and remove h , leaving the flow data unchanged.

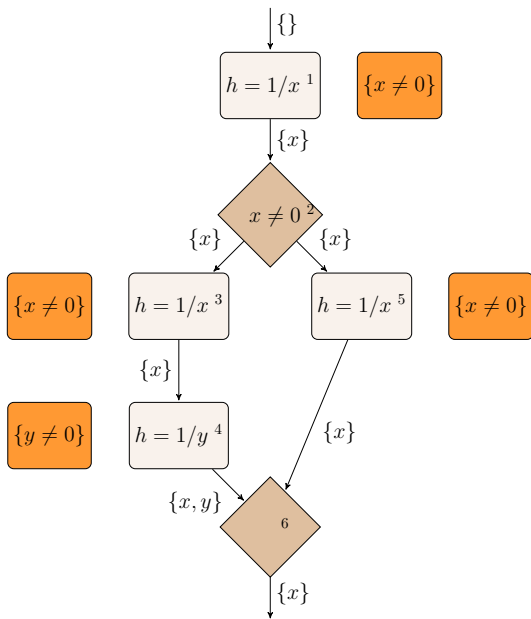


Figure 2.12: Combination of If Example

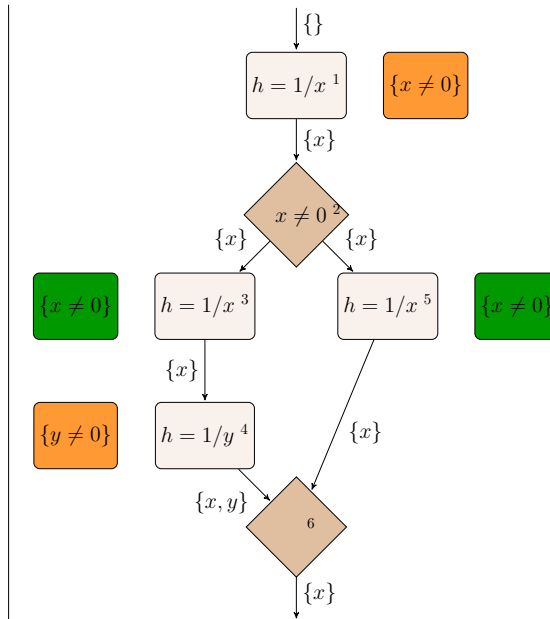


Figure 2.13: Result of If Example

The only task that is left to do in our analysis is to combine the branches again. We have two incoming edges to the sixth block. One branch has the flow data containing x and y attached to it, while the other has the flow data containing x attached to it. In our analysis, we want to know whether a variable cannot be zero. In both branches the variable x cannot be zero. Therefore, x also cannot be zero after we combined the branches. However, y cannot be zero only in one branch. So depending on the path taken, y can either be not zero or y is allowed to be zero. Thus, y is not in the flow data anymore after the combination since we cannot be certain that the variable cannot be zero. Figure 2.12 shows the flow data after the combination. Let us define that in our analysis, a variable has to be in the flow data of each branch to survive the combination. This behaviour corresponds to an intersection of the flow data while we combine them.

After the application of the analysis, we can again go through the gained information and look for information that allows us to remove a warning. In Figure 2.13, the result of the step is shown. To leverage the gained information, we go through each block that has a corresponding warning and check if the variable mentioned in the warning is in the flow data of the incoming edge to that block. Table 2.1 shows the flow data of the

incoming edges to the blocks as well as the variables that are subject of a warning in the block. Every time we have a variable in the set of the incoming edge as well as in the warnings set, we can remove the corresponding warning.

Block	1	2	3	4	5	6
Incoming edge	\emptyset	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$
Warning	$\{x\}$	\emptyset	$\{x\}$	$\{y\}$	$\{x\}$	\emptyset
Warning to be removed			✓		✓	

Table 2.1: Warning Removal of the If Then Else Example

2.2.2 While Loops

Now that we know how we combine branches in our analysis, we have a look at loops, namely `while` loops. Until now, the calculation of the flow data was straightforward. We started at the entry point of the function and worked our way down to the exit points visiting every block only once. With loops we might need to go through blocks multiple times since we have to calculate the information in the loops, until we reach a fixed point. We will see how we reach a fixed point in the following example.

```

1 void f1(int x){
2   int b, y;
3   b = 1;
4   /*@ assert Value:
      division-by-zero: x ≠
      0; */
5   y = 1 / x;
6   while (b < 10) {
7     x = 1;
8     b ++;
9   }
10  /*@ assert Value:
      division-by-zero: x ≠
      0; */
11  y = 1 / x;
12 }

```

Listing 2.2: While Loop Example

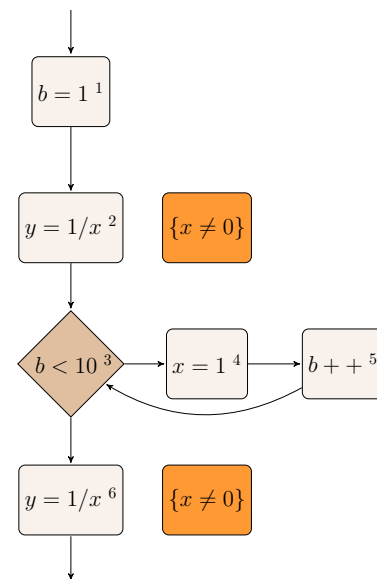


Figure 2.14: CFG of While Loop Example with Warnings

Listing 2.2 contains a function written in C, with a `while` loop. Further, the code is annotated by Frama-C's value analysis with division-by-zero warnings. In the function, we first set `b` to 1. Afterwards, we execute the statement `y = 1/x`. Next, we have a

loop, which we execute until b is greater or equal to 10. In the loop, we set x to 1 and increase b . Lastly, we calculate $y = 1/x$ after the loop.

In Figure 2.14, we see the graphic representation of the code as well as the corresponding annotations. The interesting part of the CFG is the `while` loop. The third block contains the condition of the `while` loop. To the right of the third block, we have the code from inside the loop. Below the third block, we have the case that we leave the `while` loop and go on with the next statement.

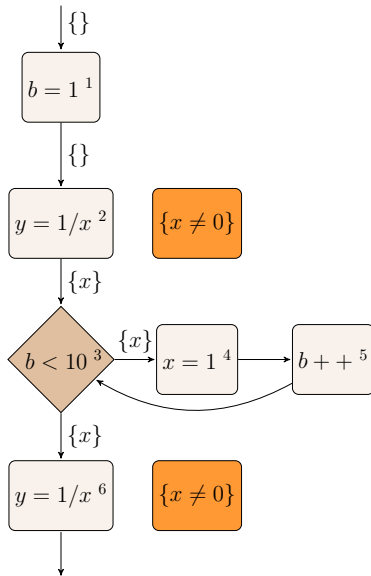


Figure 2.15: First Steps of While Loop Example

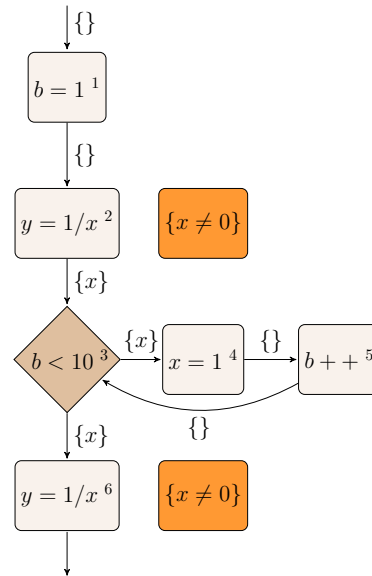


Figure 2.16: First Iteration of While Loop Example

Let us now apply our data flow analysis to the CFG of Figure 2.14 to find out whether a variable cannot be zero. In Figure 2.15, the first steps are depicted. We start with the empty set as flow data. In the first block, we set b to 1. There is no warning attached to that statement. Without a warning, we do not add any new information to the flow data. Further, we have to remove b from the flow data since it is reassigned. Removing it does not change the flow data and we still have the empty set.

The statement of the second block has an attached warning. The warning states that if x is not equal to zero, then a division-by-zero fault will occur. Therefore, we generate the information that x cannot be zero after this block and insert x to our flow data. In that block, we reassign y , which we therefore have to remove from the flow data. After the second block, the flow data contains the variable x . The third block only contains the condition of the `while` loop. We do not use this condition to calculate new flow data. Therefore, we simply pass on the incoming flow data to the successor blocks four and six.

In Figure 2.16, the first iteration of the loop is depicted. In the fourth block, x is reassigned. Hence, we have to remove x from the flow data, leaving us with an empty

set. In the next block, we increase b . There is no warning attached to it, so we do not generate any new flow data. Since we reassign b while we increase it, we have to remove it from the flow data. Next, we pass the flow data on to the third block. The third block has now two incoming edges with corresponding flow data. Thus, we combine these information as we did in Subsection 2.2.1.

We intersect the flow data from the two incoming edges, i.e. $\{\}$ and $\{x\}$, yielding that no variable is in the flow data. We now have to propagate the updated flow data. Therefore, we change the flow data of the edges from the third to the fourth and to the sixth block to the empty set. However, now the flow data that we used to calculate the information inside the loop changed. Thus, we have to enter it again until we reach a fix point where the information does not change anymore between iterations.

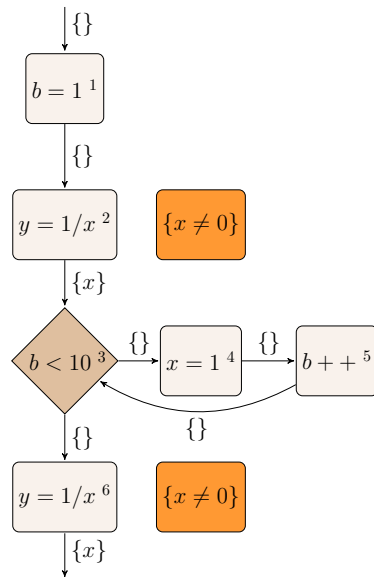


Figure 2.17: Second Iteration of While Loop Example

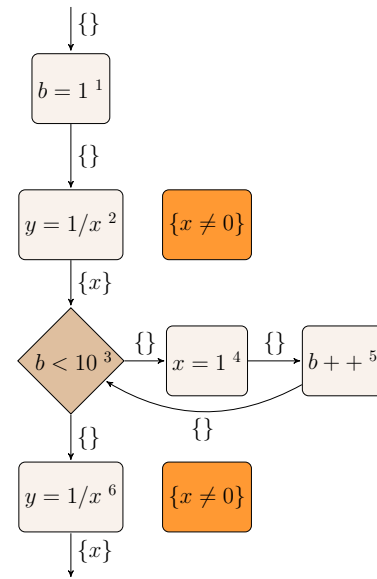


Figure 2.18: Result of While Loop Example

In Figure 2.17, we see the second and final iteration of the loop as well as the last steps of applying the data flow analysis. In the fourth block, we again remove x from the flow data. However, this time x is not in the flow data of the incoming edge. Therefore, removing x from the flow data has no effect. Next, we have to remove b due to the statement in the fifth block.

Now, we reach the third block for the third time. The flow data coming from the block 2 still contains x . The edge coming from block 5 carries the empty set. Combining these flow data yields the empty flow data. Passing on the information to the successors does not change the information. Thus, we reached a fixed point and we can go on with the blocks after the `while` loop. The sixth and last block has a corresponding warning. Due to the warning, we generate the fact that x cannot be zero after the block and add x to the flow data. Removing the reassigned y does not change the flow data.

After we applied the data flow analysis, we can again check if there are warnings that can be removed. In this example, we have two warnings. The incoming flow data to blocks with a corresponding warning are empty. Hence, we cannot remove any warning. By looking at the CFG, we can see why we cannot remove any warning. We have one warning before and one after the while loop. Unfortunately, we reassign x in the loop. Therefore, we cannot say that on all paths that reach the second warning x cannot be zero. Due to that reason, we cannot remove any of the warnings.

Now that we have an intuition of how a forward data flow analysis works, let us define it formally.

2.2.3 Formal Definition of Forward Data Flow Analysis

We can formulate the way we calculate the next flow data as mathematical equations. These data flow equations can be calculated for each block of the CFG. Further, we have to iteratively calculate them until we reach a fixed point. Kildall was the first to develop this approach in his paper “A Unified Approach to Global Program Optimization”[18]. The basic idea is that we solve for each block n of the CFG the following equations:

$$\begin{aligned} In_n &= \text{confluence}_{p \in \text{pred}_n}(Out_p) \\ Out_n &= \text{transfer}_n(In_n) \end{aligned}$$

Figure 2.19: General Forward Data Flow Equations

The first equation from Figure 2.19 states that what goes into a block is defined by the combination of the predecessors. In our data flow analysis so far, we combined the predecessors by taking the intersection of the flow data. If we wanted to know if on at least one path reaching this point in the program, a variable cannot be zero, then we would take the union.

With the intersection, a piece of information has to be true on every path to survive the combination. For the union, a piece of information has to be true on at least one path to survive the combination. Further, if the *confluence* operation is the intersection, then the analysis is called a *must* analysis. For a *must* analysis a piece of information must be true reaching this point in the program. To ensure that the information is indeed true, it must be true on all incoming paths. Only then the information is in the flow data. Otherwise, if the *confluence* operator is the union it is called a *may* analysis. For a *may* analysis a property may only be true at this point in the program. For a *may* analysis it is sufficient if one path reaching a certain point in the program yields that this information holds. In that case the information is in the flow data. The *confluence* operator is not restricted to the intersection and union but can also be another operator. However, intersection and union are the most commonly used *confluence* operators.

The second equation deals with what we propagate on from the block. In the so-called *transfer* function, we define the calculations we do in a block. The calculations we

perform define the information of interest for our analysis. The *transfer* function for our analysis would have to state that we add a variable to the flow data if it is the subject of a division-by-zero warning. Further, the *transfer* function for our analysis removes any variable that are reassigned.

2.2.4 Definition of our Forward Analysis

Let us now formally define our data flow analysis:

$$\begin{aligned}
 In_n &= \begin{cases} \emptyset & n \text{ is the entry block} \\ \bigcap_{p \in \text{predecessors}(n)} Out_p & \text{otherwise} \end{cases} \\
 Out_n &= Transfer_n(In_n) \\
 Transfer_n(In_n) &= (In_n \cup Gen_n) \setminus Kill_n \\
 Gen_n &= \{e \mid e \text{ is contained in a division-by-zero warning in block } n\} \\
 Kill_n &= \begin{cases} \{e\} & e \text{ is reassigned in } n \\ \emptyset & \text{no variable is reassigned in } n \end{cases}
 \end{aligned}$$

Figure 2.20: Data Flow Equations of our Example Forward Analysis

The first equation from Figure 2.20 defines the flow data at the entry of a block. As we always start with empty flow data, we assign the empty flow data to the entry block. Otherwise, if we are somewhere in the CFG, we take the intersection of the predecessor blocks. Since our *confluence* operator is the intersection, we have a *must* analysis and ensure that we know that a certain variable cannot be zero.

The second equation calls the *transfer* function. The third equation defines the *transfer* function, i.e., the calculation inside a block resulting in the flow data at the exit of the blocks. In the equation, we define that we take the information we had entering the block and might add a certain piece of information as well as remove some again. What we add is defined in the *gen* set. The fourth equation defines the *gen* set. A variable is in the *gen* set in case we have a division-by-zero warning attached to the block we are currently working on and that variable caused the warning.

The fifth and last equation defines the variables we remove from the flow data. We add a variable to our *kill* set in case it is reassigned in the current block. If no variable is reassigned, then our *kill* set is empty. In short, the expression $(In_n + Gen_n) - Kill_n$ means we take what we had and add to it the variables we generated. From that set, we remove all the variables that got reassigned.

Let us now have a look at two example calculations of the expression $(In_n + Gen_n) - Kill_n$. The first example is that we have the statement $x = 1/x$ and our *in* set is the empty set. That statement has a division-by-zero warning. Therefore, x is in our *gen* set. Adding the set containing x to the empty set results in the singleton set containing x . The *kill* set consists of the variable x . Removing the *kill* set from the combination of *in* and *gen* results again in the empty set. Hence, in case we generate and kill the same variable in a block, it will not be in the flow data at the exit of the block.

Another example is that we have the statement $h = 1/x$ and our *in* is the set containing h . Here, the *gen* set contains x . Adding the *gen* set to the *in* set results in the set containing h and x . The *kill* set is $\{h\}$. Removing h leaves us with the set containing x . Thus, the flow data at the exit of the block contains only x .

Example Calculation

Now, we apply the data flow equations to the example of Subsection 2.2.1.

First of all, we have to determine the flow data that flow into the first block. Here the first equation is of help. It defines that in case we are dealing with the entry block, our incoming flow data for the first block is the empty set. After we know the flow data entering the block, we can apply the *transfer* function.

For the *transfer* function, we have to calculate the *gen* and *kill* set. The first block has an attached division-by-zero warning. Therefore, our *gen* set will not be empty. The warning states that if x is zero, a run time error will occur. Thus, we add x to our *gen* set so that we know that after this block x cannot be zero anymore.

In the first block, we reassign h . Since we reassign h in this block, we have to add it to our *kill* set. Having the *in*, *gen*, and *kill* set, we can calculate the *transfer* function. Inserting them, we get $Out_1 = (\emptyset + \{x\}) - \{h\}$, which is the set containing x .

Moving on to the second block, we start by calculating the *in* set. To calculate the *in* set, we have to apply our *confluence* operator to the predecessors of the second block. However, the second block only has one predecessor. Therefore, we have $In_2 = Out_1$, which is $\{x\}$. The *gen* set of the second block is empty because there is no warning attached. Further, we do not reassign any variable leaving the *kill* set empty as well. Applying the *transfer* function we get $Out_2 = In_2$ meaning we simply pass on the flow data.

The third block has only the second block as predecessor, so we get that $In_3 = Out_2$. Further, we have a warning attached about x , making up our *gen* set. The *kill* set contains h since we reassign it. So we have $Out_3 = (\{x\} + \{x\}) - \{h\} = \{x\}$.

Applying the first equation from our data flow equations to the fourth block we get that $In_4 = Out_3$. The *gen* set for this block consists of y and the *kill* set of h , leaving us with $Out_4 = (\{x\} + \{y\}) - \{h\} = \{x, y\}$.

Going on with the fifth block in the other branch. The fifth block is equal to the third block. So, all the calculations we did there apply here in the same way.

The last block has one interesting aspect. Until now, the application of the *confluence* operator had no effect. In the last block, we apply our *confluence* operator to two

predecessors. The application has an effect this time and changes the flow data. The fourth and fifth block are predecessors of the sixth block. Hence, we have that $In_6 = Out_4 \cap Out_5 = \{x, y\} \cap \{x\} = \{x\}$. In the sixth block, we do not reassign anything nor do we have a division-by-zero warning leaving us with $Out_6 = In_6$. In Table 2.2, all the sets calculated are depicted.

Block	1	2	3	4	5	6
In	\emptyset	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$	$\{x\}$
Warning	$\{x\}$	\emptyset	$\{x\}$	$\{y\}$	$\{x\}$	\emptyset
Gen	$\{x\}$	\emptyset	$\{x\}$	$\{y\}$	$\{x\}$	\emptyset
Kill	$\{h\}$	\emptyset	$\{h\}$	$\{h\}$	$\{h\}$	\emptyset
Out	$\{x\}$	$\{x\}$	$\{x\}$	$\{x, y\}$	$\{x\}$	$\{x\}$
Warning to be removed			✓		✓	

Table 2.2: Result of Applying the Data Flow Equations to the If Then Else Example

Having the set representation, we can easily select warnings that we can remove. For every block, we compare the *in* set and the *warning* set. In case a variable is in both sets, we can remove the warning in that block with that variable. Using the data flow equations, we come to exactly the same result as we did in Subsection 2.2.1, namely that we can remove the warning attached to the third and fifth block.

2.3 Backward Data Flow Analysis

So far we only looked into a forward data flow analysis. However, there are also backward data flow analyses. Further, our bidirectional predicate propagation consists of a forward and a backward analysis. Basically, there only is one difference between forward and backward data flow analyses. In a forward analysis, we start at the entry point of the function and check every successor until we reach the exit points. In a backward analysis, we start at the exit points of the function and check every predecessor until we reach the entry point.

Let us apply our analysis to the CFG of Figure 2.2 as a backward analysis. Now our starting point for the analysis is the fourth block. Initially, our flow data is empty, which can be seen in Figure 2.21. Entering the fourth block, we have to calculate our *transfer* function. We add x to our flow data and remove h from it. We pass on our flow data containing x to the third block, as shown in Figure 2.22.

Figure 2.23 shows the remaining steps. In the third block, we reassign x . Therefore, we remove x from the flow data, leaving us with empty flow data, which we pass on to the second block.

In the second block, we add x to our flow data and pass it on to the first block. There, we add x and remove h , which does not change the flow data, leaving us with the flow data containing only x .

