Robust Dynamically Deployed Static Analysis for Java

by

Sebastian Gfeller

BSc., Computer Science
École Polytechnique Fédérale de Lausanne (2007)

Submitted to the School of Computer and Communication Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

June 2009

© Sebastian Gfeller, MMIX. All rights reserved.

The author hereby grants to EPFL permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author .....
School of Computer and Communication Sciences
June 19, 2009

Robust Dynamically Deployed Static Analysis for Java

by

Sebastian Gfeller

Submitted to the School of Computer and Communication Sciences
on June 19, 2009, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Guaranteeing the absence of errors in a program is a great challenge and many static analysis techniques have been developed to see whether a program is correct before we run it. There is, however, a whole class of programs that are especially difficult to analyze in this manner. These programs have a very different behavior that depends on configuration files, command line parameters and user input. Also, they are long running. For such programs, there are advantages to *dynamically deployed* analyses that can tell us after the program has been run for some time whether crashes are still possible in its future execution.

In this thesis, we see how to combine current program state with static analysis techniques to predict absence of errors in the future. We show how, if a program can not yet be guaranteed to run correctly, we can make future checks more efficient. The thesis also explores how the analysis can run in parallel with program execution without slowing down execution too much.

The techniques that are explored are implemented as an extension for the Java PathFinder Java Virtual Machine.

Thesis Supervisor: Viktor Kuncak
Title: Professor

Acknowledgements

First of all, I'd like to thank Viktor Kuncak for advising me on my thesis. For being enthusiastic about my ideas and helping me to express them correctly. Every discussion that we had motivated me to dig into another subject, another paper, in an attempt to better understand the relations and implications of what we were just talking about.

My thanks also go to Philippe Suter. His explanations early on in the project and comments all along helped me to keep going. He offered his help many times and while I did not use it very often, it always forced me to make my vague ideas more conveyable.

I express my gratitude to Ondřej Lhoták for the Scala-Soot bridge. It's much more fun to program if you have good tools at your disposal, and thanks to his library I could write my code in an intuitive and succinct manner.

This thesis would not have been finished were it not for the many coffee breaks that I shared with Daniel. All these years at EPFL, he has been a good colleague.

I am grateful for every distraction that I had during these last four months. Be it the poker nights with Ken, the long hike along lake Zurich with Christian and Adrian, late season skiing with Michael or the card playing evenings with Kathrin and Jeanine. I also thank Eva and Tobias for inspiring correspondence.

Finally, I'd like to thank my parents, my brother and my sister for supporting me all these years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We would like that the programs we write run without errors all the time. To this end, many static analysis techniques were developed. These techniques can verify properties that hold in all possible executions of the program, for instance whether an array access will always be done with an index within bounds.

Sometimes the properties that can be guaranteed for all runs are not strong enough. It could be that the program does wildly different things depending on initial parameters like command line arguments or configuration files. It might even be that some of those combinations are not error-free.

While a global guarantee of correctness is desirable, it is also useful if we know early on that the current execution of the program will be free of errors. This is of course a solved problem when we consider the end of execution – either the program crashed or it performed its task as expected. Obtaining the information at that point is too late.

But let's consider a long running program, like a web server. The server depends on an initial configuration to decide which modules to load and what behavior to allow. The server will be running a long time, and only in some extraordinary case, it might crash, given this configuration.

It is for such programs that dynamically deployed static analysis can give useful results. A dynamically deployed analysis takes the current state of the program into

account. If we know ten seconds after the web server is started that for this run we will not crash, then we have obtained a useful bit of information.

1.1 Problem statement

We are interested in dynamically deployed static analysis. That means that we want to use the current program state - what is stored in memory and where we are in the execution, and combine it with an analysis of the structure of the program.

The specific scenario that we will be looking at is byte code executed in the Java Virtual machine. Running in parallel with program execution, we want to see whether the program can still produce errors in the future.

1.2 Contributions

This thesis outlines how we can use current program state together with static analysis techniques to find out whether errors can still happen in the future. More precisely, we present the following contributions:

1. We present a combination of forward and backward analysis based on abstract interpretation [7] of an intermediary language obtained from Java bytecode. This analysis allows us to state whether a program can still run into errors in its future.

2. We describe the implementation of the RDDSA extension to the Java PathFinder [2] virtual machine and model checker. In this tool, we use the analyses presented.

3. Using the extension, we show how the analysis performs on different Java programs.

Chapter 2

An example

As an example of a long-running application which might benefit from early detection of errors, we mentioned a web server. I will now illustrate what the analysis can do using a simple server whose code can be found in figure 2-1.

The server consists of an initialization part and a big while loop. First, a configuration file is read. Depending on its contents, the field `safe_mode` is set to true or false. Then, inside the loop, requests are handled. Depending on the server configuration, requests are handled differently.

Suppose now that in our configuration file, we disabled safe mode. As we can see on line 18, we are performing a calculation that, depending on the request that we received on line 9, might produce an arithmetic exception.

We can model these crash possibilities as failed assertions - the program did not behave as expected. In our case, we expected that `r.n` was not equal to 0. Note that we would not run into this problem if we enable safe mode. With safe mode enabled, we could assure that such a crash does not happen even before execution enters the main loop (but after we have read the configuration file).

Another way to look at program errors are error conditions. At every program point, we can describe what could go wrong in the future in an error condition. If this condition is satisfied, then the program can run into an error. The web server is initialized on line 5. For this line, the error condition is simply $\neg$s.safe_mode $\wedge \exists r.n(r.n = 0)$. That means that safe mode was disabled and there could be a request with r.n equal to 0.

```
1  public class WebServer {
2
3      public static void main(String[] args) {
4          boolean done = false;
5          Server s = new Server(8080);
6          s.readConfig(".server_config");
7          while (!done) {
8              Connection c = s.acceptConnection();
9              Request r = c.getRequest();
10
11             int response = 0;
12
13             if (s.safe_mode) {
14                 // Compute response
15                 if (r.n != 0) response = r.f / r.n;
16             } else {
17                 // Compute response without checks
18                 response = r.f / r.n;
19             }
20
21             c.writeResponse(response);
22             c.close();
23         }
24     }
25 }
```

Figure 2-1: A simple web server

Note that a satisfied error condition is necessary but not sufficient for an error to occur - it might for example be the case that no request with r.n equals to 0 is ever sent to the web server, in which case it will not crash. However, if the error condition is not satisfiable (in our example when safe mode is enabled), then we can be sure that no error will occur in the future of the current execution.

2.1 What the analysis should do

If we have an error condition for the program point at which we are currently, then we can check whether we can run into errors in the future by checking the satisfiability of the error condition given the current state.

We have, therefore, two tasks that our analysis must perform.

- Find a *good* error condition for the current state
- Obtain the current state

Note that finding a *bad* error condition is easy: just take the formula *true*, which is satisfiable in any state and therefore gives absolutely no useful information.

To achieve the main goal of obtaining good error conditions, we have to have some sort of intermediate representation of the program. We look at this representation in chapter 4. But before that, let's examine what information we can obtain with dynamically deployed analysis.

Chapter 3

Dynamically deployed analysis

As we have seen in the last chapter, using static analysis might help us detect errors earlier on. Dynamically deployed static analysis might be done in two ways: Forward analysis propagates information from program entry (or the current program point) towards the end of the program. Finding out which actual methods get called and constant propagation are two of the forward analyses that we use. Backward analysis, on the other hand, begins at program exit and propagates information towards program entry.

Finding error conditions is a backward analysis in principle. We collect all the information from program points where we can fail and propagate this information towards the position in the program that is currently being executed. In the end, what we want to obtain is a condition that we can evaluate in the current state. If that condition does never hold, then the rest of the execution will be free of errors.

We cannot find an algorithm that tells us for all programs whether they will definitely fail or definitely run without errors. The goal of the analysis that we will propose later is to be cautious: It never reports that we will run without errors if an error is possible, on the other hand it might report some programs as not error-free at the current position when in fact they are.

3.1 How to obtain information about the future

Our analyses will be fixpoint calculations. This means we will start with some initial facts about the program (for example the current program state), and then add facts that follow until no more facts can be added. For constant propagation, we start with the initial assumption that every variable and every field is a constant. Then we look what happens in the future execution and have to see that some things that we thought were constant in fact are not.

For backward analysis, we start with the assumption that no errors occur and then add all the errors that could occur from the end of the program to the beginning.

3.2 How to refine information

As execution progresses through the program, we get more and more information about future program execution. This is expressed in the new program state. The most extreme new information is that the error condition is unsatisfiable, and therefore our future program execution will run without errors. But even errors might still occur, we want to use the new information. We are especially interested in error conditions that are easier to check.

First, we can already partially evaluate the old error conditions with all the information that we know will stay constant, generating new error conditions that only depend on information that we do not already have.

We might be tempted to just insert the information that we already have in the old formulas and run the analysis again until it converges. Unfortunately, this does not work because of the bottom-up way that we calculated our error conditions: we started with the assumption that there were no errors and then added all the possible errors. If we now want to propagate this information, we run into the old value of the next state that is more general than what we obtained.

To use the information, we have to erase at least parts of the old facts and redo the backward analysis.

```
fun forward_check(F : formula representing forward analysis information,
                  S : program state) =
  let EC = error_condition(S)
  if EC ∧ F not satisfiable then
    return true
  else
    let F′ = propagate_forward(F)
    for T in successor states of S do
      if ¬forward_check(F′, T) then return false
    done
    return true
  fi
```

Figure 3-1: Forward checking for errors

3.3 Moving toward absence of errors

As mentioned before, forward analysis can be used to partially evaluate the error conditions in the states of future execution. Now suppose we partially evaluate such a future error condition and find out that it will not hold. This means that, if we reach that state, our program will run without errors in the future.

Put another way: suppose the information obtained by the forward analysis is stored in formula F representing what we know about the future of this program state. Let G represent the collected error condition in the same state. If $F \wedge G$ is then not satisfiable, this means that in the future of this state, no errors will occur. From this, we can obtain a forward checking algorithm, as seen in figure 3-1.

Chapter 4

Conversion to intermediate language

To analyze a program, we must obtain a representation of it that is suitable to such an analysis. In this chapter, we see how we can obtain such a suitable representation.

In our scenario, the program is given in the form of class files containing the methods as bytecode instructions. This is not the format that we want to analyze. For our analysis, we want to deal with Control flow graphs with a simple language approximating what happens in the program. We describe this language in the next section. Afterwards, I will explain how to obtain the control-flow-graphs from Java class files.

4.1 The intermediate language

The intermediate language that we use for our analysis is very simple. It contains arithmetic expressions and assignments to fields and variables. This language does not capture the full expressiveness of Java byte-code (like static fields, monitors, and more basically, array functions), but instead focuses on describing arithmetic operations in a heap.

Table 4.1 shows the grammar of our intermediate language.

S	::=	V.F := e | V := V | V := e | V := new | V.F := $input()$ | $invoke$(M(, A)*) | $[\phi]$ | $\{\phi\}$ | nop
ϕ	::=	$e = e$ | $e \neq e$ | $e \leq e$ | $e \geq e$ | $e < e$ | $e > e$ | $true$ | $false$
e	::=	V.F | C | $e + e$ | $e - e$ | $e * e$ | e | e/e | ?
A	::=	V | e
C	::=	integer
M	::=	method identifier
V	::=	variable identifier
F	::=	field identifier

Table 4.1: Context free grammar of the intermediate language

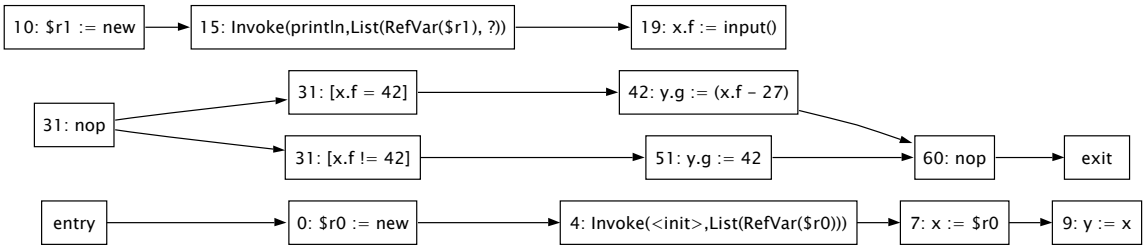


Figure 4-1: The elements of the intermediate language

The curly braces stand for an assertion, the brackets for an assumption. *nop* does nothing and is needed to identify branching points. This language is then used for the code blocks of a control flow graph with nondeterministic branching. See figure 4-1 for an example using all the constructs that are mentioned.

4.2 From bytecode to CFG via Soot

The analysis that we perform will be most useful for a virtual machine interpreting bytecode, because we have good information about where we are in the program in such a system.

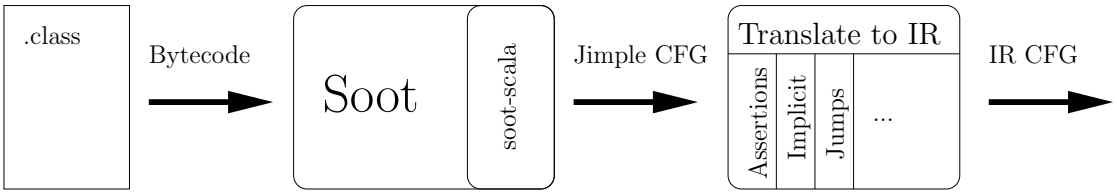


Figure 4-2: Intermediate language translation schema

Let's now look at the steps required to get from Java bytecode [15] to a control flow graph of our intermediate language. Java bytecode consists of over 200 individual instructions operating on an operand stack. This is not very suitable for translation, as many of these instructions are only different in what type of arguments they take.

So our starting point is not Java bytecode, but the Jimple intermediate format [19], which we obtain by using the Soot [18] framework on the class files that need translation. The whole translation process is outlined in figure 4-2. After a translation to Jimple, we are left with far fewer cases to handle: Jimple has less than 20 different instructions.

What we obtain from this transformation is a CFG containing Jimple instructions in the basic code blocks. These instructions are then further processed by the Soot Scala interface [12].