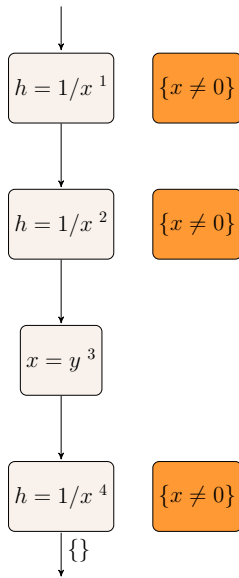


Figure 2.21: Initial

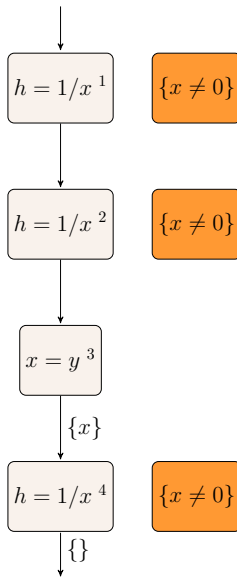


Figure 2.22: Intermediate

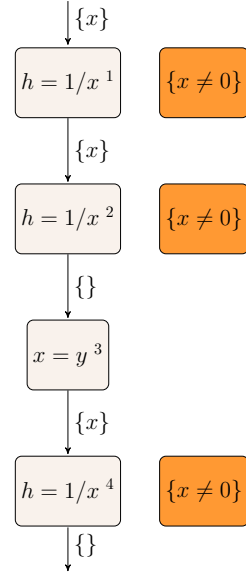


Figure 2.23: Result

The result of applying the analysis backwards tells us which variable cannot be zero already. Using the forward analysis, we only know from which point onwards a variable cannot be zero. However, the variable valuation is not changed by a warning about it. Applying the backwards analysis, we push that point as far as possible upwards to the last assignment of that variable. Thereby, we get an understanding since when a variable actually cannot be zero.

2.3.1 Formal Definition

Let us now have a look at the data flow equations for our analysis as a backward analysis. Figure 2.24 depicts the general backward data flow equations. The main difference to the forward equations is that *in* and *out* are exchanged. Further, we look at successors and not the predecessors of the blocks.

$$\begin{aligned} Out_n &= Confluence_{p \in succ_n}(In_p) \\ In_n &= Transfer_n(Out_n) \end{aligned}$$

Figure 2.24: General Backward Data Flow Equations

In a backward analysis, we start at the exit points of the function. Thus, we have to assign the initial information to the exit blocks. We do that in the first case of the *out* equation in Figure 2.25. The second case of the *out* equation defines how we combine branches. We take all the successors of the current block and apply the *confluence* operator to them. In our *must* analysis the *confluence* operator is the set intersection.

$$\begin{aligned}
In_n &= (Out_n \cup Gen_n) \setminus Kill_n \\
Out_n &= \begin{cases} \emptyset & n \text{ is a exit block} \\ \bigcap_{p \in \text{successors}(n)} In_p & \text{otherwise} \end{cases} \\
Gen_n &= \{e \mid e \text{ is contained in a division-by-zero warning in block } n\} \\
Kill_n &= \begin{cases} \{e\} & e \text{ is reassigned in } n \\ \emptyset & \text{no variable is reassigned in } n \end{cases}
\end{aligned}$$

Figure 2.25: Data Flow Equations of our Example Backward Analysis

Unlike the forwards analysis in the backwards analysis the *transfer* function defines the *in* set and uses the *out* set. Further, we still use the *gen* and *kill* set as we do in the forward analysis. In the next subsection, we see how we apply the equations from Figure 2.25.

Example Calculation

Let us apply the equations to the CFG of Figure 2.2. Table 2.3 shows the result of the application.

Block	4	3	2	1
Out	\emptyset	$\{x\}$	\emptyset	$\{x\}$
Warning	$\{x\}$	\emptyset	$\{x\}$	$\{x\}$
Gen	$\{x\}$	\emptyset	$\{x\}$	$\{x\}$
Kill	$\{h\}$	\emptyset	$\{h\}$	$\{h\}$
In	$\{x\}$	\emptyset	$\{x\}$	$\{x\}$

Table 2.3: Result of Applying the Backward Data Flow Equations to the Straightforward Example from Chapter 1

The fourth block is the exit point of our function. Hence, we set Out_4 to the empty set. Next, we have to compute the *transfer* function for the fourth block. The *gen* set for that block consists of x and the *kill* set of h . Therefore, In_4 is the set containing x .

In the next block, we reassign x , which leads to the removal of x from the flow data leaving us with an empty set. In the next two blocks, we add x to the flow data. Thus, the *in* set of both consists of the variable x . The result of applying the equation is exactly the same as what can be seen in Figure 2.23. Having the data flow equations for forward and backward analysis, we still need a way to automatically compute them.

The worklist algorithm [14] is one method to automatically compute the result of a data flow analysis.

2.4 Worklist Algorithm

So far we presented a formal definition for both forward and backward data flow analysis. In this section, we present a way to automatically calculate the data flow equations until we reach a fixed point. The method to automatically apply the equations is called worklist algorithm.

We have to provide the worklist algorithm with a so-called instance of a monotone framework. An instance of a framework is a combination of the data flow equations and the CFG we used for the computation. In the following, we use label, which refers to the label of the CFG blocks. An instance consists of a six-tuple $(\mathcal{L}, \mathcal{F}, F, E, \iota, f)$:

- \mathcal{L} is the complete lattice of the framework, our property space.
- \mathcal{F} is the space of monotone *transfer* functions of the framework.
- F is the finite flow, the set of edges from the CFG.
- E is the finite set of extremal labels, either the entry block or exit blocks.
- ι is an extremal value from \mathcal{L} for the extremal labels.
- f is a mapping from labels to transfer functions in \mathcal{F} .

A lattice is a partially ordered set where any two elements always have a supremum (join) and an infimum (meet). Let us take the source code shown in Listing 2.1. The source code has three variables, namely \mathbf{x} , \mathbf{y} , and \mathbf{h} . In our case, we only add the variables to our flow data since we only looked at division-by-zero warnings.

Figure 2.26 shows how the lattice with these variables looks like. The lattice is actually a complete lattice. For a complete lattice, also all subsets have a join and a meet. Specifically, this lattice is the power set of the variables in the program ordered by an is-superset-of relation. Thereby, we have in the lattice all possible combinations that could be in the flow data. A lattice built from a power set is always a complete lattice. So let us have a look at the lattice.

First of all, we have our top element, which is the empty set. In case our flow data would always be the empty set, we would not remove any warning. Therefore, it is safe and sound to always use the empty set. However, just using the empty set makes us imprecise. The next levels of the lattice depict all combinations of the three variables. On the first level we have them individually. On the second level, we have two of them combined and on the last, all three of them together. So by taking the join of any two elements of the lattice, we get the result of the combination in our analysis.

The space of functions, \mathcal{F} , guarantees that we have a function $\mathcal{L} \rightarrow \mathcal{L}$. That means that after we apply the *transfer* function, we are still inside the lattice. \mathcal{F} looks like this for our analyses: $\{f : \mathcal{L} \rightarrow \mathcal{L} \mid \exists l_k, l_g : f(l) = (l - l_k) + l_g\}$. An element of this space

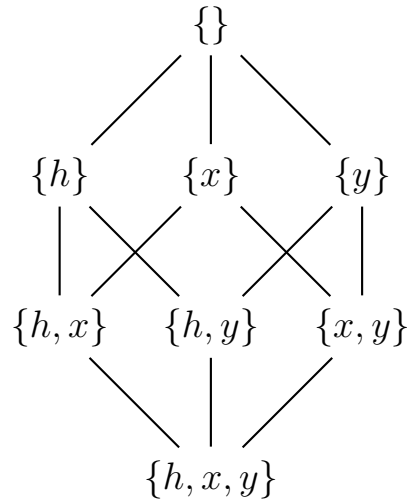


Figure 2.26: Lattice

is a function that stays in the lattice and there exists for label l a *gen* and *kill* set, l_g and l_k , so that we can compute the transfer function. The *gen* and *kill* set have to be constant for every block. The definitions of our *gen* and *kill* set are independent of the *in* set. Thus, the sets are constant for every block.

The flow, F , is basically our CFG. We use it to define how we can transverse the blocks. Further, we define if we have a forward or backward analysis by either giving the flow from the function entry to the exits for a forward analysis or reversing it for a backward analysis.

The extremal labels are the set containing the starting points for the analysis. Thus, the extremal labels are the entry point of the function for a forward analysis and the exit points of the function for a backward analysis. The extremal value, ι , is the data value that should be assigned to the extremal labels. Normally, the top symbol of the lattice is used for this. In our analysis, this is the empty set, which we assigned to the first block in the forward analysis. f , is the specific mapping from the current block to the corresponding *transfer* function which we have to apply in that block.

Now that we know what an instance of a monotone framework is, we can have a look at the worklist algorithm. The worklist algorithm computes the maximal fixed point (MFP) for a monotone framework. The MFP is a fixed point, which is always computable. We are interested in this decidable fixed point since otherwise our analysis might not terminate [14].

The worklist algorithm is depicted in Figure 2.27. In the first step of the worklist algorithm, we initialise the worklist with an empty list and add the edges, F , to the worklist. For that, we go through the flow set and add every edge (l, l') to the worklist. After we added all edges to the worklist, we assign the initial value to the extremal labels. In our forward analysis, we set the *in* set of our entry point to the empty set. Further, we set all the blocks that are not extremal labels to the bottom symbol of the

lattice. In the lattice of Figure 2.26 the bottom symbol is the set containing h , x and y .

In the second step all the work is done. As long as the worklist is not empty we take the first element, an edge, from that list. Now we take that edge and split it up in the source l and the destination l' . Afterwards, we check if we generate new information. We do this by checking if $f_l(\text{Analysis}[l]) \not\sqsubseteq \text{Analysis}[l']$ holds. In our analysis, \sqsubseteq is the superset relation. Further, $f_l(\text{Analysis}[l])$ is the *transfer* function for the block l .

```

Step1   Initialisation (of W and Analysis)
        W := nil;
        for all (l, l') in F do W := cons((l, l'), W);
        for all l in F or E do
            if l ∈ E then Analysis[l] := ι else Analysis[l] = ⊥L
Step2   Iteration (updating W and Analysis)
        while W ≠ nil do;
            l := fst(head(W)); l' = snd(head(W)); W := tail(W);
            if fl(Analysis[l]) ⊈ Analysis[l'] then
                Analysis[l'] := Analysis[l'] ⊔ fl(Analysis[l])
            for all l'' with (l', l'') in F do W := cons((l', l''), W);
Step3   Presenting the result
        for all l in F or E do
            MFPentry(l) := Analysis[l];
            MFPexit(l) := fl(Analysis[l]);

```

Figure 2.27: Worklist Algorithm [14]

In case we generate new information, we propagate it on. The new information is propagated via the assignment $\text{Analysis}[l'] := \text{Analysis}[l'] \sqcup f_l(\text{Analysis}[l])$. The symbol \sqcup is the *confluence* operator of our analysis. So, we calculate the new information for block l' , by applying the *confluence* operator to the information of the entry of block l' and the information from the exit of block l . After we updated the information of l' , we add all the edges from l' to its successors to the worklist. We do this to ensure that the newly generated information is passed on. To summarise, the worklist is a list of pairs (l, l') that tells us that the analysis returned new information for block l .

We iterate through the second step until our worklist is empty. When the list is empty, we still have to present the result. We go through all the blocks and assign the flow data to the entry and exit of each block.

2.4.1 Calculation using the Worklist Algorithm

To apply the worklist algorithm to our while loop example, we need to define the instance of the monotone framework for the example. Figure 2.28 shows the complete lattice, \mathcal{L} , of the example. The set of flow, F , contains the edges (1, 2), (2, 3), (3, 4), (4, 5), (5, 3), (3, 6). Further, the set of extremal labels, E , consists of the label 1. Our ι is the top element from \mathcal{L} , which is the empty set. The *transfer* function is defined in Table 2.20.

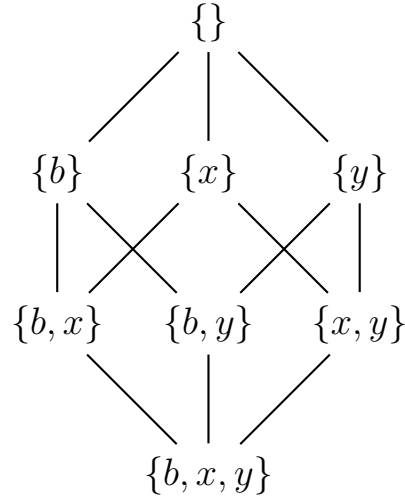


Figure 2.28: Lattice of While Loop Example from Subsection 2.2.2

Let us now apply the worklist algorithm. Table 2.4 depicts the steps of the application of the worklist algorithm.

- **Initialisation & Iteration 1:** The extremal block, 1, is initialised with the empty set. We assign the bottom element of the lattice, which is the set $\{b, x, y\}$, to all the other blocks. Further, we see the first element of the worklist, the edge from the first to the second block. Having that edge, we can apply the steps from the second step of the worklist algorithm. To apply the steps, we first have to check if $f_1(\text{Analysis}[1]) \not\sqsubseteq \text{Analysis}[2]$ holds. $f_1(\text{Analysis}[1])$ is the *transfer* function applied to the first block with the information that the analysis calculated for the first block.

Here, $\text{Analysis}[1]$ is the empty set. Therefore, we apply the *transfer* function with the empty *in* set. Applying the *transfer* function to the first block returns the empty set. In the initialisation step, we assigned $\text{Analysis}[2]$ to the set $\{b, x, y\}$. Now we have to check if $\emptyset \not\sqsubseteq \{b, x, y\}$ holds, which is the same as checking $\emptyset \subset \{b, x, y\}$. The check, $\emptyset \subset \{b, x, y\}$, holds.

Therefore, we set $\text{Analysis}[2]$ to $\text{Analysis}[2] \sqcap f_1(\text{Analysis}[1])$. Our *confluence* operator \sqcap is the intersection, which yields $\text{Analysis}[2] := \{b, x, y\} \cap \emptyset = \emptyset$.

$$W := ((2, 3), (3, 4), (4, 5), (5, 3), (3, 6)), \perp := \{b, x, y\}$$

Iteration	W	1	2	3	4	5	6
Init & 1	$((1, 2), W)$	\emptyset	\perp	\perp	\perp	\perp	\perp
2	$((2, 3), W)$	\emptyset	\emptyset	\perp	\perp	\perp	\perp
3	$((3, 4), (3, 6), W)$	\emptyset	\emptyset	$\{x\}$	\perp	\perp	\perp
4	$((4, 5), (3, 6), W)$	\emptyset	\emptyset	$\{x\}$	$\{x\}$	\perp	\perp
5	$((5, 3), (3, 6), W)$	\emptyset	\emptyset	$\{x\}$	$\{x\}$	\emptyset	\perp
6	$((3, 4), (3, 6), W)$	\emptyset	\emptyset	\emptyset	$\{x\}$	\emptyset	\perp
7	$((5, 3), (3, 6), W)$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\perp
8	$((3, 6), W)$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\perp
9	$((3, 4), (4, 5), (5, 3)(3, 6))$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
10	$((4, 5), (5, 3)(3, 6))$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
11	$((5, 3)(3, 6))$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
12	$((3, 6))$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
13	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

Table 2.4: Applying the Worklist Algorithm to the `While` Loop Example from Subsection 2.2.2

Each step in Table 2.4 shows the values at the entry point of the blocks at the very beginning of each iteration of the algorithm. Thus, the update of $Analysis[2]$ is not already shown in this step but appears only in the next step. The only thing left to do is to add the edge $(2, 3)$ to the worklist.

- Iteration 2: Now we inspect the edge $(2, 3)$. First of all, the check $f_2(Analysis[2]) \not\sqsubseteq Analysis[3]$, which is $\{x\} \subset \{b, x, y\}$, evaluates to true. Afterwards, we assign $\{x\} \cap \{b, x, y\} = \{x\}$ to $Analysis[3]$. Lastly, we add the edges $(3, 4)$ and $(3, 6)$ to the worklist.
- Iteration 3: The first element in the worklist is the edge $(3, 4)$. Our check $\{x\} \subset \{b, x, y\}$ holds. Now, we set $Analysis[4]$ to $\{x\} \cap \{b, x, y\} = \{x\}$ and add the edge $(4, 5)$ to the worklist.
- Iteration 4: For the edge $(4, 5)$, the check is $\emptyset \subset \{b, x, y\}$. Therefore, we set $Analysis[5]$ to $\emptyset \cap \{b, x, y\} = \emptyset$ and add the edge $(5, 3)$ to the worklist.
- Iteration 5: Taking the first element of the worklist, we get the edge $(5, 3)$. The check $\emptyset \subset \{x\}$ holds. Now, we update the entry information of the third block with the empty set. Further, we add the edge $(3, 4)$ to the worklist. Even though the algorithm tells us to, we do not add the edge $(3, 6)$ to the worklist in this example calculation as it is the second element in the worklist. We actually only need to ensure that the edge is inside the worklist so that we propagate the information onwards. Since the worklist is a set, we do not have to add elements that are already in the worklist.

- Iteration 6: Now, we have to inspect the edge (3,4) for a second time. The check $\emptyset \subset \{x\}$ still holds. Therefore, we actually have new information to propagate onwards. Thus, we set $Analysis[4]$ to the empty set and add the edge (4,5).
- Iteration 7: Examining the edge (4,5) for the second time around the check $f_4(Analysis[4]) \not\subseteq Analysis[5]$ fails for the first time. $f_4(Analysis[4]) \not\subseteq Analysis[5]$ evaluates to $\emptyset \subset \emptyset$, which does not hold. Therefore, we do not reassign $Analysis[3]$ nor do we add any new edge to the worklist.
- Iteration 8: The first edge in the worklist is the edge (3,6). For that edge, the evaluation of $\emptyset \subset \{b, x, y\}$ holds. The next action we have to take is to reassign $Analysis[6]$. We reassign $Analysis[6]$ to the empty set. The sixth block has no successors. Hence, we cannot add any edge to the worklist.
- Iteration 9-12: In these four steps the check $f_l(Analysis[l]) \not\subseteq Analysis[l']$ fails. So, we do not do anything in these steps.
- Iteration 13 & Presenting the result: In the last step our worklist is empty, which is the end of the second step of the worklist algorithm. We go on to the last step of presenting the result. In the last column of the table, the information at the entry points for each block is depicted. For the information at the exit of these blocks, we would have to apply the *transfer* function. Applying them we get that every block but the second and the sixth has the empty set as the exit. The second and the sixth block have the set containing x as information as can be seen in Table 2.5. Figure 2.18 depicts the same information graphically.

	1	2	3	4	5	6
<i>in</i>	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
<i>out</i>	\emptyset	$\{x\}$	\emptyset	\emptyset	\emptyset	$\{x\}$

Table 2.5: Result of Applying the Worklist Algorithm to the `While Loop` Example from Subsection 2.2.2

So far, we presented how a data flow analysis works. Further, we have the means to automatically calculate the result of a data flow analysis. However, until now we only looked at a data flow analysis that tracks division-by-zero warnings. In the next chapter, we introduce our bidirectional predicate propagation. Using that predicate propagation, we are able to handle all run-time warnings. Further, we will also make use of a backwards analysis to be able to remove even more warnings. How we use the backward analysis and other aspects are discussed in depth in the next chapter.

3 Data Flow Analyses for Bidirectional Predicate Propagation

The goal of our bidirectional predicate propagation is to select as many safely removable warnings as possible. In this chapter, we present how we can use a predicate-based data flow analysis to achieve the goal. The data flow analysis from Chapter 2 is already close to the one we present in this chapter. However, the analysis from Chapter 2 only covers division-by-zero warnings. In this chapter, we present an analysis that is able to cover all run-time warnings. In the next section, we introduce the forward data flow analysis from our bidirectional predicate propagation.

3.1 Forward Data Flow Analysis to Track Predicates

Using the forward data flow analysis from our bidirectional predicate propagation, we want to gain knowledge about warnings that we can remove safely. Therefore, we want to make use of the annotations from Frama-C. More precisely, we are interested in the annotated assertions about warnings. In the last chapter, we restricted the warnings to be division-by-zero warnings. In our bidirectional predicate propagation, we do not want such a restriction. Therefore, we have to change the forward data flow analysis from the previous chapter in a way that it can handle all asserted warnings.

In Chapter 2, we could simply add the subject of the warning to the flow data. Without the division-by-zero warning restriction, we have to add the whole predicate to the flow data. Otherwise, having an unsigned-overflow warning about x and a division-by-zero warning about x , we would add x for each warning to the flow data in both cases making it impossible to distinguish them. Adding simply the variable that is the subject of the warning could lead to a case where we remove the division-by-zero warning because we had an unsigned-overflow warning before. By adding the complete predicate such a mix-up cannot occur. However, now we cannot simply remove a reassigned variable from the flow data. We have to remove all predicates that contain a reassigned variable.

Let us have a look at an example calculation to see how the analysis has to change.