The transformation from a Jimple CFG to a CFG in our intermediate language takes multiple steps. Let's outline these steps now.

4.2.1 Null pointers

In Java, reference variables might also point to *null*. We solve this case by introducing another special variable, $NULLVAR$. So x = null becomes x := NULLVAR.

4.2.2 Non-primitive operations

Some operations will not be represented in our intermediate language. Say you want to obtain the length of an array. Even though we don't obtain the true value, we want to model the fact that the variable now contains *some value*. For this, we introduce the value unknown or ?. So obtaining the length of an array and storing it in field $x.f$ is modeled as follows: x.f := ?.

4.2.3 Translating assertions

Since version 1.4, Java contains an `assert` statement. There is, however, no bytecode instruction for this. Instead, assertions are compiled as represented in figure 4-3.

```
1 assert p;
2
3 // becomes
4
5 if ( !$assertionsDisabled )
6     if ( !p ) throw new java.lang.AssertionError();
```

Figure 4-3: How assertions are translated

Where $assertionsDisabled is a static field that determines whether assertions will be checked at runtime.

At the moment when we have to throw the AssertionError exception, we have arrived in an error state. So we can replace throwing the assertion with the intermediate language construct $\{false\}$.

4.2.4 Finding implicit assertions

Some instructions can go wrong even if there is no assertion compiled into the code. This has to be made explicit. The Java specification [15] outlines which instructions can output which exceptions. Field accesses, for example, can fail if they are performed on null. Integer divisions fail when the divisor is 0. So if a division is performed, the translation first has to assert that the divisor is not zero.

4.2.5 Removing conditional jumps

You may have noted that our intermediate language contains no if expression. This is because we replace conditional jumps with inconditional jumps + assertions.

How this translation works can be best seen with the small example 4-4. For the then part of the block, we first assume that the condition is true. Conversely, for the else block, we assume that the condition is false.

```
1 if (x.f == 42) {
2   foo = bar;
3 } else {
4   foo = baz;
5 }
```

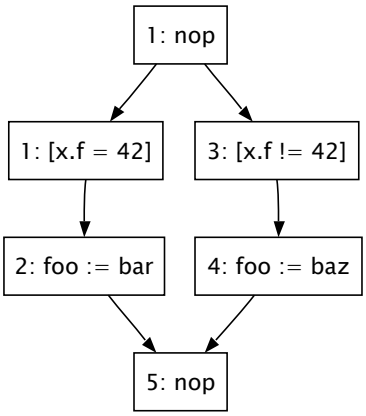


Figure 4-4: Translation into intermediary language CFG

4.2.6 Array accesses, class fields and more

To get a full translation of Java bytecode into our intermediate language, we will have to handle many more special cases. These are, however, only of marginal value to the rest of this work. The interested reader is referred to the documentation of the rddsa extension for more information.

Chapter 5

Backward analysis with formulas

This chapter presents our solution to detecting possible errors earlier on. We base our error detection on the concept of *error conditions*, which are the dual of weakest liberal preconditions. Our goal is to produce a condition on the current heap. If this condition is satisfiable in some state, then we might run into an error. If it is unsatisfiable, then no error will occur.

First we will look at how such an error condition might be structured and what we can express with it. We then specify the transformation rules for instructions of our intermediary language. Using these rules on a program with loops, we may obtain an arbitrarily complex error condition. So we have to produce an approximation of the error condition.

We will ponder on the handling of method calls in a separate section as an illustration of the finer points of the analysis. Finally, we will see how to combine the information that we obtained in the last chapter with our error conditions to actually check whether our program may run into errors.

5.1 Error conditions

Given the current program state in the middle of execution, there is one way to detect whether our program will run without errors that is always correct. We can describe the error condition as the following function:

fun Perfect_Error_Condition(*state* : current program state) =
run program with current state
if program crashed **then return** true
else return false

Of course this error condition is not useful at all. The aim of our analysis will be to obtain an error condition that is not only computable but even efficiently computable. Our error condition is efficiently computable if we obtained the information whether our program will execute correctly before we have executed it. Another important requirement is that we do not have false negatives. If we have obtained an error condition representing a certain class of errors and have shown that it cannot be satisfied, then the program really should not crash because of such an error. The converse does not have to be true: it might be that we obtain a satisfiable error condition, but the program will run correctly anyway.

5.1.1 Error conditions as formulas

A very natural way to represent this error condition is as a logical formula with inequalities on the elements of the heap. Going back to our web server example, one such formula would be

$$server.safe\_mode \neq 1$$

To be more precise, for our error conditions, we will combine inequalities on fields, variables and arithmetic expressions with the usual logical operations $\neg, \wedge, \vee$. We will also add existential quantification over reference variables, $\exists$.

Because we want to collect all the errors that may occur in the future execution, it seems natural to express the generation of these error conditions as a backward analysis. Like the forward analysis, we will specify our domain, preorder and abstract interpretation and calculate a fixpoint.

$$
\begin{array}{lll}
F & ::= & F \wedge F \mid F \vee F \mid \neg F \mid F \Rightarrow F \mid \exists V \ . \ F \mid B \\
B & ::= & E < E \mid E \leq E \mid E = E \mid E \neq E \mid E \geq E \mid E > E \mid \top \mid \bot \\
E & ::= & E + E \mid E - E \mid E * E \mid E/E \mid C \mid V \mid V.F(.F)^* \\
V & ::= & \text{variable name} \\
C & ::= & \text{integer constant} \\
F & ::= & \text{field name}
\end{array}
$$

Table 5.1: BNF grammar of the error condition formulas

5.1.2 The domain of error conditions

We were a bit vague when we said that we wanted to use formulas for error conditions in the last section. Table 5.1 shows the grammar of such formulas.

The grammar uses the symbols $\top$ and $\bot$. These represent the values *true* and $false$ respectively, and they hint already at what function they will have in the preorder of error conditions.

5.1.3 The preorder of error conditions

As with the forward analysis, we have to define an order $\sqsubseteq_F$ on the formulas. During our analysis, we will start with the assumption that no error occurs and work our way to the correct error condition. The error condition $\bot$ expresses exactly this fact. No possible assignment to the fields of variables could render this condition true. On the opposite end, we have $\top$. This error condition will be true no matter what state. We can now define the order on F_1 and F_2 based on which error condition is "easier" to satisfy. $F_1 \sqsubseteq_F F_2$ if every time we can satisfy F_1, we will also be able to satisfy F_2, but if F_1 is unsatisfiable, then F_2 might still be satisfiable. This is exactly what is expressed by implication:

$$F_1 \sqsubseteq_F F_2 \text{ iff } F_1 \Rightarrow F_2$$

We can easily see that $\bot$ is the lowest element of this preorder:

$$\bot \Rightarrow F \text{ for any } F$$

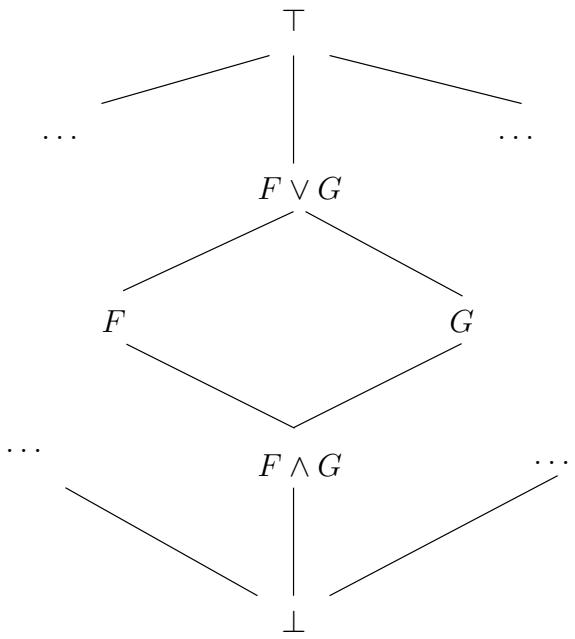


Figure 5-1: Example of an error condition preorder

and $\top$ is the highest:

$$F \Rightarrow \top \text{ for any } F$$

The least upper bound between two elements F and G is also quite natural: it is simply logical disjunction $\vee$. To verify this, it is easy to see that the following holds by definition:

$$F \sqsubseteq_F (F \vee G) \text{ and } G \sqsubseteq_F (F \vee G)$$

In the same way , we can see that $\wedge$ represents a logical choice for greatest lower bound. Figure 5-1 shows the partial Hasse diagram with two distinct error conditions F and G.

Now that we have the preorder defined to our needs, it is time to look once again at abstract interpretation of the program instructions. This time, however, we will go through the control flow graph from the exit to the entry. Given a formula F representing the error condition after an instruction, we want to calculate F', the

$[\![x.f := \epsilon]\!]_{\mathcal{B}}$	If we can calculate expression ϵ, then $F' = F[x.f \to \epsilon]$. Otherwise, let y be a fresh variable not present in the program and $F' = \exists\ y\ .\ F[x.f \to y.f]$.
$[\![x := y]\!]_{\mathcal{B}}$	$F' = F[x \to y]$
$[\![x := y.f]\!]_{\mathcal{B}}$	$F' = F[x \to y.f]$
$[\![x.f := y]\!]_{\mathcal{B}}$	$F' = F[x.f \to y]$
$[\![x := new()]\!]_{\mathcal{B}}$	Let y be a fresh variable, $F' = \exists\ y\ .\ F[x \to y]$
$[\![x.f := input()]\!]_{\mathcal{B}}$	For the fresh variable y, $F' = \exists\ y\ .\ F[x.f \to y.f]$
$[\![[\phi]]\!]_{\mathcal{B}}$	$F' = F \wedge \phi$
$[\![\{\phi\}]\!]_{\mathcal{B}}$	$F' = \phi \Rightarrow F$

Table 5.2: Abstract interpretation for error conditions

error condition before that instruction. Table 5.2 shows how the error condition is affected by the different instructions.

If an assertion ϕ occurs in the program, the resulting error condition is just $\neg\phi$. This has to be combined with the errors E the program might run into after this instruction. So either ϕ holds, in which case it depends on E whether the error condition is true, or ϕ does not hold, in which case the error condition is true.

Assumptions, on the other hand, constrain the assertions. Let's say before the assumption, we were certain that we would fail: $F = \top$. Now the assumption says that this path is only taken if condition C holds. In this case, the error condition becomes $C \wedge \top = C$.

All assignments have one thing in common: They are performed as substitutions on the formula. If we cannot obtain a value needed to calculate the expression, we existentially quantify over this value. This means that any possible result contributes to the error condition.

An instruction might have multiple outgoing edges. In this case, all the formulas arriving from the different edges have to be joined. Say we have error condition C on one path and error condition D on another. Then after the instruction that branches to these two paths, we have the error condition $C \vee D$ - either we fail on one path or on the other.

```
1  A x = new A();
2  A y = new A();
3  y.g = 13;
4  x.f = AbstrUtil.input();
5  if (x.f == 42) {
6    x = y;
7  } else {
8    y = x;
9  }
10 assert(x.g == 13);
```

Figure 5-2: A loop-free program

For an example of a loop-free program, see figure 5-2. With some details left out, our translation to intermediary language and backward analysis of error conditions transforms the program approximately to the CFG in figure 5-3.

Until this moment, we have left out two important details. First, we have not looked on how to handle method invocations, and secondly, how to handle programs with loops. We will look at these problems in the next sections.

5.2 The problem with loops

The backward analysis as described propagates error conditions correctly if there are no loops. But let's look at what happens if there is a loop:

```
1  package examples;
2  public class MiniLoop {
3      public int sum; public int i;
4      public static void main(String[] args) {
5          MiniLoop x = new MiniLoop();
6          x.sum = 0; x.i = 10;
7          while (x.i > 0) {
8              x.sum = x.sum + x.i;
9              x.i = x.i - 1;
10         }
```

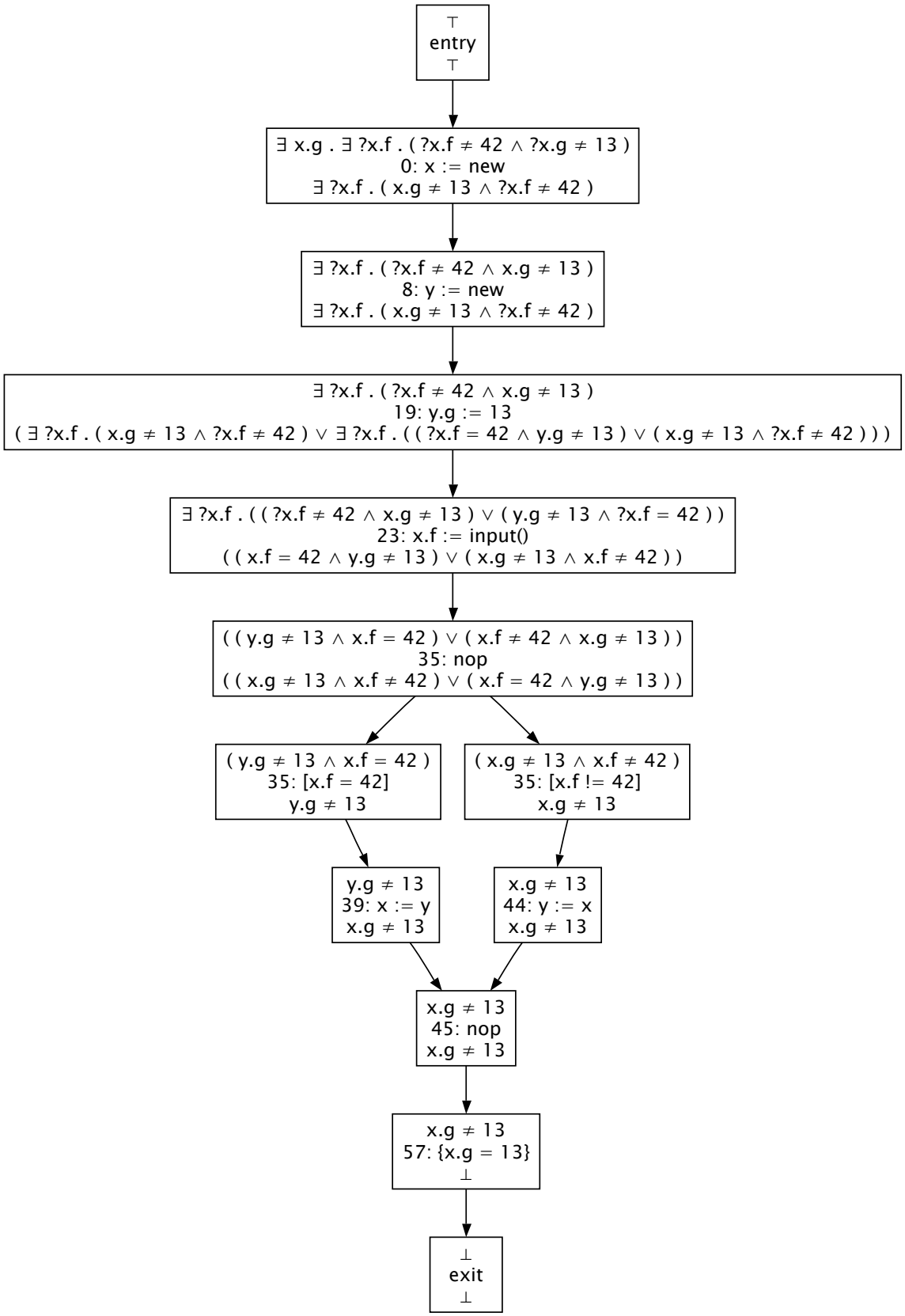


Figure 5-3: Error conditions of loop-free program 5-2

```
11            assert(x.sum >= 0);
12        }
13    }
```

At program exit, the error condition is $\bot$. Then we encounter the assertion (on line 11). This leads us to the error condition $x.sum < 0$.

Entering the loop and traversing the loop body until, we get an error condition for the loop body at line 7 that is $x.i \leq 1 \wedge x.i > 0 \wedge x.sum + x.i < 0$. Now we have to perform the join between the two error conditions:

$$(x.i \leq 0 \wedge x.sum < 0) \vee (x.i \leq 1 \wedge x.i > 0 \wedge x.sum + x.i < 0)$$

We have not reached a fixpoint yet, and thus have to go through the loop again. After the second time around, we obtain the error condition

$$x.i > 0 \wedge ((x.i \leq 1 \wedge x.sum + x.i < 0) \vee (x.i \leq 2 \wedge x.i > 0 \wedge x.sum + 2x.i < 1))$$

You can imagine that the formula at the loop head grows bigger and bigger every time we go through the loop body again. From hand, we could work out the (strongest) error condition holds at the loop head, $x.sum < \frac{x.i(x.i+1)}{2}$, but we want a fully automatic solution.

In the normal weakest precondition calculations, this is referred to as inferring a loop invariant: A formula that holds At the loop head and after each loop iteration. In our case, we want to find the loop error condition.

Our loop error condition E_l has to satisfy the following properties:

- The error condition after the loop, E_{after} implies E_l
- The error condition obtained after going through the loop body once more, F, implies this error condition: $F \rightarrow E_l$

The problem is that such a loop error condition needs not have a direct link to the error conditions that we obtain by going through the loop, as we have seen with the error condition that we came up by hand.

Thus, we need a heuristic to come up with a loop error condition that is general enough so that every loop iteration implies it. For this, we turn to the induction-iteration method. The method was used by Xu, Miller and Reps [20] to find loop invariants in machine code, which is very close to the use we have in mind here. It was first introduced by Suzuki and Ishihata [17] for array bounds checking. The notation for error conditions is quite different, however.

The first thing we must give up to find a loop error condition is termination. Even if the error condition is not satisfiable, this does not mean that the program will *terminate* without errors, just that it will run without errors.

The basic idea of the induction-iteration method is to accumulate formulas that express the loop error condition. Let $E(0)$ be the error condition obtained by going through the loop body once. $E(i+1)$ is obtained by propagating $E(i)$ through the loop body again. Now the error condition before the loop can be expressed:

$$E_l = \bigvee_{i \geq 0} E(i)$$

Figure 5-4 shows this graphically (imagine the rectangles to be the set of program states that could crash). Now suppose we have gone through the loop n times, and as it just so happens, at iteration $n+1$ we obtain an error condition that implies one of the previous error conditions:

$$\begin{aligned}
&\exists i \;.\; 0 \leq i \leq n \wedge (E(n+1) \Rightarrow E(i)) \\
&\Leftrightarrow \textstyle\bigvee_{0 \leq i \leq n} E(n+1) \Rightarrow E(i) \\
&\Leftrightarrow \textstyle\bigvee_{0 \leq i \ leqn} \neg E(n+1) \vee E(i) \\
&\Leftrightarrow \neg E(n+1) \textstyle\bigvee_{0 \leq i \leq n} E(i) \\
&\Leftrightarrow E(n+1) \Rightarrow \textstyle\bigvee_{0 \leq i \leq n} E(i)
\end{aligned}$$

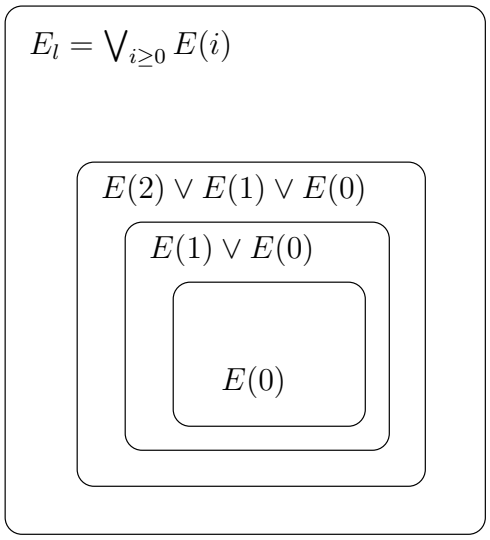


Figure 5-4: Loop error conditions

Then we know that we are done: instead of plugging in $E(n+1)$ and going through the loop again, we could have just as well used the "bigger" / less precise error condition $E(i)$, which would only lead us again to $E(i+1)$, an element we already have in the disjunction.

Because we are dealing with dynamically changing state (and thus dynamically changing truths), we can't do a second test that could be done when doing the method purely statically: In this case, we could check for each result in between whether, when we propagate the intermediary loop condition to the beginning of the program, it could be satisfied. If that was the case, we would be finished: The error condition grows with each iteration, and we already have an example of a state that can go wrong. But because our state might advance in such a way that errors are no longer possible, we cannot do this step.

Figure 5-5 outlines the algorithm. We query a theorem prover to see whether the implication holds. Currently, the analysis uses the CVC3 theorem prover [1]. CVC3 returns satisfiability results, so we have to check whether the negation of the implication is not satisfiable, actually.