3.1.1 Example

In this subsection, we take a look at a function with different kinds of warnings. In Listing 3.1, a C function is depicted. The C function has statements with different kinds of warnings. We pass the variable x to the function. In the first statement, we define an array with five elements. In the following, we assign two times the x th element of that array to h . The assignments can have an array-index-out-of-bound error in case x is not in the range $[0, 4]$. Therefore, Frama-C throws two index-bound warnings. Further, we reassign x at some point as well as divide once by x and another time by $a[x]$. The divisions result in two division-by-zero warnings.

```

1 void main(int x, int y)
2 {
3   int h;
4   const int a[] = {0, 1, 2,
5                   3, 4};
6   /*@ assert rte:
7     index_bound: 0 ≤ x;
8   */
9   h = a[x];
10  /*@ assert rte:
11    division_by_zero: y ≠
12    0; */
13  h = 1 / y;
14  /*@ assert rte:
15    division_by_zero: a[x]
16    ≠ 0; */
17  /*@ assert rte:
18    index_bound: 0 ≤ x;
19  */
20  /*@ assert rte:
21    index_bound: x < 5; */
22  h = 1/a[x];
23  x = y;
24  /*@ assert rte:
25    division_by_zero: y ≠
26    0; */
27  h = 1 / y;
28  return;
29 }

```

Listing 3.1: Example with Different Kinds of Warnings

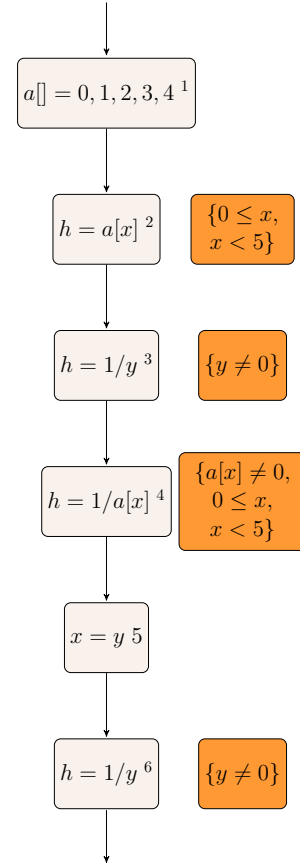


Figure 3.1: CFG of the Example with Different Kinds of Warnings

Figure 3.1 shows the CFG of the code in Listing 3.1. In addition to the CFG, we also have the predicates that are asserted in the warnings corresponding to the statements in Figure 3.1. In this example, we have multiple warnings attached to one statement for the first time. The fourth statement, for example, has three warnings attached to it, two of them are array-index-out-of-bounds warnings. One states that x has to be greater than or equal to zero. The other warning states that x has to be smaller than five. Otherwise, an array-index-out-of-bounds error occurs. In addition to these warnings, we also have a division-by-zero warning stating that $a[x] \neq 0$ has to hold.

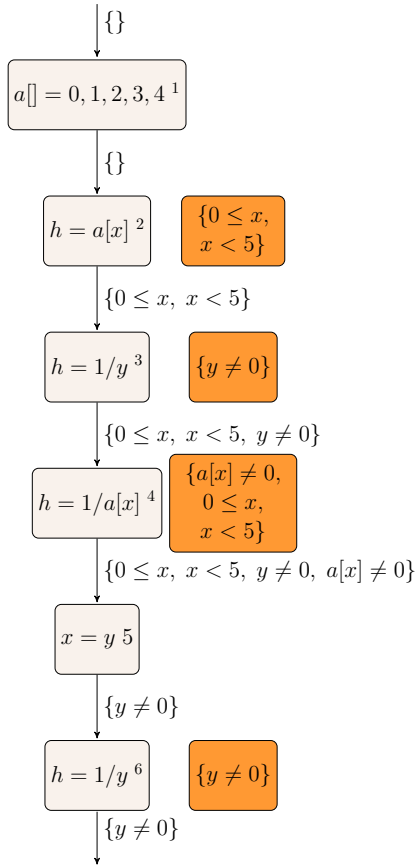


Figure 3.2: Application of the Forward Data Flow Analysis to the Example with Different Kinds of Warnings

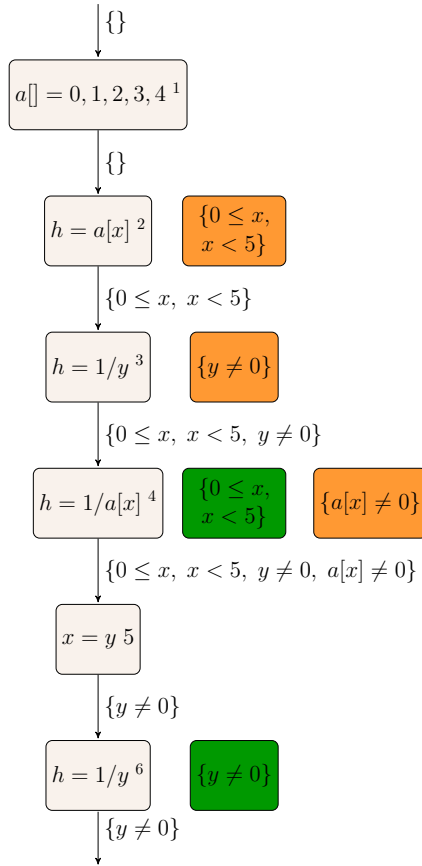


Figure 3.3: Final Result of the Example with Different Kinds of Warnings

Figure 3.2 shows the flow data we gain by applying the analysis to the example. Let us have a look at how these flow data are calculated:

- Initialisation: We start with the empty flow data.
- Block 1: We have no warnings corresponding to the first block. Therefore, we cannot add any predicates to our flow data. We leave the block with the empty flow data.
- Block 2: We have two warnings corresponding to this block. The two warnings contain the predicates $0 \leq x$ and $x < 5$, which we add to our flow data. Further, we reassign h . h is not the subject of any predicates. Thus, we do not remove any predicates from the flow data. Exiting the second block, we have the predicates $0 \leq x$ and $x < 5$ in the flow data.

- Block 3: We add the predicate $y \neq 0$ to our flow data. Since we reassign \mathbf{h} we do not remove any predicates from the flow data. Leaving the third block, our flow data set is $\{0 \leq x, x < 5, y \neq 0\}$.
- Block 4: In this block, we generate the predicates $0 \leq x$, $x < 5$, and $a[x] \neq 0$ and kill nothing. Hence, we add the predicate $a[x] \neq 0$ to our flow data in this block.
- Block 5: Here, we remove all predicates that contain the variable \mathbf{x} . The predicates $0 \leq x$, $x < 5$, and $a[x] \neq 0$ contain the variable \mathbf{x} . After we remove these predicates from the flow data, only the predicate $y \neq 0$ is left in it.
- Block 6: In the last block, we add the predicate $y \neq 0$ to the flow data and remove nothing, which does not change the flow data.

After the application of the data flow analysis, we can check if we can remove any warnings. Therefore, we go through the result of the application and check if the predicates of the warnings are already in the incoming flow data.

- Block 1: The first block has no warnings that we could remove.
- Block 2: Even though we have two warnings for the second block, the incoming flow data is empty. Thus, we cannot remove any warnings in this block either.
- Block 3: The incoming edge to the third block has two predicates in its flow data. Now, we have to check if we have a warning with these predicates in the block. Unfortunately, we do not have any warning with either $0 \leq x$ or $x < 5$ as predicate. So, we cannot remove any warnings in this block.
- Block 4: We have three warnings with the predicates $a[x] \neq 0$, $0 \leq x$, and $x < 5$ for the fourth block. Two of these predicates are also in the incoming flow data, namely $0 \leq x$ and $x < 5$. Therefore, we can remove the warnings with the predicates $0 \leq x$ and $x < 5$.
- Block 5: We have no warnings corresponding to this block that we could remove.
- Block 6: We have one warning corresponding to the last block. The predicate of that warning is in the incoming flow data. Since our check holds, we can remove the warning.

In this example, we could remove three out of seven warnings as depicted in Figure 3.3. However, we still need to define the analysis formally. We do that in the next subsection.

3.1.2 Formal Definition

To formally define the forward analysis, we first need to have a look at how the lattice for such an analysis could look like. In the second chapter, we went through the program and looked for all the variables in it. From the variables, we took the power set and

constructed the lattice with the is-superset-of relation. If we do the same here, we would have to generate all possible predicates for the variables. Therefore, we would have to generate predicates like $x > 0$, $x > 1$, $x > 2$, $x > 3$, $x > 4$, and so on for all variables. That would make the lattice infinite. Having a lattice of infinite height, it can be the case that we have to apply our *confluence* operator infinite times before we reach a fixed point.

However, we already know all possible predicates that we can encounter before we apply the analysis, as a Frama-C plug-in has to annotate the C code with the warnings. Going through the code and collecting all predicates of these warnings provides us with all predicates that we can encounter. If we now take the power set of these predicates with the is-superset-of relation, we get a finite lattice, which we can use. Knowing exactly what predicates we can encounter beforehand has another advantage. We can implement the analysis as a bit-vector analysis. The interesting part of a bit-vector analysis is that the *gen* and *kill* set have to be constant for every block. In our analyses the *gen* and *kill* set are independent from the *in* sets. Since we have all the predicates we can encounter, we can use these by looking at the statements to define the *gen* and *kill* sets before we actually start the analysis. Due to the fact that the *gen* and *kill* sets are constant, the application of the *transfer* function is considered to be fast since it is idempotent, which means that $f(f(x)) \sqsubseteq f(x) \sqcup x$ holds [14].

$$\begin{aligned}
In_n &= \begin{cases} \emptyset & n \text{ is the entry block} \\ \bigcap_{p \in \text{predecessors}(n)} Out_p & \text{otherwise} \end{cases} \\
Out_n &= (In_n \cup Gen_n) \setminus Kill_n \\
Gen_n &= \{p \mid p \text{ is the predicate in a warning in } n\} \\
Kill_n &= \begin{cases} killInfo(e) & e \text{ is reassigned in block } n \\ \emptyset & \text{no variable is reassigned in } n \end{cases} \\
killInfo(e) &= \{p \in \text{Predicates}_* \mid e \text{ is a variable in the predicate } p\}
\end{aligned}$$

Figure 3.4: Data Flow Equations of our Forward Analysis

In such a bit-vector analysis, we have a vector with all the predicates we can encounter with a 1 for a predicate that is in the flow data and a 0 if it is not included. Having a bit-vector analysis, we can also take the bitwise **and** as the *confluence* operator for a *must* analysis. For a *may* analysis the *confluence* operator would be the bitwise **or**. Further, every bit-vector analysis is an instance of a monotone framework. Therefore,

we can use the worklist algorithm from Section 2.4 to calculate the result.

Figure 3.4 shows the data flow equations for the forward analysis. The first equation, in which we define the information at the entry of a block, is the same as in Subsection 2.2.4. The *transfer* function is the same. However, the definition for the *gen* and *kill* sets are changed. The *gen* set has one small, but important change. In Subsection 2.2.4, we took the variables that are the subjects of division-by-zero warnings for the given block. Here, we take the predicates of all warnings corresponding to the given block.

In the equation for the *kill* set, we first check if we reassign any variable. If we do not reassign any variable, we return the empty set. Otherwise, we pass the reassigned variable on to the *killInfo* equation. In that equation, we check if that variable is contained in any predicate from our analysis. The set Predicates_* consists of all the predicates that we can encounter for the given program. Therefore, our *kill* set contains all the predicates that we can encounter which contain the reassigned variable. Having the formal definition, we can apply it.

Application of the Data Flow Equations

Let us apply the data flow equations to the CFG from Figure 3.1. The set Predicates_* is $\{0 \leq x, x < 5, y \neq 0, a[x] \neq 0\}$. Table 3.1 shows the result of the application of the data flow equations.

Block	1	2	3	4	5	6
<i>In</i>	\emptyset	\emptyset	$\{0 \leq x, x < 5\}$	$\{0 \leq x, x < 5, y \neq 0\}$	$\{0 \leq x, x < 5, y \neq 0, a[x] \neq 0\}$	$\{y \neq 0\}$
<i>Warning</i>	\emptyset	$\{0 \leq x, x < 5\}$	$\{y \neq 0\}$	$\{0 \leq x, x < 5, a[x] \neq 0\}$	\emptyset	$\{y \neq 0\}$
<i>Gen</i>	\emptyset	$\{0 \leq x, x < 5\}$	$\{y \neq 0\}$	$\{0 \leq x, x < 5, a[x] \neq 0\}$	\emptyset	$\{y \neq 0\}$
<i>Kill</i>	$\{a[x] \neq 0\}$	\emptyset	\emptyset	\emptyset	$\{0 \leq x, x < 5, a[x] \neq 0\}$	\emptyset
<i>Out</i>	\emptyset	$\{0 \leq x, x < 5\}$	$\{0 \leq x, x < 5, y \neq 0\}$	$\{0 \leq x, x < 5, y \neq 0, a[x] \neq 0\}$	$\{y \neq 0\}$	$\{y \neq 0\}$
Warnings to be removed	\emptyset	\emptyset	\emptyset	$\{0 \neq x, x < 5\}$	\emptyset	$\{y \neq 0\}$

Table 3.1: Result of Applying the Data Flow Equations to Different Kinds of Warnings
Example from Subsection 3.1.1

- Block 1: The equation for the *in* set defines that the set is empty. Further, the *gen* set is empty. The *kill* set contains all the predicates from Predicates_* containing the variable *a*, which is $a[x] \neq 0$. Thus, the *out* set is empty.
- Block 2: The *in* set is defined by the *out* set from the first block. The *gen* set contains $0 \leq x$ and $x < 5$. The reassigned variable *h* is not contained in any of the predicates of the Predicates_* set. Therefore, our *kill* set is empty.

- Block 3: All the remaining blocks have also only one predecessors, which *out* set defines the current *in set*. Here, the *gen* set contains the predicate $y \neq 0$ and the kill set is empty. Therefore, the *out* set is $\{0 \leq x, x < 5, y \neq 0\}$.
- Block 4: Here, we generate the predicates $0 \leq x, x < 5$, and $a[x] \neq 0$, and kill nothing.
- Block 5: In this block, we do not generate any predicates but remove all the predicates that contain the variable x . Hence, our *kill* set is $\{0 \leq x, x < 5, a[x] \neq 0\}$, which results in the *out* set containing only $y \neq 0$
- Block 6: Now, we generate $y \neq 0$ and kill nothing, which does not change the flow data.

Having the formal definition for the forward analysis, we still need to define how we can use the result of the application of the data flow analysis to remove warnings.

3.2 Warning Removal

Figure 3.5 shows the algorithm to select warnings that we can remove.

```
for all Blocks do  
  for all p : Predicates from the warnings of the current block do  
    if p is implied by the flow data from the in set of the current block then  
      Remove warning with predicate p in current block  
    end if  
  end for  
end for
```

Figure 3.5: Algorithm to Select Warnings We can Remove

We go through all the blocks of the CFG. For each block, we go through all predicates of the warnings in the current block. For each predicate, we check if it is implied by some predicates from the *in* set of the current block. If it is implied, we remove the warning that corresponds to the predicate. Otherwise, we do not do anything with that predicate and go on with the remaining predicates of the current block. After we went through all blocks and predicates, we know all the warnings we can remove.

From our bidirectional predicate propagation, we presented so far the forward analysis. However, the analysis is bidirectional. Therefore, we present the backward analysis of our bidirectional predicate propagation in the next section.

3.3 Backward Data Flow Analysis to Track Predicates

The goal of our bidirectional predicate propagation is to remove as many warnings as possible. However, how can a backward analysis help us to remove even more warnings?

To answer that question, let us first have a look at a little example. Listing 3.2 depicts the C code of that example with all its warnings. The example basically just consists of an `if then else` block. In both the `then` as well as the `else` block, we have a division-by-zero warning. We can see the CFG of that example in Figure 3.6.

```

1 void main(int x, int y) {
2   int h;
3   if (x != 0)
4     /*@ assert Value:
       division-by-zero: x
       != 0; */
5     h = 1/x;
6   else
7     /*@ assert Value:
       division-by-zero: x
       != 0; */
8     h = 1/x;
9 }

```

Listing 3.2: Backward Analysis Example

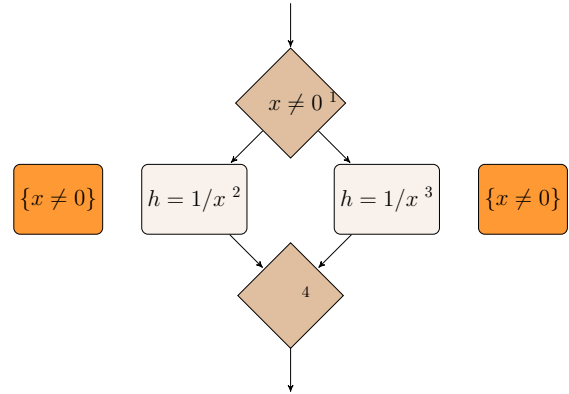


Figure 3.6: CFG of Backwards Analysis Example

Figure 3.7 shows the result of the forward analysis applied to the CFG from Figure 3.6. The blocks 2 and 3 have a corresponding warning, so we generate the predicate $x \neq 0$ in these blocks. After these blocks, we combine the branches again. The predicate $x \neq 0$ is in both branches. Therefore, the predicate is still in the flow data after the combination. However, we do not have any warnings with the predicate $x \neq 0$ after the combination. Thus, we cannot remove any warnings as depicted in Figure 3.8.

As mentioned before, a backward analysis can be used to propagate a predicate as far as possible upwards in the CFG. That can help us here. We only assert that $x \neq 0$ has to hold when we see the warning. However, x has to be not zero since the last reassignment. Further, the last reassignment is before the `if` condition. Hence, we could propagate the predicate and thereby the warning upwards in the CFG and place it before the `if` condition. Doing so, we would be able to remove more warnings.

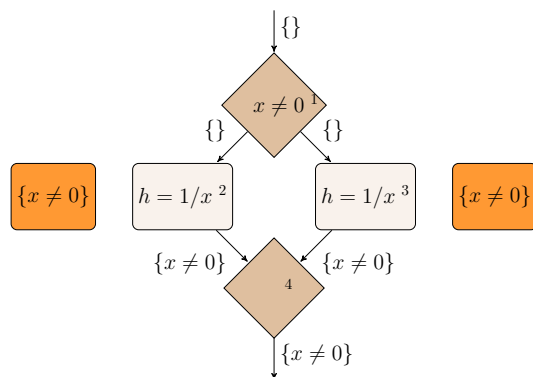


Figure 3.7: CFG with Complete Flow Data

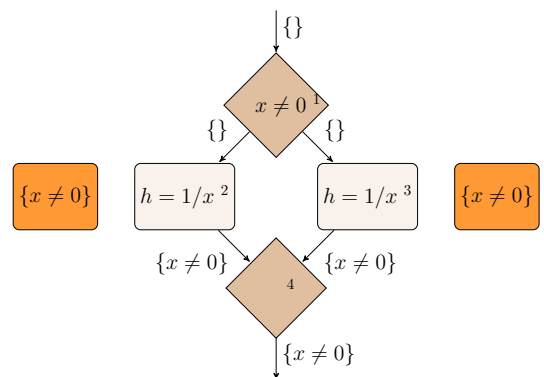


Figure 3.8: CFG with Removed Warnings

3.3.1 Example

Now, we apply the backwards analysis of our bidirectional predicate propagation to the CFG of Figure 3.6. Figure 3.9 shows the result of applying the backwards analysis to the CFG.

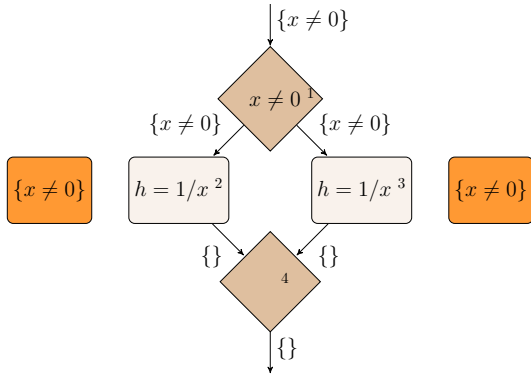


Figure 3.9: Backwards Analysis Applied to CFG from Figure 3.6

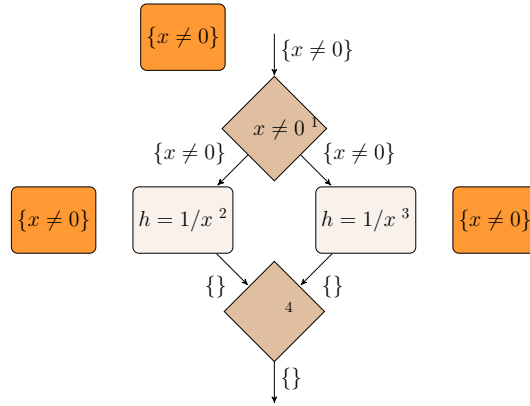


Figure 3.10: CFG with Inserted Warnings

The CFG only has one exit block, the fourth block.

- Block 4: We assign our initial value, the empty set, to the exit of the fourth block. In the fourth block, we do not have any corresponding warning nor do we reassign any variables. So, we simply pass the information from the exit to the entry of the fourth block.
- Block 3: In the third block, we generate the predicate $x \neq 0$. Further, we reassign h but h is not in any of the predicate from our Predicates_* set. Hence, we only add the predicate $x \neq 0$ and do not remove anything resulting in the *in* set containing only $x \neq 0$.
- Block 2: The second block is the very same as the third block. Therefore, the *in* set is $\{x \neq 0\}$.
- Block 1: The *out* set is defined by the combination of the *in* sets from the second and third block. Taking the intersection of $\{x \neq 0\}$ and $\{x \neq 0\}$ returns $\{x \neq 0\}$ for the *out* set. In the block itself, we do not reassign anything nor do we have a corresponding warning, resulting in the *in* set being the *out* set.

Next, we can use the result of the backward analysis to insert a new warning. The result of the analysis shows that the predicate $x \neq 0$ has to hold even before the `if` condition. Therefore, we can insert a new warning before the `if` condition.

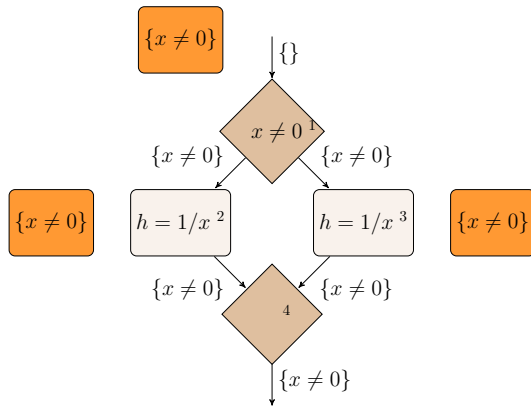


Figure 3.11: Forwards Analysis Applied to the Result of the Backwards Analysis Applied to CFG from Figure 3.6

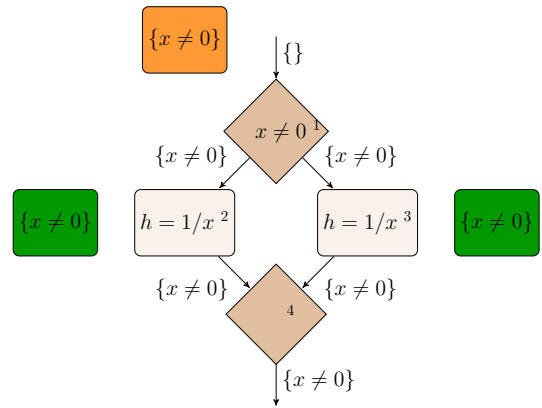


Figure 3.12: CFG with Removed Warnings

After we inserted the warning we can apply our forward analysis to the CFG with the inserted warning from Figure 3.10. The main difference compared to the earlier application of the forward analysis is that we have a warning corresponding to the `if` condition. This warning results in the difference that the incoming edge to the second and third block has the predicate $x \neq 0$ attached to it. The rest stays the same.

Next, we can go on and try to remove warnings. With the additional warning inserted, we are able to remove two warnings as depicted in Figure 3.12. Now, we have the predicate $x \neq 0$ already in our flow data reaching the second and third block. Therefore, we can remove the corresponding warning. By adding one warning, we are able to remove two warnings.

The example shows how we can improve our result by applying the backwards analysis first and then the forward analysis. Since we can improve our result by first applying the backwards analysis, we apply a bidirectional predicate propagation. Unfortunately, the backwards analysis has also some drawbacks, which we discuss later in this section.

3.3.2 Definition

Figure 3.13 shows the formal definition of the backward analysis from our bidirectional predicate propagation. The equations are nearly the same as the ones from the forward analysis in Figure 3.4. Basically, we simply exchange the *in* and *out* set and take the successors instead of the predecessors. Further, we have possibly more than one exit point to which we assign the initial value.

Let us now apply the data flow equations to the example from this section. Table 3.2 shows the result of the application.

- Block 4: We set the *out* set to the initial value, the empty set. Since the *gen* and *kill* set are empty, we define the *out* set to be the *in* set.

$$\begin{aligned}
 Out_n &= \begin{cases} \emptyset & n \text{ is a exit block} \\ \bigcap_{p \in \text{successors}(n)} In_p & \text{otherwise} \end{cases} \\
 In_n &= (Out_n \cup Gen_n) \setminus Kill_n \\
 Gen_n &= \{p \mid p \text{ is the predicate in a warning in } n\} \\
 Kill_n &= \begin{cases} killInfo(e) & e \text{ is reassigned in block } n \\ \emptyset & \text{no variable is reassigned in } n \end{cases} \\
 killInfo(e) &= \{p \in \text{Predicates}_* \mid e \text{ is a variable in the predicate } p\}
 \end{aligned}$$

Figure 3.13: Data Flow Equations of our Backward Analysis

- Block 3: The *out* set is defined by the *in* set from the fourth block. The *gen* set contains the predicate $x \neq 0$ and the *kill* set is empty. Therefore, we set the *in* set to $(\emptyset \cup x \neq 0) \setminus \emptyset = x \neq 0$.
- Block 2: We do the very same calculations as we did in the third block. Therefore, the same result as in the third block hold for the second block.
- Block 1: The *out* set is defined by the combination of the successors $\{x \neq 0\} \cap \{x \neq 0\}$, which is $x \neq 0$. The *gen* and *kill* sets are empty, leaving us with the *in* set being equal to the *out* set.

Block	4	3	2	1
<i>Out</i>	\emptyset	\emptyset	\emptyset	$x \neq 0$
<i>Warning</i>	\emptyset	$x \neq 0$	$x \neq 0$	\emptyset
<i>Gen</i>	\emptyset	$x \neq 0$	$x \neq 0$	\emptyset
<i>Kill</i>	\emptyset	\emptyset	\emptyset	\emptyset
<i>In</i>	\emptyset	$x \neq 0$	$x \neq 0$	$x \neq 0$

Table 3.2: Result of Applying the Backward Data Flow Equations to the Example from Section 3.3

Knowing how the backward analysis can help us, we still need to have a look at an example where the backward analysis does not help us with our goal to reduce the overall number of warnings.

3.3.3 Drawbacks

Let us have a look at the `if then else` example from Subsection 2.2.1.

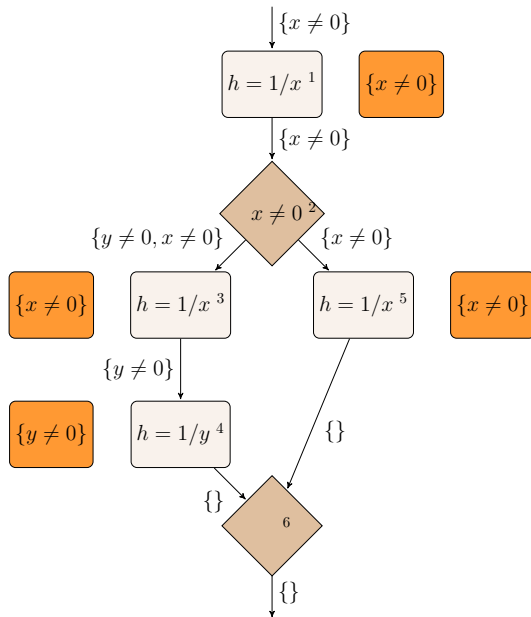


Figure 3.14: Backwards Analysis Applied to the CFG of If Then Else Example from Subsection 2.2.1

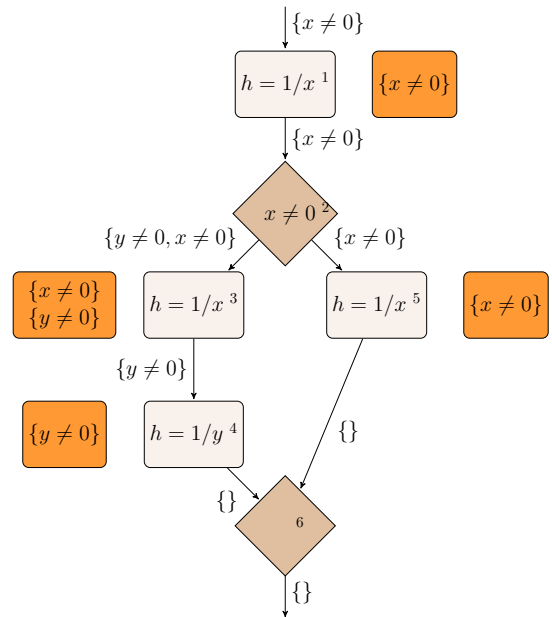


Figure 3.15: If Then Else Example from Subsection 2.2.1 with Inserted Warning

Figure 3.15 shows the result of the application of the backwards analysis to the example. The interesting part is that we generate the predicate $y \neq 0$ in the fourth block. In the second block, we combine the flow data from the third and fifth block. In the combination, the predicate $y \neq 0$ gets removed. Hence, $y \neq 0$ still has to hold in the third block but does not have to hold in the second block anymore. Further, in the third block there is no corresponding warning with the predicate $y \neq 0$. Due to these facts, we can insert a corresponding warning with the predicate $y \neq 0$ to the third block.

Inserting the warning, we unfortunately only propagated the warning one statement upwards. Even though we could propagate it a bit upwards and thereby closer to the last assignment of the variable, this propagation is not of big help. However, the propagation also does not increase the overall number of warnings. Applying the forward analysis, we would remove the warning corresponding to the fourth statement again, which otherwise we could not. Thus, we inserted one warning to be able to remove a warning.

The `if then else` example shows that the backward analysis is not always helpful but did not really have a drawback since it did not increase the number of warnings. Figure 3.16 shows the application of the backwards analysis to an example with drawbacks.

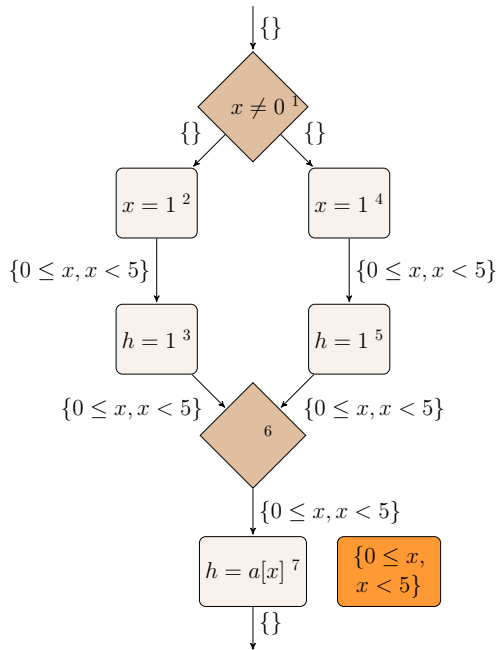


Figure 3.16: Example with a Drawback when Applying the Backward Analysis

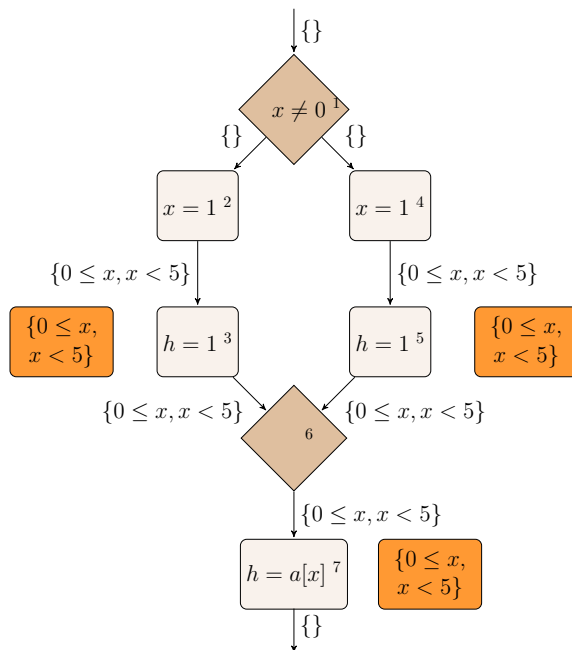


Figure 3.17: Drawback Example with Inserted Warning

In the seventh block, we have a corresponding warning with the predicates $0 \leq x$ and $x < 5$ that we add to our flow data. We propagate the flow data upwards in both branches of the `if then else` construct. In each branch, we encounter a block where we reassign x . By reassigning x , we remove the predicates from the flow data.

The result of the backward analysis states that the predicates $0 \leq x$ and $x < 5$ still hold at the third and fifth block. However, the predicates do not hold anymore at the second and fourth block. Therefore, we add corresponding warnings with the predicates to the third and fifth block as shown in Figure 3.17. So, we pushed the warning from the seventh block to the reassignments in the second and fourth block.

Even though we achieved our goal to push the warning to the last assignment, we pushed the warning into two branches. Hence, we had to add two warning based on one original warning. Applying the forward analysis this time, we can only remove the warning in the seventh block. Thus, by inserting two warnings, we are able to remove one warning. However, our goal is to reduce the number of warnings with our bidirectional predicate propagation. In this case, we increased the number of warnings by one.

To summarise, the backward analysis can be of great help to remove even more warnings, since the warnings get pushed closer to the assignment. Having the variable closer to the assignment might make it easier to determine if the predicate actually holds. However, by pushing the warning into branches it can be the case that we end up with more warnings than we originally started with.

3.4 Warning Insertion

In this section, we present an algorithm to select when we insert a new warning after we applied the backward analysis. In Figure 3.18, the algorithm for the warning selection is depicted.

```
for all Blocks do
  n := label of the current block
  for all p : Predicate of the in set from block n do
    if p is not an element of at least one in set from block n's predecessors then
      if p is not a warning in block n already then
        Attach warning with predicate p to block n
      end if
    end if
  end for
end for
```

Figure 3.18: Algorithm to Select Warnings We Insert

To select warnings to insert, we have to go through all the blocks from the CFG. We start with the exit blocks from the CFG to insert warnings. However, it does not matter with what block we start as long as we check all the blocks.

For the block we check, we take the *in* set from the result of the backward analysis. Next, we go through all the predicates from the *in* set. Having the predicate in the *in* set means that the predicate holds for this block. In case the predicate still holds in the current block but not anymore in the predecessor statement, we pushed it as far as possible upwards. Having it pushed as far as possible upwards, we attach a warning with the predicate to the current block.

In other words, we take the *in* set of the current block l' and compute the set difference from the *in* set from block l' with the *in* sets of block l' 's predecessors. Taking the set difference is the same as going through all the predicates of the first set and checking if they are not included in the second set. Further, in case block l' has more than one predecessor, we take the set difference to each of them individually. Having the difference between the sets, we know what predicates still hold in the current block but not anymore in a predecessor block.

Next, we check if we already have a warning with a predicate from the calculated difference set. In case there is already a warning with that predicate, we do not have to insert it. Otherwise, we insert a new warning with the predicate. We do that for every predicate from each difference set. Applying this algorithm to every block once, we get for every block the warnings that we can add since we cannot push the corresponding predicate any further.

Having the equations for the bidirectional predicate propagation and algorithms to use the information to remove as many warnings as possible, we still need to discuss the correctness. In the next section, we argue that the analyses and the algorithms applied afterwards produce a sound result, assuming that we were sound in the beginning.

3.5 Correctness

In this section, we have a look at the correctness of our bidirectional predicate propagation. Our propagation consists of two *must* analyses. To start, we have a look at examples showing what it means for a predicate to be in the flow data having a *must* analysis. Afterwards, we discuss for both the backward and the forward analysis under which circumstances a predicate is allowed to be in the flow data. Lastly, we still have to check that we use the result of the analyses correctly.

3.5.1 Must Analysis

Having a CFG without any branches, the result would be the same independent of having a *must* or a *may* analysis. Therefore, we have a look at a few CFGs with branches in the following to see why our analyses need to be *must* analyses.

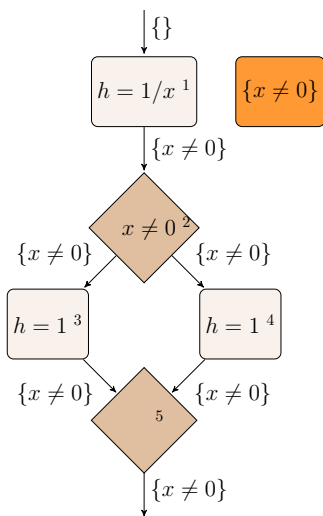


Figure 3.19: Example of *Must* Analysis without Reassignment

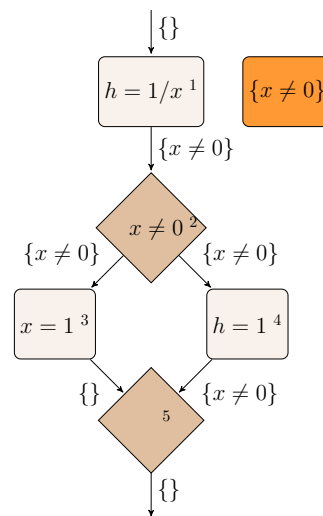


Figure 3.20: Example of *Must* Analysis with Reassignment

Figure 3.19 has one warning before the branching in the CFG. In neither of the branches, we reassign the variable of the warning. The predicate remains in both branches until we reach the combination. For a *must* analysis, a property has to hold on every path before the combination to still hold after the combination. Therefore, the predicate $x \neq 0$ still is in the flow data after we combine the branches. That means that we definitely had a warning that asserted $x \neq 0$ on every path reaching this point, without x being reassigned in between. Thus, the predicate still has to hold.

Figure 3.20 shows a CFG very similar to the one in Figure 3.19. The only difference is that we reassign x in one of the branches. Thereby, when we reach the point of combining the branches again, the predicate $x \neq 0$ is not in the flow data of both branches anymore. In one branch, we have empty flow data. In the other, we have the flow data with the predicate $x \neq 0$ in it. Combining these flow data with a *must* analysis results in taking

the set intersection of the two flow data. Taking the intersection, we end up with an empty flow data after the combination. Hence, we did not encounter a warning asserting $x \neq 0$ on every path without x being reassigned.

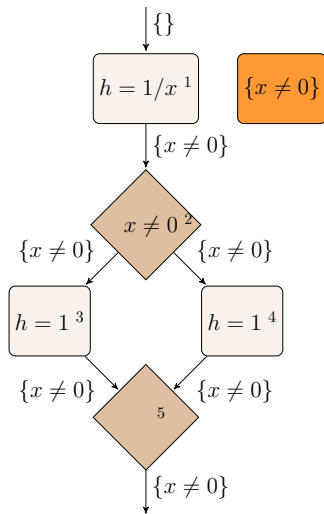


Figure 3.21: Example of *May* Analysis without Reassignment

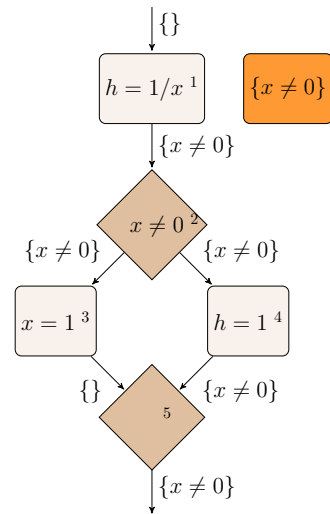


Figure 3.22: Example of *May* Analysis with Reassignment

Figure 3.21 shows the very same CFG with the very same flow data as Figure 3.19 does. However, in Figure 3.21 we applied our analysis as a *may* analysis. Taking the union of the sets $\{x \neq 0\}$ and $\{x \neq 0\}$ returns $\{x \neq 0\}$, which we also got by taking the intersection. Thus, for this example it does not make a difference whether we have a *may* or *must* analysis.

Figure 3.22 shows the same CFG as Figure 3.20 does. The only difference is that we apply a *may* analysis. Thereby, the predicate $x \neq 0$ is still in the flow data after the fifth statement. Taking the union of \emptyset and $\{x \neq 0\}$ we get $\{x \neq 0\}$ as the result. Thereby, the predicate is in the flow data even though it did not hold on every path reaching this point. Having a *may* analysis, we cannot be certain that a predicate has to hold reaching this point no matter what path we take. The predicate only holds for certain paths we take. For other paths it would be wrong to assume that the predicate holds.

Having a *must* analysis, we can be certain that if a predicate is in the flow data that it has to hold. The predicate has to hold since we encountered a warning with that predicate on every path reaching this point without reassigning any variable of the predicate between this point and encountering the warnings. Having two *must* analyses helps us to prove their correctness. However, we still have to have a look at both the forward and backward analysis as well as what we do with the result of the analyses.

3.5.2 Correctness of the Backward Analysis

Our backward analysis is sound exactly when the following holds: If a predicate has to be in the flow data, then the predicate has to hold already for a run-time error free

execution. In the analysis, we only add a predicate to our flow data if we encounter a corresponding warning. Further, if we reassign a variable, we remove all the predicates that we can encounter in the program that contain the reassigned variable. Having a *must* analysis, we can be certain that if a predicate is in the flow data it has to hold on every path for the warning's fault to not occur. Thus, the result of the analysis is sound since predicates are only in the flow data if they certainly prevent an error to occur.

The goal of our backwards analysis is to get to know since when a predicate has to hold. Even more, the result of the analysis tells us correctly what predicates have to hold for every block. So every time we encounter a predicate in the flow data, we are certain that the restriction already has to hold in this block.