Because the loop condition is the infinite disjunction of $E(i)$s, we know that we will converge to it using this algorithm. However, convergence is not enough, as

```
proc Induction_Iteration() =
    Create error condition E(0)
    for i = 0 to M do
        Theorem_prover(E(i) ⇒ ⋁_{0≤j≤i−1} E(j)) match
            case VALID : return ⋁_{0≤j≤i−1} E(j)
            case INVALID | UNKNOWN :
             E(i + 1) = propagate E(i) through loop body
    done
    return ⊤
```

Figure 5-5: Induction-iteration algorithm for error conditions

we specified before, we would actually like to obtain results before the program has finished running. Therefore we introduced an upper bound M on the number of iterations. If we don't succeed with so many iterations, we have to give up using only induction-iteration.

Unfortunately, this case happens often, even in quite simple programs. For example, the induction-iteration algorithm will not find a loop error condition for the sum program. Still, we would like to be able to find at least some nontrivial error condition.

The problem with the sum program is that we approach the solution from below, but we never find one that holds for the next bigger integer value of $x.i$. One way to handle this is to "overshoot". We convert the current formula to conjunctive normal form and then try out the conjuncts as candidates for the loop error condition. Figure 5-6 shows this case: the current disjunction of $E(i)$s does not cover the loop error condition E_l, but if we choose either conjunct 2 or conjunct 3, then we obtain a formula that covers the minimal loop condition.

There are more ways to obtain better precision for error conditions, and the other techniques used by Xu, Miller and Reps can also be adapted. In invariant generation, it is also common to try out templates for the loop invariant, which can of course also be used for finding loop error conditions. However, the problem is then reduced to trying out the right templates in the right order, something which depends strongly on the functionality of the programs to be analyzed, so therefore this approach is less interesting for our situation.

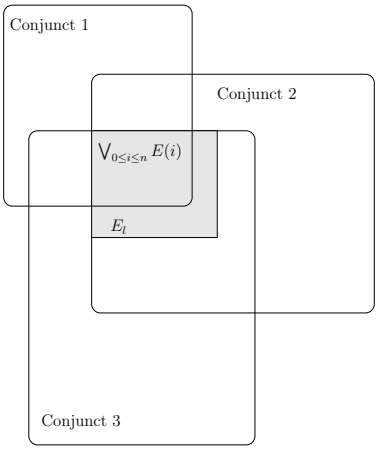


Figure 5-6: Finding loop error conditions by overshooting

5.3 How to inline method calls

As you might remember, we left out one case in our backward analysis, the case of method invocations. Unlike the forward analysis, the backward analysis has to be interprocedural. The problem is that every method invocation might change the error condition by adding additional ways a program could fail, but also by changing the error condition of the invoking method.

If there are no recursive calls, we can easily define a costly abstract interpretation for method invocations. For a method call $invoke(M, a_1, \ldots, a_n)$, we analyze all the methods that M may point to, taking F, the error condition after the instruction as initial value for the exit node. We then obtain $F_1, \ldots, F_m$ from each possible method that could have been called. The new error condition becomes the disjunction of all these method error conditions.

Finding out which methods might be invoked at each invocation instruction is actually in itself a very interesting problem. For our analysis, we leverage the Soot framework to efficiently find those candidate methods for us using call-graph analysis [16].

If we are dealing with recursive programs, we have to work with the induction-iteration method again. Figure 5-7 shows the sum program, but rewritten with a recursive method invocation.

```
1  package examples;
2
3  public class RecursiveMiniLoop {
4    public int sum;
5    public int i;
6    public static void main(String[] args) {
7      RecursiveMiniLoop x = new RecursiveMiniLoop();
8      x.sum = 0; x.i = 10;
9      calculateSum(x);
10     assert(x.sum >= 0);
11   }
12
13   public static void calculateSum(RecursiveMiniLoop x) {
14     if (x.i > 0) {
15       x.sum = x.sum + x.i;
16       x.i = x.i - 1;
17       calculateSum(x);
18     }
19   }
20 }
```

Figure 5-7: Recursive Sum example

Because we are calling the method calculateSum again inside its definition, we cannot just inline the method body once again. Instead, we prepare the arguments and link the method invocation to the method entry (which becomes a loop head). The induction-iteration now works in a very similar fashion. Figure 5-8 shows how the CFG would look for the calculateSum method, leaving out some details of parameter naming.

5.4 Combining with forward information

When we have completely run the backward analysis, every instruction in the control flow graph will be mapped to an error condition. We are now ready to combine it with the current state. After that, we will see how the constant propagation that we have also found can be useful.

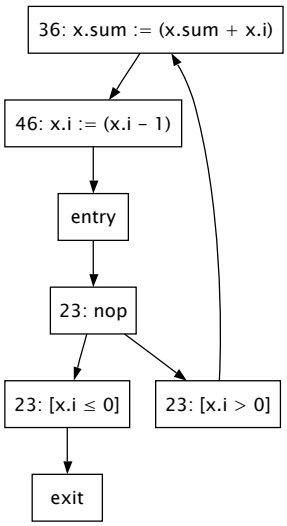


Figure 5-8: Recursive invocation of calculateSum

5.4.1 Performing a check

Figure 5-9 shows how to perform the actual test. We presuppose that the results of the backward analysis are stored in `Fact_Map`. After having obtained the current position, we evaluate the error condition at this position in the current state. For these steps, the virtual machine execution has to be stopped because we require an unchanging heap when we evaluate the formula.

After this step, the virtual machine can continue executing the program.

Finally, using the theorem prover, we see whether the error condition that we evaluated with the current state is satisfiable. If it is not satisfiable, then this means that the current heap does not lead us in an error. If we could find a satisfying assignment, then errors might still happen.

If the error condition is still satisfiable, the algorithm waits some time before it rechecks. One sensible possibility is to wait for the next garbage collection to occur, because access to the heap will be blocked anyway.

Figure 5-10 shows how the analysis interacts with the running program. To not slow down execution too much, it is important that the analysis runs in a separate thread. Note that while the backward analysis does not depend on current program state, we have to know which concrete methods will be called. This information depends on

```
proc Check_errors() =
    Interrupt VM.
    let position = Get_Current_Position()
    let bw_formula = Fact_Map(position)
    let mapped_formula = evaluate bw_formula with current state
    Continue VM execution.
    Theorem_Prover(mapped_formula) match
        case UNSATISFIABLE :
            Log("No more errors will occur")
        case SATISFIABLE|UNKNOWN :
            Sleep()
            Check_Errors()
```

Figure 5-9: Basic error checking algorithm

what classes are actually available at runtime. Without that factor, we could also perform the analysis before program execution, storing the results as annotations to be checked.

5.5 How to incorporate forward analysis

One feature that is suspiciously missing from the analysis is the constant propagation that we developed. We can reintroduce the information obtained from forward propagation to obtain better error conditions. Suppose a first check was performed and the program is not yet error-free. We will do a second check later on, combining the state formula with the formula of the future instruction in backward analysis.

Constant propagation can help us make this second test cheaper. After the first test, no matter how the program progresses, some fields will remain constant. A second check will incorporate their value in the state formula and at the moment of evaluation, use their values to evaluate the error condition. However, to better use the idle time between checks, we can already partially evaluate the error conditions, making future evaluations faster.