Next, we have to use the result soundly. To use the result soundly means that we are only allowed to add a warning when we are certain that the restriction holds for the block. The easiest way to use the result in a sound way is to simply add a warning for every predicate in the flow data for every block. However, that would not be precise and not help reducing the number of warnings. Nonetheless, we could do it in such an imprecise fashion since the forward analysis would remove nearly all of them again.

The algorithm from Figure 3.18 aims at only adding warnings that will not be removed by the forward analysis again. In the algorithm, we only add a warning to a block if we have a corresponding predicate in the flow data. Hence, the result after applying the backward analysis and the insertion of the warnings is sound. Next, we have to check that we only insert warnings that will not be removed by the forward analysis instantly but insert a warning every time when we pushed the predicate as far as possible upwards.

In the algorithm, we take the set difference of the *in* set from the current block with the *in* sets from its predecessors individually. Each difference shows us which predicates still hold in the current block but not in the predecessor. Therefore, we pushed the predicate the furthest and can insert it.

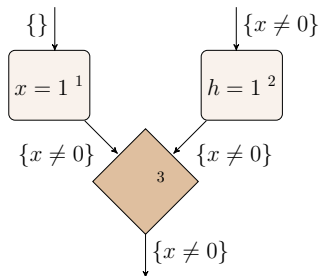


Figure 3.23: Example of Computing Difference Sets for all Predecessors in Insertion Algorithm

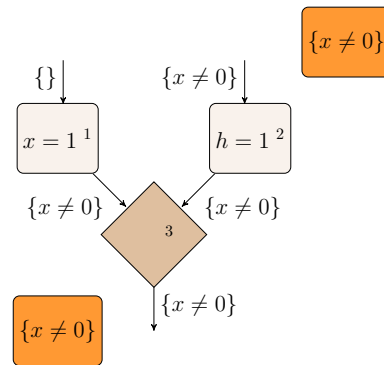


Figure 3.24: Example from Figure 3.23 with Inserted Warnings

Figure 3.23 depicts a part of a CFG. We use the example to show why we take the difference sets individually. In the example, we push the predicate $x \neq 0$ upwards. Reaching two branches, we push the flow data into both branches. In one branch, we

instantly reassign x . Therefore, the *in* sets of the predecessors of the third block contain the empty set for the first block and $\{x \neq 0\}$ for the second block.

Taking the difference individually, we get for the one difference that $x \neq 0$ is still true in the third block but not anymore in the first. The other difference set is empty. Thus, we attach a warning to the third block. In the other branch, we did not encounter an assignment yet and push the predicate further upwards as can be seen in Figure 3.24.

In case we did not take them individually but take the union of the predecessor sets, we would not attach a warning to the third block. However, we also cannot push it any further along the branch with the first block. Without the warning corresponding to the third block, we would not know that in the forward analysis, the predicate $x \neq 0$ holds after the combination in the third block. Further, we could not remove the warning whose predicate we inserted into the flow data and propagated upwards, even though we apply the backwards analysis before the forward analysis. To not lose any information that we calculated, we take the set difference individually. Therefore, applying our algorithm to insert warnings is sound since we only add warnings when we are certain the restriction has to hold. Further, we are more precise than just adding warnings everywhere our flow data allows it.

3.5.3 Correctness of the Forward Analysis

Our forward analysis is sound exactly when the following holds: If a predicate has to be in the flow data, then the predicate has to hold for a run-time error free execution. As for the backward analysis, we have a *must* analysis and we only insert a predicate when we have a corresponding warning. Further, if we reassign any variable, we remove all the predicates containing the variable that can be encountered in the program. Since the result of our forward analysis only contains predicates in the flow data that certainly prevent a run-time error to occur, the result of the analysis is sound.

The result of the forward analysis tells us in which blocks a predicate has to hold. In case a predicate is in the *in* set of a block, then the restriction has to hold in the block. Figure 3.5 shows the algorithm we apply to use the result of the forward analysis. The algorithm states that we remove a warning of a block iff the predicate of the warning is in the *in* set for that block. We can remove the warning since the restriction of the predicate has to hold for a run-time error free execution. By applying the algorithm, we only remove warnings that are covered by others. A warning is covered by another, in case the restriction of the warning was already inserted in the flow data by another warning and still holds. In case a predicate is in the flow data, we know that the restriction has to hold for a run-time error free execution. Therefore, we only remove warning which are safely removable.

The result of the forward analysis actually shows how the valuations of variables are restricted for each block for an error free execution. Thus, we cannot only use it to remove warnings but also to get a better knowledge about the possible valuations of variables. Knowing the theory behind the bidirectional predicate propagation and that its application is correct, we still have to have a look at the implementation. In the next chapter, we take a look at Frama-C and the plug-in developed for it.

4 Plug-In to Remove Redundant Warnings

We start by having a look at the Frama-C framework and the benefits it provides to us implementing our bidirectional predicate propagation. Afterwards, we present the plug-in and all the implemented features.

4.1 Frama-C

Frama-C is an open-source, platform-independent framework to statically analyse C code. The whole framework is written in OCaml. OCaml is a functional programming language. It has influenced languages such as F# and Scala.

The framework consists of different plug-ins. Each plug-in has a different goal when analysing the C code. The analyses are performed statically. An analysis is performed statically by going through the code without actually executing the corresponding program. The plug-ins can be used in combination to analyse C code [10]. To combine different plug-ins, and thereby, analyses, we need a way for them to communicate. For that, we need the representation the plug-ins actually work with. Frama-C works not directly on the C code but transforms it into an internal representation (IR), namely the C intermediate language (CIL) [21].

During the execution of the analysis, the plug-in annotates the IR with what it concludes analysing the code. In our examples so far, that corresponds to the annotated warnings. Each of these annotations have a unique id to distinguish them. The plug-ins can see in the IR what other plug-ins annotated and work with that information. Thereby, one can call different plug-ins in combination. Further, Frama-C provides the means to register new plug-ins. A newly written plug-in can then be combined with already existing plug-ins.

In the listings so far, we saw two different plug-ins that annotated the code. The code in Listing 3.1 is annotated by the `rte` [16] plug-in. The `rte` plug-in is a plug-in to go through the code and annotate all possible run-time-errors. Listing 2.1 was annotated by Frama-C's value analysis, which is probably one of the most widely used plug-ins. The value analysis plug-in is a *correct* static analyser [7]. *Correct* means that if anytime during run time a fault can occur, it warns about it. The value analysis can be used to verify C code against provided specifications. In case one does not provide any specifications, the value analysis only annotates run-time errors. Unlike the `rte` plug-in, the value analysis does not only annotate run-time errors but also tries to prove if the assertion of the run-time error holds. In case it comes to the conclusion that a certain run-time error cannot occur, it also does not warn about it and thereby, does not annotate the code.

We use the annotations about run-time errors for our bidirectional predicate propagation. Having these annotations is only one reason why we picked Frama-C to implement our analysis in. The framework also facilitates implementing a data flow analysis. To that end, the framework provides a data flow analysis module [17]. The module includes

a submodule for a forward analysis and another one for a backward analysis. Using that module is of great help implementing data flow analyses. The module includes a worklist algorithm to compute the result of the analysis. Therefore, we only need to provide the definitions for the *transfer* function and *confluence* operator. We use the Frama-C version Fluorine-20130601 on both Linux and Mac.

In the next section, we have a look at how we can use the provided module to implement our bidirectional predicate propagation.

4.2 Plug-In

The plug-in that implements our bidirectional predicate propagation is split up into five modules. For every task in our analysis, we have one module. The first module provides the means to go through the CIL and collect all the annotations containing a warning with an assertion. The other four modules deal with the forward and backward analysis as well as the warning removal and insertion.

4.2.1 Warning Collection Module

Frama-C provides different means to collect annotations from the IR CIL. To that end, we use the `in-place visitor` module. With the `in-place visitor`, we can go through the complete CIL code of the program at hand. Going through the CIL code, we have a look at all the annotations. In case an annotation is a warning with an asserted predicate, we collect that warning. Thereby, we collect all the annotations that are of interest for our analysis. While we collect the annotations, we also save the predicate of the annotation and the statement that the annotation corresponds to. Having all the warnings collected, we can start the forward analysis by passing these warnings to it.

4.2.2 Forward Analysis Module

In the module for our forward analysis, we implement the data flow equations from Figure 3.4. To that end, we use the `Forwards Dataflow Analysis` module from the `Dataflow` module of Frama-C. Using the module has one big advantage, we only have to define a few functions, which correspond to the data flow equations. After we implement these functions, Frama-C does the rest for us and computes the result of applying the equations until a fixed point is reached.

In the `Forwards Dataflow Analysis` module, there are three functions of particular interest since in these three functions, most of the work is done. The first function is called `combinePredecessors`. As the function's name already suggests, in here we define how we combine predecessors. We use our *confluence* operator, namely the intersection, to combine predecessors. Besides the *confluence* operator, we also define if we calculated new information. In case the intersection returns the same result as in the iteration before, we state that we gained no new knowledge. Otherwise, we return the result set of the intersection. No new knowledge means the edges to the successors are not added

to the work list. In case we gained new knowledge, the edges to the successors are added to the work list. This function is crucial for the analysis to reach a fixed point.

The second important function is called `doInstr`. In a statement, an instruction assigns a variable with an expression like $x + y$. The expression could also be a function call. For each instruction, we check if the enclosing statement has corresponding annotations. We go through the warnings we retrieved before the forward analysis and collect all the warnings that correspond to the current statement. For these warnings, we add the predicates to the flow data. Further, we have a look at the variable on the left-hand side of the assignment. Since that variable is reassigned, we remove all predicates that contain the variable from the flow data as defined in the data flow equation. In case the instruction is a function call, we call the forward analysis on that function.

The last function we have a look at is called `doStmt`. In Frama-C, we can have annotations attached to every statement and not only to assignments. Therefore, we have a look at all the statements that are not instructions and add to the flow data the predicates, which correspond to warnings of these statements. The `doInstr` and `doStmt` functions combined make up our *transfer* function.

Having implemented all the needed functions, we can start the analysis. The result of the application of the analysis returns a mapping from statements to flow data. In the flow data, we have all the predicates that have to hold. Thus, by applying the forward analysis, we know for every statement what restrictions for which variables have to hold to be certain that the execution is error-free. The next module, which removes warnings, makes use of these restrictions.

4.2.3 Warning Removal Module

For the module to remove warnings, we need to implement the algorithm from Figure 3.5. From the warning collection module, we get a list with all the warnings, its predicates, and the statement that they correspond to. From the forward analysis, we get a mapping stating what predicates have to hold in what statements. Therefore, we have all the information we need to apply the algorithm.

In the module, we go through the result of the forward analysis, which includes every statement of the function. For every statement, we retrieve the warnings that correspond to the statement from the list containing all warnings. In case the predicate from the warning is in the flow data of that statement, we remove the warning with that predicate. By applying the module, we remove all the warnings that are covered by others and, therefore, not needed anymore.

Having the modules to retrieve warnings from the IR, to apply the forward analysis, and to remove warnings, we still need to have a look at the backward analysis and the warning insertion.

4.2.4 Backward Analysis Module

In the module for our backward analysis, we implement the data flow equations from Figure 2.25. The backward analysis uses, as the forward analysis does, the retrieved

warnings from the warning collection module. As for the forward analysis, Frama-C also offers a module for backward analyses. To use that module, we again have to implement a few functions.

For the backward analysis, we have four functions of interest. The difference to the forward analysis is that the functionality of the `combinePredecessors` function is split up into two functions, `combineStmtStartData` and `combineSuccessors`. The functions `doInstr` and `doStmt` still make up the *transfer* function and the implementation is the same as described for the forward analysis.

The function we take a closer look at first is `combineStmtStartData`. The function determines if we reached a fixed point and also applies the *confluence* operator. The function combines the flow data calculated for the statement in the last iteration with the newly generated flow data using the *confluence* operator. In case the intersection of the old flow data with the new returns the old flow data, we reached a fixed point. Otherwise, we return the intersection, which means that we have not yet reached a fixed point. For the function to work, we have to go through all the statements and assign it with the bottom element from our lattice before we start the analysis. When we calculate flow data for the first time for a statement, we assign it to the statement using the intersection. Taking the intersection of the bottom element with x returns x .

The second function, `combineSuccessors`, only applies the *confluence* operator. In case a block has two successors it takes the intersection of the flow data of these successors.

After we specified all the functions needed for the backwards analysis, we can tune the analysis. The output is again a list containing mappings from statements to flow data that have to hold as the result. Our application of that result is the insertion of warnings.

4.2.5 Warning Insertion Module

For the module to insert warnings, we implement the algorithm from Figure 3.18. We need the result of the backward analysis as well as the retrieved warnings as inputs. Going through the result of the backward analysis, we take the flow data of each statement and compare it with the flow data of each predecessor individually. We compare the flow data by taking the set difference. The result of the set difference tells us which warnings we have to add in this statement.

Next, we check if we already have a warning in that statement with the predicate we want to add. To check if we already have a warning, we use the warnings we retrieved. In case there already is a warning with that predicate, we do not add it another time. Otherwise, we add a warning with the predicate. Going through every statement, we add all warnings needed to annotate the information we gained when we applied the backward analysis by inserting a new annotation in the IR.

Having all the modules of which our plug-in consists, we still need to define the order in which they are called.

4.2.6 Combination of the Modules

To apply our bidirectional predicate propagation, using the `-remove-annots` argument to Frama-C, we first retrieve all the warnings of the IR using the warning collection module. Having all the warnings from the IR, we start the backward analysis. In case the backward analysis reached a fixed point for a function, the warning insertion module is called. For the forward analysis, we cannot simply use the warnings we retrieved earlier. The warning insertion module possibly inserted new warnings. Therefore, we again retrieve all the warnings from the IR. We pass these warnings to the forward analysis. The forward analysis in turn calls the warning removal module every time it reaches a fixed point for a function. Thereby, we get our bidirectional predicate propagation.

We showed earlier that by applying the backward analysis, we sometimes end up with overall more warnings than we started with. Thus, it might be the case that one does not want to perform the backward analysis. To give the user the choice between the bidirectional predicate propagation and the forward analysis, the user can call our plug-in with the option `-backwards` to include the backward analysis and the warning insertion. Without this option, using the `-remove-annots` argument to Frama-C, only the forward analysis and the warning removal are applied.

The goal of our bidirectional predicate propagation is to reduce the review time by lowering the number of overall warnings. To this end, the forward analysis calls the warning removal module when it reached a fixed point and the backward analysis calls the warning insertion module. However, the result of the forward and backward analysis can be used for more than reducing the number of warnings. The results provide restrictions of variables for statements. Hence, one could use the results in other analyses of Frama-C to improve the concretisation, as the valuations of variables are further restricted.

While implementing the plug-in, we encountered a few issues. Among them, pointers turn out to be particularly troublesome. In the next section, we have a look at what problems pointers can cause.

4.3 Problem with C Pointers

The flow data equations from Figure 3.4 and Figure 2.25 both state that we have to remove all predicates that contain a reassigned variable. Having an instruction like `h = 1;`, we know easily what variable is reassigned, namely the variable `h`. Hence, we have to remove all predicates that contain the variable `h`. However, we could also have an instruction like `*h = 1;`. A pointer reassignment makes it harder to determine what variables are reassigned. The pointer could actually point to every other variable in the program. Therefore, every time we reassign a pointer we empty the flow data completely since we cannot be certain which variables actually get reassigned.

The other option to handling pointers would be to determine to which variables a pointer may point. To gain this information, we would have to implement and run a points-to-analysis or depend on the value analysis. In case we had the result of a points-to-analysis, we would only have to remove all the predicates that contain any of the

variables the pointer points to.

Another pointer-related problem are function calls. In a function call, one can pass a pointer as an argument. Further, C has global variables and pointers. Therefore, one can actually change the valuation of a variable during a called function. Having the possibility that basically any variable got reassigned in a function, we also remove all flow data after a function call. Handling pointers in such a fashion results in the fact that our implementation stays sound. However, it also unfortunately makes it slightly more imprecise.

With the pointer problem in mind, let us have a look at what we handle in the implemented plug-in.

4.4 Restrictions in the Plug-In

In Table 1, we can see all the types a predicate can have in Frama-C. Some of them are pointer-related like `Pfreeable`, `Pinitialized`, and `Pallocable`. However, we do not handle pointers properly in our analysis. Therefore, we restrict the predicates we add to our flow data to not be pointer related. Further, many of these predicates are not useful for us. Hence, we restrict ourself to predicates of either type `Pre1` or `Pand`. A predicate of type `Pand`, which is a conjunction of two predicates, is recursively split it up and examined again for predicates of type `Pre1` or `Pand` until we end up only with predicates of type `Pre1`.

Relational predicates, `Pre1`, are predicates with two terms and a relation operator. A relational operator is one of the operators $\leq, <, \geq, >, =, \neq$. Restricting ourselves to these predicates, we are able to handle division-by-zero, array-index-out-of-bound, signed-overflow, and unsigned-overflow run-time warnings. Therefore, we are able to handle all run-time warnings but the memory/pointer-related warnings. There is another restriction we made in the plug-in. A programmer is allowed to write assembly code in a C program. However, we do not include assembly written in a C program in our analysis.

We did not only make restrictions in the plug-in to implement the data flow equations for the C programming language but we also included a few refinements.

4.5 Improvements to the Plug-In

Using predicates to track warnings plus the `Dataflow` module of Frama-C provides us with opportunities to remove even more warnings. Overall, we implemented four enhancements to the plug-in that all use the fact that we are tracking predicates. The first is to make the forward data flow analysis path-sensitive. For path sensitivity, we additionally consider the guarding conditions.

4.5.1 Path Sensitivity

The first refinement we present is to make the forward analysis path-sensitive. In C, some constructs, like an `if then else` block, have a guard. Based on the evaluation of

the guard, we either take the **then** branch or the **else** branch. Further, we actually can insert the predicate that has to hold to take a certain branch into the flow data for that branch.

To depict how path sensitivity helps, let us have a look at the example from Figure 3.6. We used that example to show how the backward analysis can help us in our bidirectional predicate propagation to remove even more warnings.

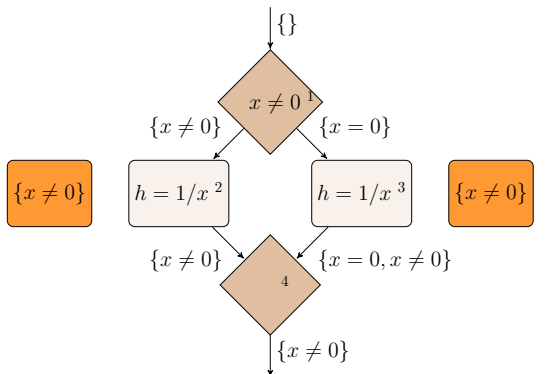


Figure 4.1: CFG with Predicates from Path Sensitivity Included in Flow Data

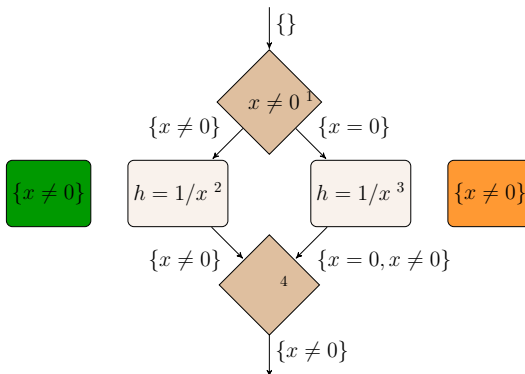


Figure 4.2: CFG with Removed Warnings

Figure 4.1 shows the flow data of applying the forward analysis with the path sensitivity. The interesting part is the first block. In the first block, we have a guard, namely $x \neq 0$. The guard does not throw a warning. However, we can use the guard in our flow data. In case the guard holds the program will take the **then** branch. Otherwise, the program will take the **else** branch. Therefore, we can add the guard to the flow data of the outgoing edge of the **then** branch. Further, we can negate the guard and add it to the flow data of the **else** branch. Adding these predicates, we now have predicates in the flow data for the outgoing edges from the first to the second and third block.

The predicate $x \neq 0$ is in the *in* set of the second block. The second block also has a warning containing that predicate. Thus, we can remove the warning. Figure 4.2 depicts the result with the removed warning.

Another interesting part is that the outgoing flow data of the third block contains $x = 0$ and $x \neq 0$. Having such mutually exclusive predicates, we know that this path is not feasible in an execution without run-time errors. For such a path, the user has to ensure that the run-time error actually cannot occur.

In our plug-in, the path sensitivity for the forward analysis is called by the option `-path`. In Frama-C, the function `doGuard` of the `Forwards Dataflow Analysis` module from the `Dataflow` module facilitates the implementation the path sensitivity. The function allows us to easily change the flow data of each branch individually when we reach a guard. Therefore, we add the guard to the **then** branch and add the negated guard to the **else** branch. However, we cannot exploit every guard. The predicates in

our flow data are all relational predicates. Hence, we only include a guard into the flow data in case it is a relational guard, like $x \neq 0$ or $x < 10$.

Tracking predicates to remove warnings has another advantage. Some predicates imply others. Thus, we take a closer look at how we can make use of the implications to further reduce the overall number of warnings in the next subsection.