Figure 5-11 roughly describes the algorithm used. Simplifications that can be performed are:

- Fold arithmetic expressions

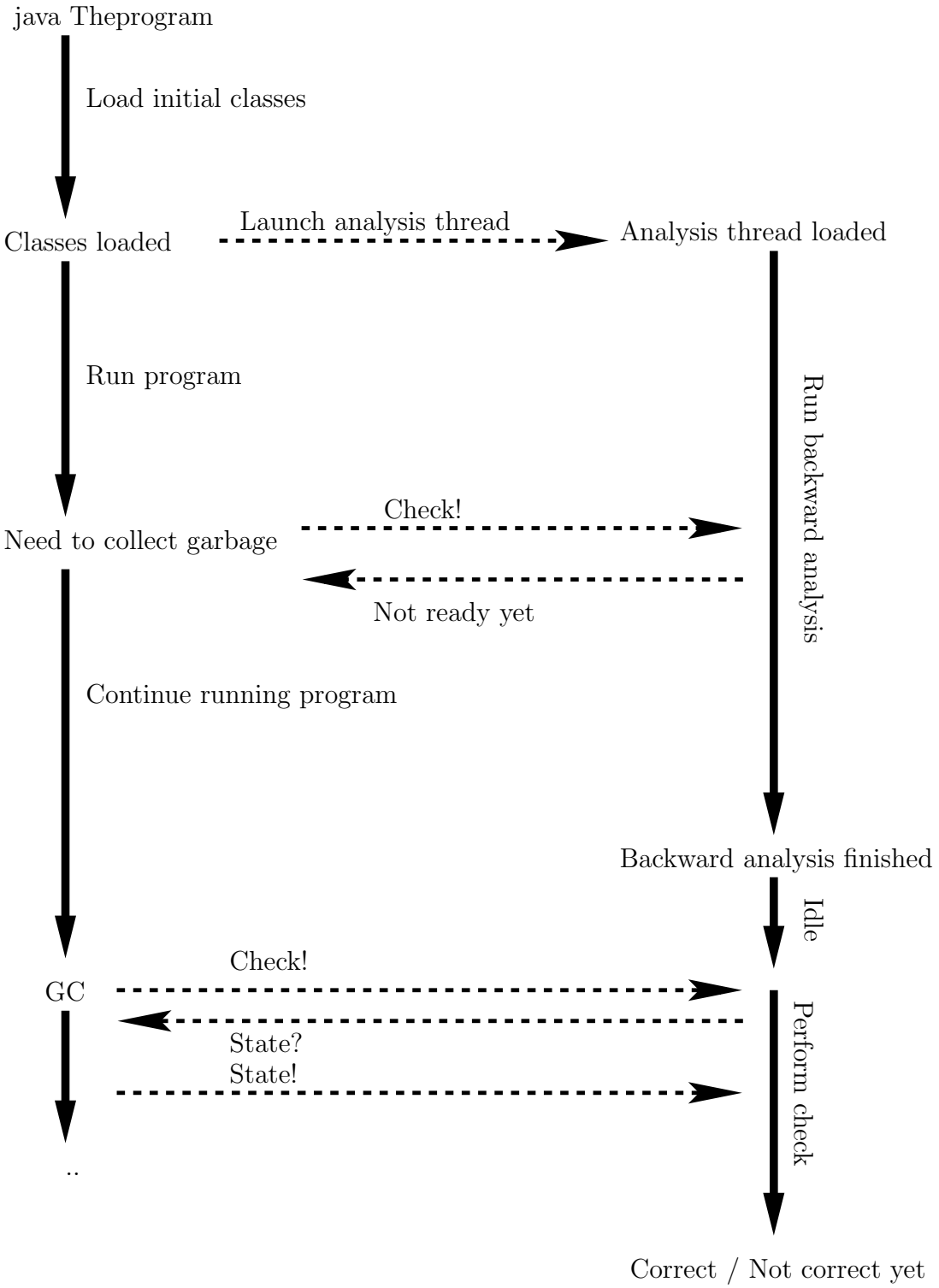


Figure 5-10: How the VM interacts with the analysis

```
proc update_formulas(mapping : Var → (Field → ℤ)) =
    let forward_facts = forward_analysis(mapping)
    for l in CFG labels do
        let φ = Fact_Map(l)
        let φ' = partially_evaluate(φ, forward_facts(l))
        if φ ≠ φ' then Fact_Map(l) = φ'
    done
proc partially_evaluate(φ : formula, m : Var × (Field → C)) =
    φ match
        case x.f :
            let c1 = ⨆_{s∈m(x)} s(f)
            let c2 = ⨅_{s∈m(x)} s(f)
            let c = c1 ⊔ c2
            if c ≠ ⊥ ∧ c ≠? then c
            else x.f
        otherwise :
            partially evaluate subexpressions
            then simplify
```

Figure 5-11: Partially evaluating formulas for speedier checks

- Evaluate inequalities over constants to truth values
- Evaluate formulas

As an example, take the formula $x.f \geq y.g + y.h \Rightarrow z.f \neq 0$. Say using constant propagation, we obtain $z.f = 0$. In a first step we obtain $x.f \geq y.g + y.h \Rightarrow 0 \neq 0$. $0 \neq 0$ evaluates to $\bot$, so we have $x.f \geq y.g + y.h \Rightarrow \bot$, which we can further simplify to $\neg(x.f \geq y.g + y.h)$, finally $x.f < y.g + y.h$ is the partially evaluated error condition.

Due to the inherent nature of the backward analysis building up error conditions from the bottom, we cannot simply insert the constant information and propagate it using backward analysis. This does not work because error conditions that do not specify that values are constant will always be more general, so the propagated information will always imply the old information and no update will be done.

However, if we erase the error conditions where more constant information has become available, and then redo the analysis over these parts, we can improve the results.

Chapter 6

Forward analysis

This chapter shows how we can obtain a representation of the program state from the virtual machine. We then present a way to propagate the information about constants over the rest of the program. This information can later be used to simplify formulas obtained with backward analysis, as explained in chapter 5.5.

6.1 Obtaining state from the runtime

The first task at hand to propagate program state is to actually obtain it. The program state of a Java program is a collection of one or multiple thread. For simplicity's sake, let us only look at one thread. This thread contains its own stack, and each method call allocates a stack frame. This stack frame stores everything that is associated with the current method call. In our case, we are interested in the local variables, the operand stack (containing intermediary results) and the program counter indicating where we are in the execution.

Figure 6-1 shows an example of such a thread. It contains three stack frames, the one for the main method, which called the method m, which in turn called the method n.

If these local variables are reference variables, they might point to heap objects. What we want to do now is to obtain a representation that is suitable for analysis and also as close as possible to a flat heap.