4.5.2 Predicate Implication

In this subsection, we present another way to make use of the relational predicates. To demonstrate how relational predicates can imply each other, we have a look at three examples.

Listing 4.1 shows a function written in C annotated by Frama-C's `rte` plug-in. In the function, we define two arrays. One array has five elements while the other has six elements. Further, we access both arrays with the variable `x` as index.

```
1  void main(int x) {
2      const int a[] = {0, 1, 2, 3, 4};
3      const int b[] = {0, 1, 2, 3, 4, 5};
4      int h;
5      /*@ assert rte: index_bound: 0 ≤ x; */
6      /*@ assert rte: index_bound: x < 5; */
7      h = a[x];
8      /*@ assert rte: index_bound: 0 ≤ x; */
9      /*@ assert rte: index_bound: x < 6; */
10     h = b[x];
11 }
```

Listing 4.1: Annotated C Code with Different Restrictions

Accessing the first array with `x` as index, Frama-C annotates the statement with the assertions $0 \leq x$ and $x < 5$. An array with five elements only permits the indexes from 0 to 4. Therefore, these restrictions are asserted. The second array, `b`, has six elements. Frama-C asserts that $0 \leq x$ and $x < 6$ have to hold for a run-time error free execution.

Applying the forward analysis, the predicates $0 \leq x$ and $x < 5$ are in the incoming flow data of the statement where we access the array `b`. A predicate in the flow data means that the restriction of the predicate have to hold at the current point in the program to guarantee a run-time error free execution. The statement from line 10 has two warnings attached to it. One warning has the predicate $0 \leq x$. We also have that predicate in the incoming flow data. Therefore, we can remove it. The other warning has the predicate $x < 6$. We do not have exactly that predicate in the incoming flow data. However, from the predicates of the incoming flow data we know that $x < 5$ has to hold. Since `x` has to be smaller than 5 for an error free execution, it will also be smaller than 6. Thus, we can remove the warning with the predicate $x < 6$ due to the predicate $x < 5$ in our incoming flow data.

```
1  void main(int x) {
```

```
2   const int a[] = {0, 1, 2, 3, 4};
3   const int b[] = {0, 1, 2, 3, 4, 5};
4   int h;
5   /*@ assert rte: index_bound: 0 ≤ x; */
6   /*@ assert rte: index_bound: x < 5; */
7   h = a[x];
8   /*@ assert rte: signed_overflow: x+1 ≤ 2147483647; */
9   /*@ assert rte: index_bound: 0 ≤ (int)(x+1); */
10  /*@ assert rte: index_bound: (int)(x+1) < 6; */
11  h = b[x+1];
12 }
```

Listing 4.2: Annotated C Code with Different Restrictions

Listing 4.2 depicts a C function similar to the one in Listing 4.1. The only difference is that we access the array `b` with the index `x+1`. We can rewrite the predicates of the warnings that are attached to the statement of line 11. The predicate $x+1 \leq 2147483647$ can be rewritten as $x \leq 2147483647 - 1 = 2147483646$. Further, we can rewrite $0 \leq (int)(x+1)$ to $-1 \leq x$ and $(int)(x+1) < 6$ to $x < 5$. We rewrite these predicates with the assumption that no overflow will occur.

Applying the forward analysis to this annotated piece of code, we again have the predicates $0 \leq x$ and $x < 5$ in the incoming flow data of the statement where we access the array `b`. For this statement, we have a warning with the rewritten predicate $x < 5$. Thus, we can remove the warning. The predicate $0 \leq x$ from the incoming flow data also implies that $-1 \leq x$ has to hold in an error free execution. Lastly, we still have a warning with the predicate $x \leq 2147483646$. From the incoming flow data, we know that `x` has to be smaller than five. Therefore, $x \leq 2147483646$ will also hold and we can remove the warning with that predicate.

```
1   void main(int x) {
2     const int a[] = {0, 1, 2, 3, 4};
3     const int b[] = {0, 1, 2, 3, 4, 5};
4     int h;
5     /*@ assert rte: index_bound: 0 ≤ x; */
6     /*@ assert rte: index_bound: x < 6; */
7     h = b[x];
8     /*@ assert rte: index_bound: 0 ≤ x; */
9     /*@ assert rte: index_bound: x < 5; */
10    h = a[x];
11 }
```

Listing 4.3: Annotated C Code with Different Restrictions

In Listing 4.3, we exchanged the access of the arrays compared to Listing 4.1. Applying the forward analysis to this function, we have the predicates $0 \leq x$ and $x < 6$ in the flow data reaching the statement accessing the array `a`. In that statement, we have warnings with the predicates $0 \leq x$ and $x < 5$. The predicate $0 \leq x$ is already in the incoming

flow data. Hence, we remove the corresponding warning. For the predicate $x < 5$, we have no predicate in the incoming flow data that implies the predicate. In the incoming flow data, we assert $x < 6$. Thus, x could be 5. However, that would not be allowed for the predicate $x < 5$. Therefore, we cannot remove the warning with the predicate $x < 5$ since no predicate in our flow data implies that the assertion holds.

To enable this feature in our plug-in, one needs to supply the `-sat` option. In the plug-in, we implemented a linear constraint solver. The solver can handle all of the examples we presented here and many more. The predicates have to be of the form *var relOp const* or *var(+|-)const relOp const*. Since from our experience the variables are restricted in most cases by integers, the constants are restricted to integers. We also normalise the predicates so that it does not matter whether the variable is on the left- or right-hand side of the relational operator. The solver is sound because it only states that we can remove a warning in case at least one predicate from the incoming flow data implies the predicate we currently investigate. However, it can be the case that our solver falsely rejects a correct implication.

4.5.3 Splitting up Predicates

We also take advantage of predicate implications during the warning retrieval for the forward analysis. The predicate $x*y \neq 0$ implies that not only $x*y \neq 0$ has to hold from this point in the program onwards but also the predicates $x \neq 0$ and $y \neq 0$. These two additional predicates have to hold since otherwise $x*y \neq 0$ could not hold. Every time we encounter a predicate of the form *variable1 * variable2 \neq 0*, we want to ensure that we also generate the predicates *variable1 \neq 0* and *variable2 \neq 0* for that statement. Therefore, we add for each new predicate an item to our warning list containing the predicate and statement. Thereby, we also insert the newly generated predicates into our flow data during our forward analysis. Inserting these statement into the warning list during retrieval has the advantage that we still know all the predicates that can be in our flow data before we start the analysis. Having all the predicates beforehand, we can still implement our analysis as a bit-vector analysis. This refinement is always applied in our forward analysis.

The last refinement for our plug-in makes use of the fact that predicates restrict the valuation of the variables.

4.5.4 Transfer of Warnings

A predicate is always a restriction of the possible valuations of a variable. Assigning a variable that is restricted by a predicate to another variable, the restriction also holds for the newly assigned variable in a run-time error free execution.

```
1 void main (int x, int y) {
2     int h;
3     /*@ assert rte: division_by_zero: x  $\neq$  0; */
4     h = 1/x;
5     y = x;
```



```

6   x=1;
7   /*@ assert rte: division_by_zero: y ≠ 0; */
8   h = 1/y;
9   /*@ assert rte: division_by_zero: x ≠ 0; */
10  h = 1/x;
11  }

```

Listing 4.4: Annotated C Code with Restriction Transfer

Listing 4.4 contains annotated C code with three warnings. The first statement has an attached warning, asserting $x \neq 0$. In the next statement, we assign the valuation of x to y . The predicate $x \neq 0$ restricts the valuation of x . Therefore, after we assign the valuation of x to y , y should also not be allowed to be zero. To state that $y \neq 0$ has to hold after the assignment, we add the predicate to our flow data. In the following, we reassign x , which does not change that $y \neq 0$ has to hold, and we have two more division-by-zero warnings.

Let us have a look at the result of the forward analysis if we include the transfer of warnings when we assign a variable to another variable.

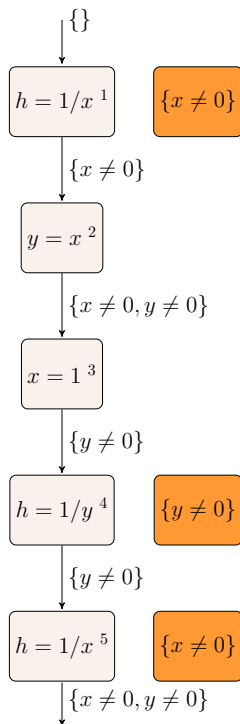


Figure 4.3: CFG of the Code from Listing 4.4

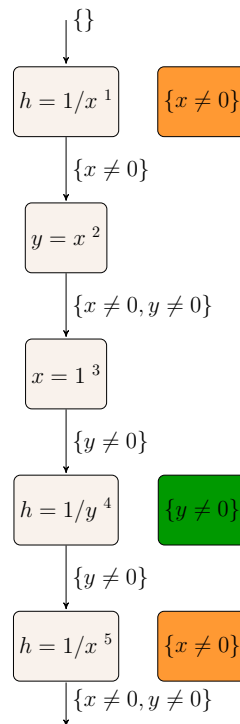


Figure 4.4: CFG of the Code from Listing 4.4

Figure 4.3 shows the result of the forward analysis with the transfer of the restriction for the code from Listing 4.4. The most interesting statement in this example is the

second statement. In that statement, we take all predicates from our flow data that contain x and replace the x with y and insert the newly generated predicates into the flow data.

In the third statement, we reassign x . Thereby, we have to remove all the predicates containing x . However, even though the predicate $y \neq 0$ was generated based on a warning with x as subject, we do not remove that predicate. We do not remove the predicate since the reassignment from the third statement does not change the possible valuations from x in the second statement and y is bounded to the possible valuations from x in the second statement. In the following, we are now able to remove the division-by-zero warning from the fourth statement but not from the fifth statement as depicted in Figure 4.4.

This refinement unfortunately has a drawback. In our bidirectional predicate propagation, we know all the predicates that we can encounter before we start the analysis. We know them since we could simply collect all the Frama-C asserted predicates of warnings and build a lattice of the power set from these predicates. Knowing all the predicates in before, gives us the advantage that we could define our analysis as a bit vector analysis. The *gen* and *kill* sets are constant for every block and we could calculate them before we started either of the analyses. Therefore, we could also provide the *transfer* function for every block beforehand. These are requirements for a bit vector analysis.

By transferring restrictions, not all predicates we can encounter are necessarily annotated by Frama-C. However, we only know all the predicates after we ran our analysis. We generate a new predicate every time we encounter an assignment of a variable that is contained in a predicate from the Frama-C annotations to another variable. The *gen* and *kill* set cannot be completely provided before we start the analysis. Hence, we cannot define it as a fast bit vector analysis anymore.

In our plug-in, the argument `-sub` enables the restriction transfer. We restrict the substitution to exactly one variable, e.g., $y = x$ but not $h = y * x$. Even if we have a predicate containing $y * x$, we would not transfer the restriction since it is not a single variable. Further, we only substitute the variable in predicates that contain only the variable we want to substitute on one side of the relation. An example of that would be that we have $x \neq 0$ in our flow data and assign $y = x$. Here would add the predicate $y \neq 0$ to our flow data. Having $x * y \neq 0$ in our flow data and assigning $h = x$, we would not substitute a variable in a predicate since x is not the only variable on the left-hand side of the relation. The restriction is made for reasons of simplicity only. Further, the variable is also only allowed to appear one time. Otherwise, we would have to add many predicates during some substitutions. The predicate $x * x * x < 10$ with the assignment $y = x$ would result in the addition of the predicates $y * x * x < 10$, $y * y * x < 10$, $y * x * y < 10$, $y * y * y < 10$, $x * y * x < 10$, $x * y * y < 10$, and $x * x * y < 10$ to the flow data. Thus, we would have to add for every n occurrences of a variable in a predicate $2^n - 1$ new predicates for $n \geq 2$.

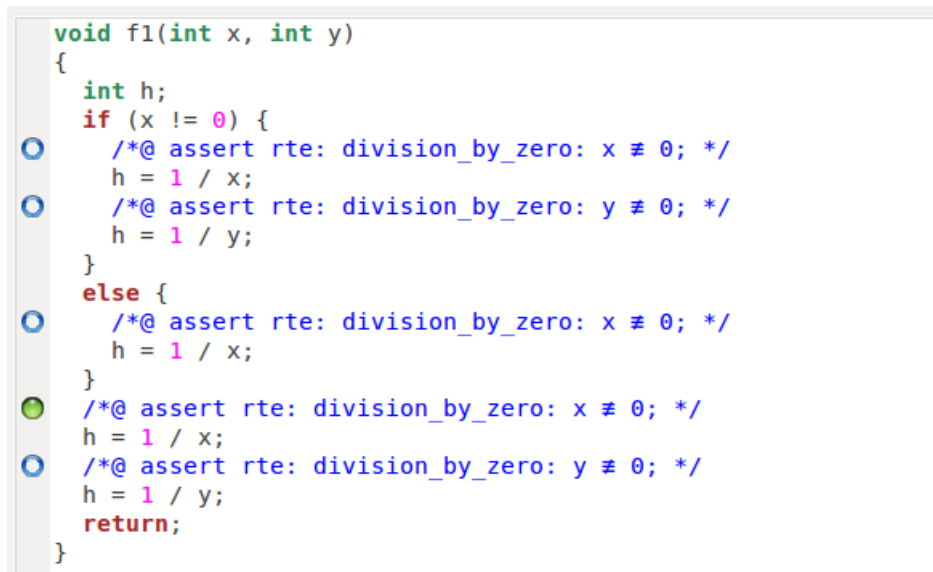
With the plug-in and all its established options, we still need to have a look at how it performs with the different options. In the next chapter, we take a look at the correctness of the implementation by testing it with small C programs. Further, we conduct a benchmarking on how many warnings we can remove using the different options.

5 Evaluation

In this chapter, we have a look at the implemented plug-in. We begin with the correctness of the implementation. To check the correctness of the implementation, we ran several tests on it. Afterwards, we take a look at the value analysis. The value analysis has a feature to remove warnings. Therefore, we compare our plug-in with theirs. Further, we discuss under which circumstances our tool can improve the result of the value analysis. Lastly, we present how well our plug-in can perform. To that end, we conduct a benchmarking. For that benchmark, we use over 300 C programs from the test suite provided by the National Institute of Standards and Technology (NIST) [4, 5].

5.1 Testing of the Implementation

So far, we discussed how the bidirectional predicate propagation works. Further, that applying our analysis only removes warnings that are covered by other warnings. What remains to show is that the plug-in correctly implements our analysis. For this purpose, we test that the plug-in returns the expected result. To test the plug-in, we checked every module independently. For every module, we tested different combinations of `if then else` blocks and `while` loops. For these combinations we used code where we expected to remove certain warnings as well code where we expected certain warnings to remain. All of these tests passed.



```
void f1(int x, int y)
{
  int h;
  if (x != 0) {
    /*@ assert rte: division_by_zero: x ≠ 0; */
    h = 1 / x;
    /*@ assert rte: division_by_zero: y ≠ 0; */
    h = 1 / y;
  }
  else {
    /*@ assert rte: division_by_zero: x ≠ 0; */
    h = 1 / x;
  }
  /*@ assert rte: division_by_zero: x ≠ 0; */
  h = 1 / x;
  /*@ assert rte: division_by_zero: y ≠ 0; */
  h = 1 / y;
  return;
}
```

The image shows a code editor window with a light gray background. The code is written in a monospaced font with syntax highlighting. On the left side of the code, there are five circular markers: four blue circles and one green circle. The blue circles are positioned next to the first, second, third, and fifth lines of the function body. The green circle is positioned next to the fourth line of the function body. The code defines a function `f1` that takes two integers `x` and `y` and returns an integer `h`. The function has an `if-else` structure. The `if` branch has two lines of code, each with an annotation. The `else` branch has one line of code with an annotation. After the `if-else` block, there are two more lines of code, each with an annotation, followed by a `return` statement.

Figure 5.1: A Test Case for the *Confluence* Operator

Figure 5.1 shows the result of the application of the forward analysis to the example from Subsection 2.2.1. We use the example to check that the *confluence* operator works

as expected in the forward analysis of our plug-in. The only difference to the example from Subsection 2.2.1 is that we added an additional warning to show that the predicate $1/y$ is not in the flow data after the combination of the branches.

To the left of the code, we can see five bubbles. The blue bubbles that are not filled mean that the status of the annotation is unknown. Therefore, we cannot assume with certainty that the assertion has to hold. The green filled bubble assures that the assertion holds. In case an assertion holds, we do not need to consider the corresponding warning anymore. Thereby, the warning is removed. Another bubble that occurs frequently in the GUI of Frama-C is orange filled. An orange filled bubble means that a verification attempt to the assertion has been made. However, during that attempt it could not be concluded with certainty that the assertion holds.

The `rte` plug-in annotates the code with the blue bubbles that are not filled. Our plug-in only changes the colour of the bubbles in case we could conclude that the assertion holds. The one assertion that has a green bubble is verified by our plug-in. We could also change all the others to orange since we tried to verify them but could not. But we only change the status of an annotation in case we could prove the assertion of the annotation.

In the example, we have in each branch an assertion stating that $x \neq 0$ has to hold for a run-time error free execution. Combining the two branches, the predicate is still in the flow data. In one branch, we have the predicate $y \neq 0$. However, the predicate is not in the other branch. Therefore, it does not survive the combination. Having only the predicate $x \neq 0$ in the flow data after the combination, we can conclude that the annotation with the predicate $x \neq 0$ has to hold. Further, since the predicate $y \neq 0$ was removed during the combination, we cannot conclude that the annotation with the predicate $y \neq 0$ has to hold. Thereby, the plug-in returned the result we expected.

We also did similar tests for the forward analysis with a `while` loop and the combination of both. Further, testing all modules like this, we fixed a few bugs. Besides testing every module with small C examples, where it is easy to calculate the correct result, we also checked the result of a few C programs from NIST which we also use for the benchmark. In these bigger C programs, we concluded that all the warnings that the plug-in removed are allowed to be removed. Having tested the program with small examples and C programs from a test suite, we are confident that the plug-in implements the data flow equations correctly.

To evaluate the plug-in further, we compare it to the value analysis and its warning removal feature in the next section.

5.2 Evaluation with the Value Analysis

As we mentioned before, the value analysis does not only annotate the IR with possible run-time errors but also tries to verify assertions that have to hold. To that end, the value analysis performs a data flow analysis to gain knowledge about the possible valuations of variables. To encode the possible valuations of variables they use different abstract domains, namely a k-set, an interval, and a congruence domain. The information of

possible valuations can be encoded as an interval like $[-10,10]$. In the verifying process, the value analysis annotates hardly any false positives in case the predicate is inside of the interval of the variable. However, it could be better in case we want to exclude a value like 0. Here the value analysis would need to track two intervals, e.g., the interval $[-10,0)$ and another interval $(0,-10]$. By simply calling the value analysis without any additional options, it does not split up the intervals like this but keeps the entire interval $[-10,10]$. In the following, we go through a few examples where our plug-in is of help and improves the result even if we call their warning removal plug-in.

5.2.1 Warning Removal Feature of the Value Analysis

The value analysis has the little known option `-remove-redundant-alarms`. The option tries to find redundant warnings and removes them from the CIL IR [9]. A warning is redundant in case its alarm is syntactically identical to a prior alarm and the variable of the warning is not reassigned between the alarms. The latter alarm is treated as redundant and removed from the IR. Therefore, the idea of the analysis is similar to our idea. However, the `-remove-redundant-alarms` option works only in combination with the value analysis. Our plug-in works in combination with every plug-in from Frama-C that annotates warnings.

Listing 5.1 shows the application of the value analysis with the option `-remove-redundant-alarms` to the code of Listing 2.1.

```
1  void main(int x, int y) {
2    int h;
3    /*@ assert Value: division_by_zero: x ≠ 0; */
4    h = 1 / x;
5    if (x != 0) {
6      h = 1 / x;
7      /*@ assert Value: division_by_zero: y ≠ 0; */
8      h = 1 / y;
9    }
10   else
11     /*@ assert Value: signed_overflow: 1/x ≤ 2147483647; */
12     /*@ assert Value: signed_overflow: -2147483648 ≤ 1/x;*/
13     h = 1 / x;
14   return;
15 }
```

Listing 5.1: `-remove-redundant-alarms` Applied to If Then Else Example

In Listing 2.1, we omitted the signed-overflow warnings in the else branch. The value analysis annotates them due to over-approximation errors. We omitted these warnings earlier since they are only false positives. Dividing by zero will result in a division-by-zero but not in a signed-over-flow run-time error. This imprecision emerged when analysing the value analysis for this work and has been fixed in the current Frama-C version.

The `-remove-redundant-alarms` option selects the very same warnings to be removed as we do. However, it removes the annotations while we set the status of the annotation to hold. In this example, both plug-ins come to the same conclusion. In the next subsection, we further compare our plug-in to the value analysis and its warning removal tool. In the process, we show different examples in which we can still remove warnings after we applied the warning removal tool of the value analysis as well as an example where the value analysis already concludes all the correct assertions but our backward analysis can still improve the result.

5.2.2 Bidirectional Predicate Propagation in Combination with the Value Analysis

Listing 5.2 shows a C function Pascal Cuoq, a developer of the value analysis plug-in, uses in his blog post “Minimizing the Number of Alarms Emitted by the Value Analysis” [9] as an example of a redundant warning that cannot easily be removed by only using the value analysis. A warning is redundant to a previous warning in case it cannot happen if a previous warning can be verified not to happen first.

```

1  int A;
2  int B;
3  void main(int x, int y)
4  {
5      /*@ assert Value: division-by-zero: (int)(x*y) ≠ 0; */
6      A = 100 / (x * y);
7      /*@ assert Value: division-by-zero: x ≠ 0; */
8      B = 333 % x;
9      return;
10 }
```

Listing 5.2: Example Containing Two Division-By-Zero Warnings

The value analysis annotates two division-by-zero warnings in the code of Listing 5.2. The first annotation states that $x*y \neq 0$ has to hold for a run-time error free execution. For the predicate $x*y \neq 0$ to hold, neither x nor y is allowed to be zero. Multiplying anything by zero returns zero, therefore, neither of the variables is allowed to be zero. Otherwise, the predicate $x*y \neq 0$ could not hold.

The other division-by-zero annotation states that the predicate $x \neq 0$ has to hold. In between the annotations, the variable x is not reassigned. Thus, we can conclude that the second warning is redundant and we can safely remove it. As we mentioned before, the value analysis, without additional options, tracks the possible valuations of a variable in a single interval. A single interval does not allow excluding values from it. Hence, the value analysis cannot conclude that the assertion of the second annotation has to hold due to the assertion of the first annotation. Further, the second annotation is not syntactically identical to the first annotation. The `-remove-redundant-alarms` option can only remove syntactically identical alarms, which makes it impossible for it to remove the second annotation.

During the forward analysis of our bidirectional predicate propagation plug-in, we also insert the predicates $var1 \neq 0$ and $var2 \neq 0$ into our flow data when we encounter a predicate $var1 * var2 \neq 0$. Having these predicates in our flow data, we can calculate that the warning corresponding to the second annotation can be removed safely. Therefore, our plug-in is able to remove the redundant warning from the example by Pascal Cuoq.

The code from Listing 5.3 is similar to the code from Listing 2.1. We left out the statement $h = 1/x$; before the `if then else` block and the statement $h = 1/y$; in the `then` block. Further, we inserted a new statement, $h = 1/x$;, after the `if then else` block.

```
1  void main(int x, int y)
2  {
3      int h;
4      if (y != 0)
5          /*@ assert Value: division_by_zero: x != 0; */
6          h = 1 / x;
7      else
8          /*@ assert Value: division_by_zero: x != 0; */
9          h = 1 / x;
10     /*@ assert Value: division_by_zero: x != 0; */
11     h = 1 / x;
12     return;
13 }
```

Listing 5.3: Example with one Redundant Warning

The Listing 5.3 shows the result of applying the value analysis with the `-remove-redundant-alarms` option. The built-in plug-in could not conclude that the warning after the `if then else` block is redundant to the warnings inside the `if then else` block. Our bidirectional predicate propagation computes that the predicate $x \neq 0$ has to hold after the `if then else` block. Thus, we can safely remove the warning after the `if then else` block. In case the value analysis could exclude the 0 from the interval of the possible valuations, it would actually not annotate the warning after the `if then else` block.

Another example where we can remove more warnings is the code of Listing 4.4. The problem is the same that the value analysis uses only one interval if one applies it without additional options. In that example, we could remove a warning since we transferred the predicates from one variable to another. The value analysis also transfers the possible valuations from one variable to another, however, it cannot exclude the zero in this example. In addition, applying the `-remove-redundant-alarms` option does not remove any warnings in this example. Hence, using our analysis, we are able to improve the concretisation of the value analysis, the possible valuation of variables.

Listing 5.4 shows C code with two array-index-out-of-bound warnings, one in each branch. After the `if then else` block, the value analysis restricts `x` to the interval $[0, 4]$. Thus, the value analysis does not annotate the third access of the array with additional warnings. The main difference is that now one interval is sufficient to represent the

implications of the warning. However, even in this example our bidirectional predicate propagation can reduce the overall number of warnings.

```

1  void f5(int x, int y)
2  {
3      const int a[] = {0, 1, 2, 3, 4};
4      int h;
5      if (y != 0)
6          /*@ assert Value: index_bound: 0 ≤ x; */
7          /*@ assert Value: index_bound: x < 5; */
8          h = a[x];
9      else
10         /*@ assert Value: index_bound: 0 ≤ x; */
11         /*@ assert Value: index_bound: x < 5; */
12         h = a[x];
13     h = a[x];
14     return;
15 }
```

Listing 5.4: Example with Array-Index-Out-Of-Bound Warning

Applying first our backward analysis and then forward analysis, we insert a new warning before the `if then else` block and can remove the warning inside the block. Here, we can still reduce the number of warnings from 2 to 1.

Comparing our bidirectional predicate propagation with the value analysis and its built-in plug-in, we come to the conclusion that there are still warnings that we can remove which the `-remove-redundant-alarms` option cannot remove. Further, our analysis can help the value analysis to improve the possible valuations of variables. In our analysis, we can have predicates that exclude certain values. The value analysis cannot exclude values from the interval of variables by splitting it up. Warnings that restrict the interval of a variable are handled well by the value analysis. Thereby, the value analysis already proves most of the assertions to be correct, leaving hardly any false positives. Yet, our backward analysis can reduce the overall number of warnings since the value analysis only applies a forward analysis. Lastly, Pascal Cuoq describes in his blog post [9] how the `-slevel` option of the value analysis can reduce the number of warnings. However, the `-slevel` option increases the time consumption from seconds to minutes. Our analysis, on the other hand, just takes a few seconds based on our experience.

In the next section, we run a benchmark to show the potential of the plug-in.

5.3 Benchmark

To conduct the benchmarking, we use two test suites, namely the “**IARPA STONESOUP Phase 1** - Null Pointer Dereference for C Version 1.0” [2, 4] and “**IARPA STONESOUP Phase 1** - Memory Corruption for C Version 1.0” [1, 4] test suite from

the Software Assurance Metrics And Tool Evaluation (SAMATE) project of NIST. As the names of the test suites suggest, they have a number of pointer operations in the programs. Having many pointer operations is not ideal for our analysis since we remove all our flow data in case a pointer is reassigned. However, it also shows how our analysis performs under far from ideal circumstances. Running the benchmark, we use the `rte` plug-in to annotate the warnings. The bidirectional predicate propagation plug-in has four options and we run the benchmark for all the combinations of these options. Therefore, we end up with 16 results per test suite.

In the following, we present the results of the test suites individually. For each test suite, we first discuss the results of our analysis using only the forward analysis and afterwards the results of the backward analysis included. Lastly, we summarise the insights of the benchmark.

5.3.1 IARPA STONESOUP Phase 1 - Null Pointer Dereference for C Version 1.0 Test Suite from NIST

We ran our benchmark on 113 of 115 programs from the test suite “IARPA STONESOUP Phase 1 - Null Pointer Dereference”. We did not get the remaining two programs to run in combination with Frama-C. Therefore, we left them out. Table 5.1 shows the result of the benchmark without the backward analysis.

Overall Warnings	Options							
	Warnings Removed Absolutely				Relatively Reduced			
1921	none		-sub		-path		-sat	
	778	40.50%	778	40.50%	849	44.20%	1008	52.47%
	-path -sub		-sat -sub		-path -sat		-path -sat -sub	
	850	44.25%	1008	52.47%	1128	58.72%	1129	58.77%

Table 5.1: Benchmark of the IARPA STONESOUP Phase 1 - Null Pointer Dereference for C Version 1.0 Test Suite from NIST without the Backward Analysis

In the 113 programs, we collected 1921 warnings that the `rte` plug-in annotated. In each cell of the table, we have the enabled options in the top. The number of warnings is in the lower left corner and the percentage of removed warnings is in the lower right corner. By only applying the forward analysis without further options we can remove 778 of the 1921 warnings, which is 40.5%. Further, we can see that the `-sat` option, our linear constrain solver, provides the most benefit from our three options. By enabling the `-sat` option, we can remove 230 warnings additionally. Thereby, we can remove 1008 warnings, which is 52.47%. Moreover, we can see that the `-sub` option, our predicate transfer, alone does not allow us to remove any additional warnings. Only in combination with the `-path` option, our path sensitivity, does the `-sub` option remove one warning more. This behaviour is the result of the rather strict restrictions we impose on the `-sub` option. It is seldom the case that we assign one variable to another and encounter the transferred predicate soon after. In the test suite, it is only the case when we generate

a predicate from a guard and transfer it to another variable. We can remove the most warnings, namely 1129, which is 55.77%, by applying all three additional options. Thus, we can remove 351 additional warnings by applying our three enhancements for the first test suite.

Figure 5.2 shows how many warnings we could remove per program as a percentage of the overall warnings in the program using only the forward analysis without additional options. The interesting part is that in case we can remove warnings, we are often above the average of 40.5%. However, we cannot remove any warnings in 30 out of the 113 programs. In Figure 5.3, all options are enabled. The combination of all options allows us to remove a few more warnings in most programs. Further, the total number of programs in which we cannot remove any warnings decreases to nine.

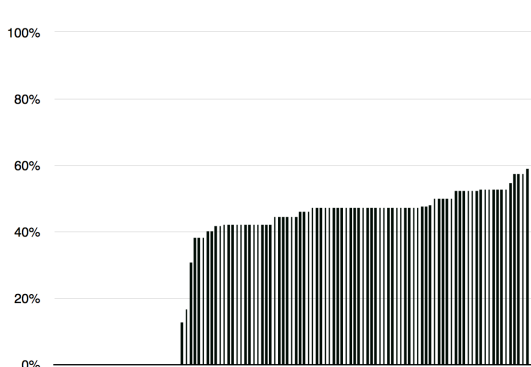


Figure 5.2: Percentage of Removed Warnings (Y-Axis) for each Program (X-Axis) using only the Forward Analysis

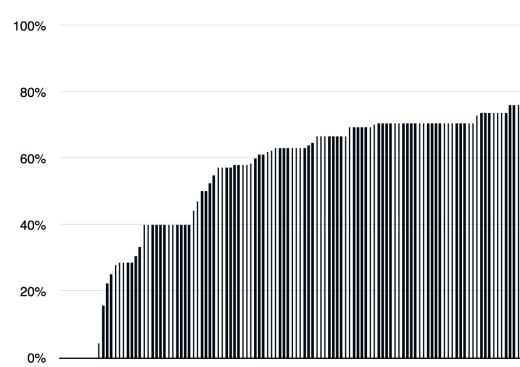


Figure 5.3: Percentage of Removed Warnings (Y-Axis) for each Program (X-Axis) using the Forward Analysis with `-path`, `-sat`, and `-sub`

Table 5.2 shows the results with the backward analysis included. The plug-in adds 107 warnings based on the results of the backward analysis. To compute the percentage of removed warnings, we take the warnings we could remove and subtract the 107 warnings added due to the backwards analysis and divide that result by the warnings annotated by the `rte` plug-in.

Unfortunately, the cleaned number of removed warnings is always seven warnings less than without the backward analysis. In the 113 programs, we never have the case that we have the same warning in two branches and push it upwards so that we can remove at least two warnings by inserting one warning. However, we actually push a few warnings in branches and insert these warning in the branches. Thereby, we insert at least two warnings to remove one warning. Therefore, the backward analysis does not improve our result but changes it to the worse in terms of total number of warnings.

Overall, the results we could achieve for the test suite are good. The plug-in can remove over 40% of the warnings by simply applying the forward analysis. In case we include the three options `-path`, `-sat`, and `-sub`, the plug-in removes another 18%.

Original Warnings	Warnings Added	Options					
		Warnings Removed		Removed - Added	Relatively Reduced		
1921	107	-backwards			-sub		
		878	771	40.14%	878	771	40.14%
		-path			-sat		
		949	842	43.83%	1108	1001	52.11%
-path -sub			-sat -sub				
950	843	43.88%	1108	1001	52.11%		
-path -sat			-path -sat -sub				
1228	1121	58.36%	1229	1122	58.41%		

Table 5.2: Benchmark of the **IARPA STONESOUP Phase 1** - Null Pointer Dereference for C Version 1.0 Test Suite from NIST with the Backward Analysis

Having to look at only 792 warnings instead of 1921 is already a huge improvement and reduces the review time drastically.

Let us go on with the second test suite.

5.3.2 IARPA STONESOUP Phase 1 - Memory Corruption for C Version 1.0 Test Suite from NIST

We ran our benchmark on 211 of 213 programs from the second test suite. We did not get the remaining two programs to run in combination with Frama-C. Therefore, we left them out. The result of the benchmark is again split up in the results without the backward analysis and with the backward analysis included. Table 5.3 shows the results containing only the forward analysis.

Overall Warnings	Options							
	Warnings Removed Absolutely				Relatively Reduced			
7211	none		-sub		-path		-sat	
	864	11.98%	864	11.89%	1148	15.92%	1334	18.50%
	-path -sub		-sat -sub		-path -sat		-path -sat -sub	
	1148	15.92%	1334	18.50%	1953	27.08%	1989	27.58%

Table 5.3: Benchmark of the **IARPA STONESOUP Phase 1** - Memory Corruption for C Version 1.0 Test Suite from NIST without the Backward Analysis

The results are not as impressive as the results of the first test suite. The plug-in with only the forward analysis can remove about 12% of the warnings. Figure 1, from the appendix, shows the chart of the percentage of the warnings we can remove for each program using only the forward analysis. As in the first test suite the **-sub** option does not improve the result. Further, the **-sat** option also improves the result the most. Applying the forward analysis with all three options, we can remove additional 16%, which is about the same percentage as for the first test suite. The percentage of the

warnings we can remove for each program with all the options enabled for the forward analysis can be seen in Figure 2 in the appendix.

The interesting part is that the `-sub` option only removes additional warnings in combination with `-path` and `-sat`. The reason for that is that we have to insert a few predicates into our flow data using the guards. After we transfer these predicates to the newly assigned variable, we still have to apply the linear constraint solver to prove that the fault cannot occur in a run-time error free execution. To do so, we need all three options enabled. Further, we can again reduce the number of programs where we cannot remove any warnings from 71 to 26.

Table 5.4 shows the result of the benchmark with the backward analysis. Unfortunately, for the second test suite the backwards analysis worsens the result even more. In four cases, we remove 135 warnings less, which also includes the case without any options. However, in this test suite, we have a few programs that contribute a lot to this reduction. Listing 1, from the appendix, shows a piece of code annotated by the `rte` plug-in. In this example, we take one warning and push it into 10 branches. Thereby, we insert 10 warnings from 1 original warning. Pushing one warning into so many branches has a big impact on the result of the backward analysis.

Original Warnings	Warnings Added	Options					
		Warnings Removed			Removed - Added		Relatively Reduced
7211	1804	-backwards			-sub		
		2533	729	10.11%	2533	729	10.11%
		-path			-sat		
		2817	1013	14.05%	2735	931	12.91%
		-path -sub			-sat -sub		
		2817	1013	14.05%	2735	931	12.91%
		-path -sat			-path -sat -sub		
		3631	1827	25.34%	3667	1863	25.84%

Table 5.4: Benchmark of the **IARPA STONESOUP Phase 1** - Memory Corruption for C Version 1.0 Test Suite from NIST with the Backward Analysis

Figure 5.4 shows how many warnings we could remove for each program. Using the backward analysis, we end up with more warnings than we started in a few cases. These are often programs where we could not remove more than one or two warnings using only the forward analysis. If we then push a warning into many branches, we end up with more warnings than we started with. The program to which the piece of code from Listing 1 belongs is one of the -42% bars.

The backward analysis performs the worst with the `-sat` options and the combination of `-sat` and `-sub` options. We cannot benefit from these options since we push many warnings not far up from the origin. Having two warnings like first the predicate $x < 4$ and then $x < 5$, the `-sat` option removes the second warning. The backward analysis pushes both of the warnings up to the last assignment of `x`, which is not necessarily in a branch. Thereby, we insert one warning to remove one warning. However, now

the warning that was removed before by the `-sat` option also gets removed due to the backward analysis and that still just counts as one removed warning. In the 213 programs, we observed a number of these cases.

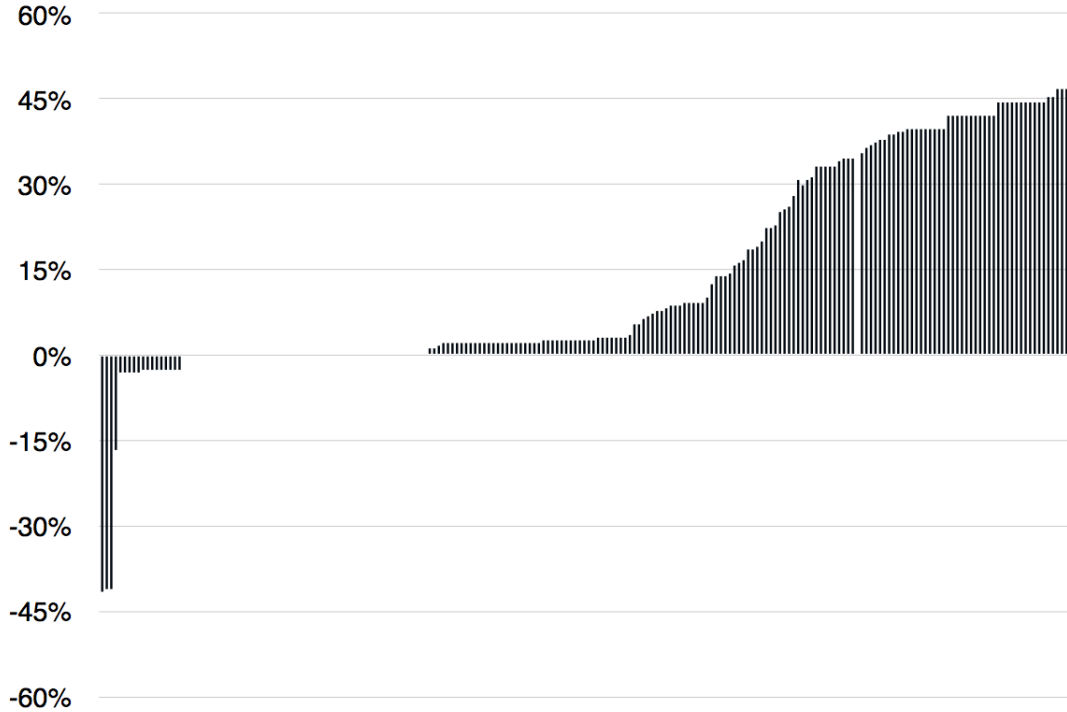


Figure 5.4: Percentage of Removed Warnings (Y-Axis) for each Program (X-Axis) using the Backward Analysis without any Additional Options

By applying the `-sat` option in combination with the `-path` option, we only remove 126 warnings less. Using these options, we can actually remove a few of the warnings the backward analysis inserted. Thus, the result slightly improves. Overall, the three enhancements, which we implemented, also improve the result of the benchmark using the second test suite a lot.

Let us now have a look at the insights we gained by the benchmark.

5.3.3 Insights of the Benchmark

The benchmark shows us that even under suboptimal conditions the bidirectional predicate propagation plug-in can achieve our goal to remove warnings and thereby, likely to reduce the review time. In the first test suite, we could remove nearly 60% of the warnings. Further, in both test suites our three additional options `-path`, `-sat`, and `-sub`, which enable the linear constrain solver, the path sensitivity, and the predicate transfer, could improve the result by nearly 20%. Thus, these enhancements pay off. However, applying our backwards analysis actually inserted more warnings than we could remove

using its result. To make the backward analysis more useful, we would have to ensure that we do not push a warning into branches and insert warnings there. To ensure this, one has to solve a graph problem to find the point exactly before the branches to insert only one warning. Thereby, in the worst case we would insert one warning to remove another warning. Therefore, we would most certainly not worsen the result of our analysis. Aside from the backward analysis, our plug-in already performs really well.

To conduct the benchmark, we used the `rte` plug-in to annotate the warnings. We also could have used the value analysis. However, the programs mostly contain warnings that restrict the variable's interval and hardly any warnings that exclude values. Hence, our analysis could hardly improve the result of the value analysis. To show the performance possibilities of our analysis, we therefore chose the `rte` plug-in to annotate the warnings. With the `rte` plug-in to annotate the warnings, the results look promising.

6 Related Work

We start our related work by having a look at the paper “Review Efforts Reduction by Partitioning of Static Analysis Warnings” by T. Muske, A. Baid, and T. Sanas [20]. Their paper presents a data flow analysis to reduce review efforts by removing warnings. It was also the starting point for this thesis. The main difference, however, is that they track expressions instead of predicates. They track the part of the expression that causes the warning. Tracking only the expressions, we still would need to track what kind of warning it was. Otherwise, we might mix up division-by-zero and array-index-out-of-bound warnings [15]. Therefore, we would have to construct something like the annotation of Frama-C ourselves if we wanted to implement the analysis tracking expressions. Further, using predicates allows us to implement refinements as described in Section 4.5. The ease of implementing the refinements was the crucial point why we decided to use the predicates of the annotations from Frama-C.

In their paper, they also have two analyses, one forward and one backward analysis. The result of each analysis is a set of redundant warnings. They combine the results of the analyses to find warnings that maximise the set of redundant warnings to remove as many as possible. We first apply the backward analysis and use the result to insert warnings. Only afterwards, we apply the forward analysis to remove the warnings again. By first applying the backward analysis, we push the warnings as far up as possible. Using the pushed warnings, we also try to increase the set of warnings that is covered by other warnings. In fact, it should not make a difference if we execute the forward and backward analysis in parallel and combine the results afterwards or if we first apply the backward and then the forward analysis and use that result. We should end up in both cases with the same number of warnings to be removed. Assuming that the number of warnings to be removed is the same, then our analysis performs as good as their proposed analysis. Further, with our refinements, which we can only do since we are tracking predicates, we are able to remove even more warnings. Therefore, we can perform even better than their approach can.

Another related paper is by R. Cytron et al., titled “An Efficient Method of Computing Static Single Assignment Form” [12]. In the paper, they define a dominance frontier. A variable x dominates y if x is defined on every path from the function entry to y . We could apply the definition to our warning removal. Here, we remove a warning if the predicate of the warning is in the incoming flow data. A predicate is in the incoming flow data only if it is asserted on every path reaching this point. Therefore, we could define our warning removal in terms of dominance. With the concept of dominance as a basis, we can also prove the correctness of the removal.

The paper “The Reduced Product of Abstract Domains and the Combination of Decision Procedures” by P. Cousot, R. Cousot, and L. Mauborgne [8] describes a way to use two analyses to improve the concretisation. We mentioned that we could use our analysis to improve the concretisation of the value analysis. In their paper, they describe how we could combine the results of two analyses using a direct product, which we could use

to improve the concretisation of the value analysis by using the result of our backward analysis and the predicates that exclude certain values of a variables valuation.

J. Fischer, R. Jhala, and R. Majumdar describe in their paper “Joining Dataflow with Predicates” [13] how to make a data flow analysis path-sensitive to make it more precise. In a way, we do the same with our `-path` option, which makes the analysis path-sensitive. In our analysis, we only add the predicates to the corresponding branches but do not exclude infeasible paths.

Another way to use predicates is described in the paper “Precise Static Analysis of Binaries by Extracting Relational Information” by A. Sepp, B. Mihaila, and A. Simon [22]. They want to reverse-engineer binaries to actual source code. To achieve their goal, they use a number of abstract domains. To get an idea of the valuation of variables, they use widening and narrowing. For the narrowing, they use the predicates of conditions. Hence, as we do, they use predicates to restrict the possible valuation of variables.

A way to reduce the manual review time for the value analysis in Frama-C is to use the SPALTER plug-in presented in the paper “Driving a Sound Static Software Analyzer with Branch-and-Bound” by S. Mattsen, P. Cuoq, and S. Schupp [19]. The SPALTER plug-in can be used to check if a possible fault, which the value analysis annotated, really can occur. In case it can occur, it provides the user with a counter-example. Otherwise, it removes all the warnings and thereby, reduces the manual review time. However, applying the SPALTER plug-in can be time-consuming.

7 Future Work & Conclusion

Future Work

In our plug-in, we still have two topics in particular that leave room for improvement. The first topic is the way we handle pointers. In the current version, we handle pointers in a safe and sound way and remove all our flow data when we might reassign a pointer since we do not know to which variables the pointer may point. One possible improvement would depend on the value analysis. It performs a points-to-analysis. However, then we would need to always call the value analysis in combination with our plug-in. The other option is to implement a may points-to-analysis [3]. Thereby, we would get a sound over-approximation of variables a pointer points to. Having the over-approximation we could remove only the predicates containing these variable instead of removing the complete flow data. Due to the pointer problem, we also remove all our flow data when we encounter a function call. The result of the points-to-analysis allows us to keep our flow data and we could also pass some flow data down to the function called. Thus, our precision would be better. Further, we could consider to also include pointer-related predicates.

The second topic is the backward analysis. In our backward analysis, we try to push the predicate as far up as possible. Hence, we sometimes push the predicate into branches and insert warnings there. Inserting warnings in branches results in the fact that we sometimes push one warning up and insert at least two warnings for that one warning. Here, considering the fact that we want to reduce the overall number of warnings it would be smarter, to solve the graph problem that we do not push warnings in the last branching and insert the warning exactly before the last branching. Therefore, we would insert only one warning for every warning we push up. With a one-to-one ratio, we could always remove at least one warning for every warning we inserted by applying the forward analysis. Formulating the algorithm to insert warnings along these lines, the backward analysis would not increase the overall number of warnings if we apply the forward analysis afterwards.

Conclusion

In this thesis, we presented the theoretical background for a bidirectional predicate propagation. Our bidirectional predicate propagation consists of two data flow analyses tracking predicates of warnings. We use the backward analysis to propagate predicates upwards and the forward analysis to know what predicates have to hold in a run-time error free execution. Further, we discussed how we can use the result of the backward analysis to insert warnings and the forward analysis to remove warnings. To that end, we presented the algorithms for the warning insertion and removal.

We implemented the bidirectional predicate propagation analysis as a Frama-C plug-in. The plug-in has options to enable the backward analysis, the path sensitivity, the predicate transfer, and the linear constraint solver. By applying our plug-in without

additional options only the forward analysis and warning removal from our analysis gets applied.

Further, we have tested our plug-in in combination with other Frama-C plug-ins. During our evaluation with test suites from the SAMATE project of NIST, we could remove up to 60% of the warnings. From these 60%, our enhancements could remove about 20%. The path sensitivity and the linear constraint solver contributed the most to the 20%. However, our backwards analysis inserted more warnings than we could remove using the inserted warnings. In our future work, we outlined how we could change the backward analysis so that it does not worsen the result anymore. In conclusion, we have a working Frama-C plug-in with enhancements that improve our result. By applying our plug-in, the manual review time can be reduced since we can drastically lower the number of warnings we have to look at.

Bibliography

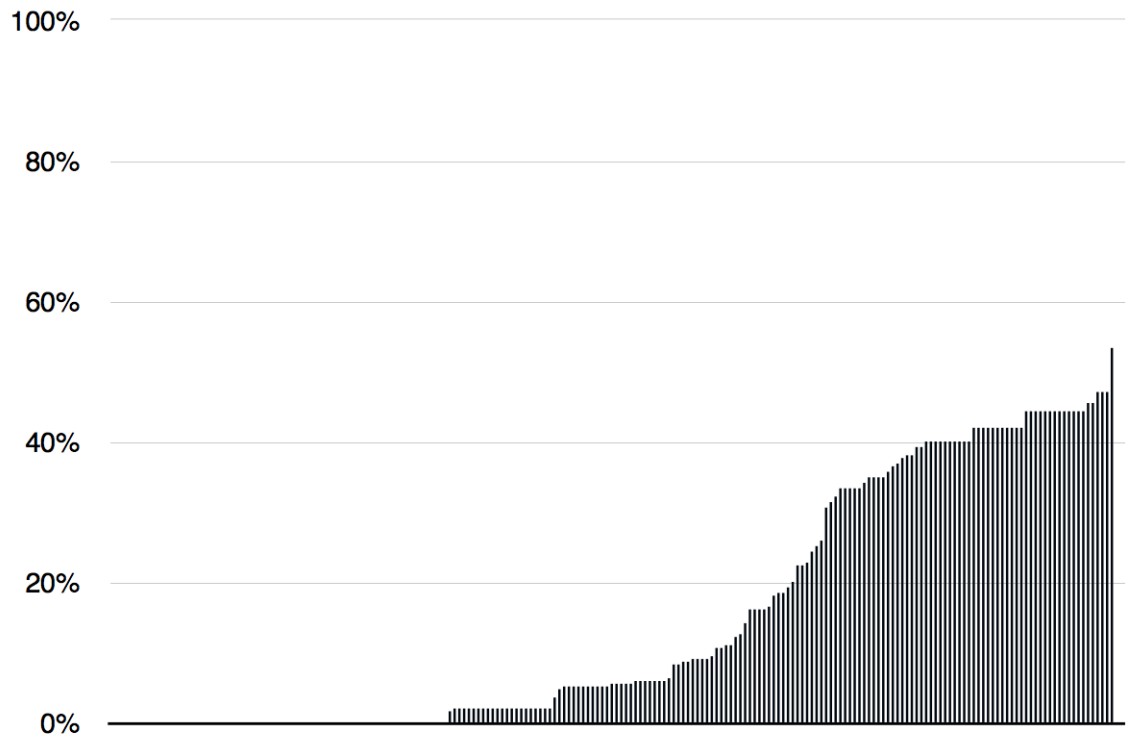
- [1] Iarpa stonessoup phase 1 - memory corruption for c. <http://samate.nist.gov/SRD/testsuites/stonessoup/stonessoup-c-mc.zip>. Accessed: 2014-07.
- [2] Iarpa stonessoup phase 1 - null pointer dereference for c. <http://samate.nist.gov/SRD/testsuites/stonessoup/stonessoup-c-np.zip>. Accessed: 2014-07.
- [3] BERNDT, M. Flow-insensitive points-to analyses for Frama-C based on Tarjan's disjoint-sets. Bachelor thesis, TU Hamburg-Harburg, Mar. 2014.
- [4] BLACK, P. E. Software assurance metrics and tool evaluation. In *Software Engineering Research and Practice* (2005), pp. 829–835.
- [5] BLACK, P. E. SAMATE's contribution to information assurance. *NIST Special Publication* (2006).
- [6] CANET, G., CUOQ, P., AND MONATE, B. A value analysis for C programs. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, 2009. SCAM'09.* (2009), IEEE, pp. 123–124.
- [7] CORRENSON, L., CUOQ, P., KIRCHNER, F., PREVOSTO, V., PUCCHETTI, A., SIGNOLES, J., AND YAKOBOWSKI, B. Frama-C user manual. <http://frama-c.com/download/frama-c-user-manual.pdf>. Accessed: 2014-01.
- [8] COUSOT, P., COUSOT, R., AND MAUBORGNE, L. The reduced product of abstract domains and the combination of decision procedures. In *Foundations of Software Science and Computational Structures*. Springer, 2011, pp. 456–472.
- [9] CUOQ, P. Minimizing-alarms. <http://blog.frama-c.com/index.php?post/2012/03/12/Minimizing-alarms>, Jan. 2014. Accessed: 2014-01.
- [10] CUOQ, P., SIGNOLES, J., BAUDIN, P., BONICHON, R., CANET, G., CORRENSON, L., MONATE, B., PREVOSTO, V., AND PUCCHETTI, A. Experience report: Ocaml for an industrial-strength static analysis framework. In *ACM Sigplan Notices* (2009), ACM, pp. 281–286.
- [11] CUOQ, P., AND YAKOBOWSKI, B. Value analysis manual. <http://frama-c.com/download/value-analysis-Neon-20140301.pdf>. Accessed: 2014-07.
- [12] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989), ACM, pp. 25–35.
- [13] FISCHER, J., JHALA, R., AND MAJUMDAR, R. Joining dataflow with predicates. In *ACM SIGSOFT Software Engineering Notes* (2005), ACM, pp. 227–236.

- [14] FLEMMING NIELSEN, H. N., AND HANKIN, C. *Principles of Program Analysis*, 2nd ed. Springer, 2005.
- [15] GEHRKE, M. A Frama-C plug-in for finding equal-valued expressions using dataflow analysis. Projektarbeit, TU Hamburg-Harburg, Jan. 2014.
- [16] HERRMANN, P., AND SIGNOLES, J. Rte manual. <http://frama-c.com/download/rte-manual-Neon-20140301.pdf>. Accessed: 2014-07.
- [17] JULIEN SIGNOLES, LOÏC CORRENSON, M. L., AND PREVOSTO, V. Plug-in development guide. <http://frama-c.com/download/plugin-development-guide-Neon-20140301.pdf>. Accessed: 2014-07.
- [18] KILDALL, G. A. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1973), POPL '73, ACM, pp. 194–206.
- [19] MATTSSEN, S., CUOQ, P., AND SCHUPP, S. Driving a sound static software analyzer with branch-and-bound. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2013), IEEE, pp. 63–68.
- [20] MUSKE, T. B., BAID, A., AND SANAS, T. Review efforts reduction by partitioning of static analysis warnings. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2013), IEEE, pp. 106–115.
- [21] NECULA, G. C., MCPPEAK, S., RAHUL, S. P., AND WEIMER, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction* (2002), CC '02, Springer-Verlag, pp. 213–228.
- [22] SEPP, A., MIHAILA, B., AND SIMON, A. Precise static analysis of binaries by extracting relational information. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering* (2011), WCRE '11, IEEE Computer Society, pp. 357–366.

Appendix

type predicate =	
Pfalse	(* always-false predicate. *)
Ptrue	(* always-true predicate. *)
Papp of logic_info * (logic_label * logic_label) list * term list	(* application of a predicate. *)
Pseparated of term list	
Prel of relation * term * term	(* comparison of two terms. *)
Pand of predicate named * predicate named	(* conjunction *)
Por of predicate named * predicate named	(* disjunction. *)
Pxor of predicate named * predicate named	(* logical xor. *)
Pimplies of predicate named * predicate named	(* implication. *)
Piff of predicate named * predicate named	(* equivalence. *)
Pnot of predicate named	(* negation. *)
Pif of term * predicate named * predicate named	(* conditional *)
Plet of logic_info * predicate named	(* definition of a local variable *)
Pforall of quantifiers * predicate named	(* universal quantification. *)
Pexists of quantifiers * predicate named	(* existential quantification. *)
Pat of predicate named * logic_label	(* predicate refers to a particular program point. *)
Pvalid_read of logic_label * term	(* the given locations are valid for reading. *)
Pvalid of logic_label * term	(* the given locations are valid. *)
Pinitialized of logic_label * term	(* the given locations are initialized. *)
Pallocable of logic_label * term	(* the given locations can be allocated. *)
Pfreeable of logic_label * term	(* the given locations can be free. *)
Pfresh of logic_label * logic_label * term * term	(* \fresh(pointer, n) A memory block of n bytes is newly allocated to the pointer. *)
Psubtype of term * term	(* First term is a type tag that is a subtype of the second. *)

Table 1: Predicates of Frama-C



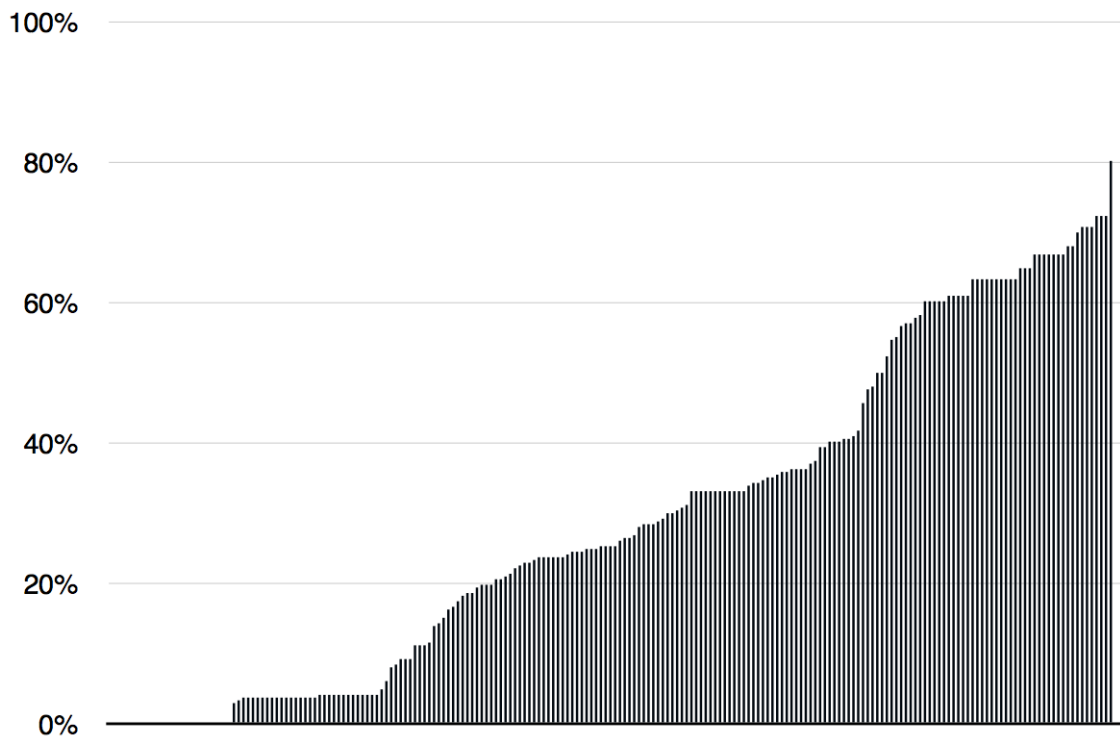


Figure 2: Percentage of Removed Warnings (Y-Axis) for each Program (X-Axis) using the Forward Analysis with `-path`, `-sat`, and `-sub` of the **IARPA STONE-SOUP Phase 1** - Memory Corruption for C Version 1.0 Test Suite from NIST

Listing 1 shows an excerpt of code annotated by the `rte` plug-in. The original code consists of a `for` loop, which got transformed into a `while` loop by Frama-C. The code uses the `for` loop to go through three `while` loops and an assignment, which could be achieved even without the `for` loop. Due to the construct of applying four statements in a row using a `for` loop and the transformations Frama-C does, we push a false positive warning into ten branches. The `rte` plug-in annotates the increasing of `i` with a signed-overflow warning. However, `i` will never go beyond 4 in this piece of code. Thus, the warning is a false positive. By removing the `for` loop, which would not make any difference in the program execution, we would only push two other warnings into two branches. Thereby, the result would be better as it is right now.

```

1  i = 0;
2  while (i < 4) {
3    switch (i) {
4      case 0:
5        while (1) {
6          /*@ assert rte: mem_access: \valid_read(s); */
7          if (*s) {
8            /*@ assert rte: mem_access: \valid_read(s); */
9            if (! ((int)*s == '\u')) {
10             /*@ assert rte: mem_access: \valid_read(s); */
11             if (! ((int)*s == '\t')) {
12              /*@ assert Warnings: rte: signed_overflow: i
13                +1 ≤ 2147483647; */
14              break;
15            }
16          }
17        }
18      else {
19        /*@ assert Warnings: rte: signed_overflow: i+1 ≤
20          2147483647; */
21        break;
22      }
23    }
24    s++;
25  }
26  break;
27  case 1:
28    while (1) {
29      /*@ assert rte: mem_access: \valid_read(s); */
30      if (*s) {
31        /*@ assert rte: mem_access: \valid_read(s); */
32        if ((int)*s != '\u') {
33          /*@ assert rte: mem_access: \valid_read(s); */
34          if ((int)*s != '\t') {

```

```

32         /*@ assert rte: mem_access: \valid_read(s);
33         */
33         if (! ((int)*s != ',')) {
34             /*@ assert Warnings: rte: signed_overflow:
35             i+1 ≤ 2147483647; */
35             break;
36         }
37     }
38     else {
39         /*@ assert Warnings: rte: signed_overflow: i
40         +1 ≤ 2147483647; */
40         break;
41     }
42 }
43 else {
44     /*@ assert Warnings: rte: signed_overflow: i+1
45     ≤ 2147483647; */
45     break;
46 }
47 }
48 else {
49     /*@ assert Warnings: rte: signed_overflow: i+1 ≤
50     2147483647; */
50     break;
51 }
52 s++;
53 }
54 break;
55 case 2:
56     while (1) {
57         /*@ assert rte: mem_access: \valid_read(s); */
58         if (*s) {
59             /*@ assert rte: mem_access: \valid_read(s); */
60             if (! ((int)*s == '_')) {
61                 /*@ assert rte: mem_access: \valid_read(s); */
62                 if (! ((int)*s == '\t')) {
63                     /*@ assert Warnings: rte: signed_overflow: i
64                     +1 ≤ 2147483647; */
64                     break;
65                 }
66             }
67         }
68         else {
69             /*@ assert Warnings: rte: signed_overflow: i+1 ≤

```

```

    2147483647; */
70     break;
71     }
72     s++;
73     }
74     break;
75     case 3:
76     /*@ assert rte: signed_overflow: -2147483648 ≤ x*x; */
77     /*@ assert rte: signed_overflow: x*x ≤ 2147483647; */
78     /*@ assert Warnings: rte: signed_overflow: i+1 ≤
        2147483647; */
79     name = (s + x * x) - 121;
80     break;
81     default:
82     /*@ assert Warnings: rte: signed_overflow: i+1 ≤
        2147483647; */ ;
83     break;
84     }
85     /*@ assert rte: signed_overflow: i+1 ≤ 2147483647; */
86     i++;
87 }
```

Listing 1: Excerpt of C Code from `./TC_C-785_v934/src/desaturate.c` of the **IARPA STONESOUP Phase 1** - Memory Corruption for C Version 1.0 Test Suite from NIST