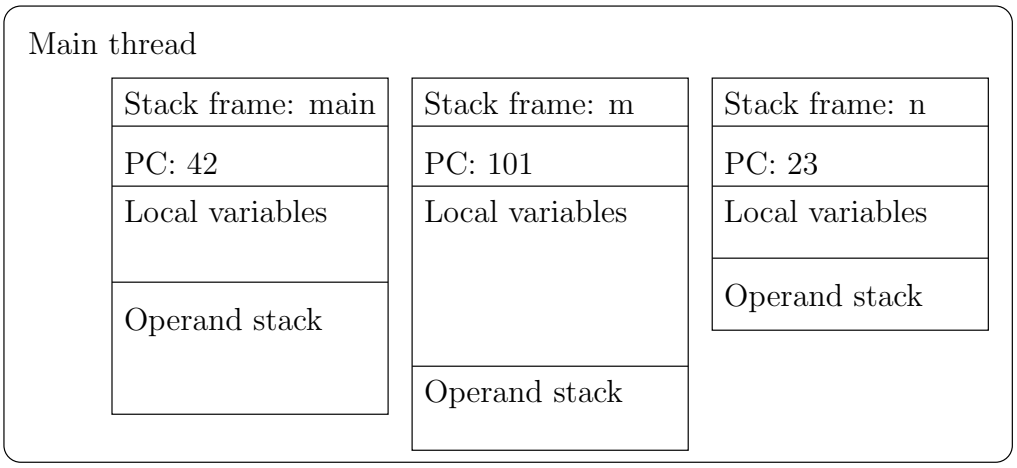


Figure 6-1: An example thread with stack frames

For this, we split the representation in two parts: *Stores* containing the content of the actual first-level objects and a mapping from variables to those stores.

Each store s_i can be thought of as a function from fields to values. For simplicity's sake, we assume that values are integers.

$$s_i : Fields \rightarrow \mathbb{Z}$$

The current state now also consists of a function $M : Vars \rightarrow (Fields \rightarrow \mathbb{Z})$. Now we have to be careful here: two stores might contain the same mapping from fields to values but represent different objects in the heap. We can mitigate this problem by assuming that each store contains an additional field, ref that has a unique value.

$$M : Vars \rightarrow (Fields \rightarrow \mathbb{Z})$$

Algorithm 6-2 shows us how to obtain stores.

6.1.1 Flat heap

As you have seen in the algorithm for obtaining stores, we will only be using the first level of the heap for our forward analysis, but our program deals of course with a more complex heap with multiple levels of reference. So we have to safely approximate these levels. Three things can happen:

```
proc Get_Current_Stores(stackframe : Var → Int,
                        heap : Int → (Fields → Int)) =
    let S = ∅
    let mapping = ∅
    let sf = ∅
    for (v, r) in stackframe do
        if v is a primitive value then
            sf = sf ∪ (v, r)
        else
            if ¬∃s ∈ S such that s.ref = r then
                let thes = heap(r)[ref → r]
                S = S ∪ thes
            fi
            let s be the store ∈ S such that s.ref = r
            mapping = mapping ∪ (v, s)
        fi
        mapping = mapping ∪ ("sf", sf)
    done
```

Figure 6-2: Algorithm: Obtaining stores from a stack frame

- The field points to the null pointer
- The field points to the same heap object as another variable
- The field points to some other heap object

To offset the loss of precision that we obtain by using a one-level heap, we have to introduce nondeterminism. In a method with reference variables x and y, the assignment y = x.f can mean one of the following things:

- y := y
- y := x
- y := NULLVAR
- y := new()

If we have to take all these cases into account, then our program becomes far more difficult. The scenario where multiple variables may point to the same object is called

the Aliasing problem. Landi and Ryder [11] present a good overview of the problem. There are various ways to reduce the number of choices we have for assignment[9], but this is not the section to talk about such matters. The analysis we perform will work well enough if we approximate field assignments by the sole instruction y := new().

6.2 Constant propagation over a flat heap

The main part of our analysis is when we propagate information backward. The forward constant propagation that we show here is primarily to give us flexibility in where we evaluate the analysis. The last section has shown us how to obtain the current program state and represent it as stores. We will combine these exact stores with our backward analysis later on, but for the moment, you can see the constant propagation that we introduce as a way to generalize the information that we obtained over all the program.

6.2.1 Calculating a fixpoint

Let us model the constant propagation in the standard manner of abstract interpretation [7]. We will

- Define an abstract representation of the program state
- Find a suitable order on this representation
- Define the abstract interpretation of the program instructions
- Calculate a fixpoint over the program

The abstract representation of the program state is very close to what we have obtained in the last section: It will consist of a mapping and a number of stores.

6.2.2 Abstract stores

The stores were first defined as a function from fields to integers. For our analysis, we will need further information about these fields. A field may

- not already be initialized
- definitely contain the constant c
- not be constant

We therefore use an abstract store, mapping from fields to $\mathcal{C} = \mathbb{Z} \cup \{?\} \cup \{\bot\}$ where ? represents the fact that a field is not constant and $\bot$ stands for a field whose value we don't know yet.

$$s'_i : Fields \to \mathcal{C}$$

we can already define the ordering $\sqsubseteq_C$ on this domain $\mathcal{C}$: A lattice ordering has to be a partial order (meaning $a \sqsubseteq a$, $a \sqsubseteq b \wedge b \sqsubseteq a \Leftrightarrow a = b$ and $a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$), and every two elements of the lattice have to have a unique least upper bound $a \sqcup b$ and greatest lower bound $a \sqcap b$. Already in the original paper, such a lattice on constants was presented, with the following definitions for the order:

$$\bot \sqsubseteq x \text{ for any } x$$
$$x \sqsubseteq ? \text{ for any } x$$
$$c_1 \sqsubseteq c_2 \text{ if } c_1 = c_2$$

and the following definition for $\sqcup$.

$$\begin{aligned} \bot \sqcup x &= x \\ ? \sqcup x &= ? \\ x \sqcup y &= x \text{ if } x = y \\ x \sqcup y &= ? \text{ otherwise} \end{aligned}$$

With this constant lattice in place, we can now describe an ordering on the stores. To see how such an ordering could work, let's see what we are *not* interested in:

we cannot make an order of stores representing different heap objects, we are only interested in the evolution of individual stores over the control flow graph.

So if two stores represent the same heap object, we perform the ordering according to what is known about the fields:

$$s_1 \sqsubseteq s_2 \Leftrightarrow s_1(ref) = s_2(ref) \wedge \forall f \in Fields \;.\; s_1(f) \sqsubseteq s_2(f)$$

The ordering of stores is therefore a product lattice of the lattices representing individual fields. This product lattice is a lattice itself, so we can still use it to calculate a fixpoint.

6.2.3 Abstract mapping

When we were obtaining the stores from the current state, we were mapping the variables to the current store with a function. However, different program paths might assign different stores to variables in the future. We therefore extend the concrete mapping to an abstract mapping by dealing with a relation instead of a function.

$$M' : Vars \times (Fields \rightarrow \mathcal{C})$$

Again, we have to define an order on the elements. What does it mean that relation M_1 is smaller than M_2? In our context of constant propagation, smaller means more restrictions on where variables point to. As an example, at the beginning of a program, z may point to the store s_1, but progressing in the program, it might also point to the stores s_2 and s_3. In the latter case, there is less certainty about z, a position higher up in the lattice. The relation, however, is not the only determining factor of the order on the mappings, we also have to take into account how the stores themselves relate to each other. Put together, we obtain:

$$M \sqsubseteq N \Leftrightarrow \forall x \in Vars, s \in Stores.\; x\; M\; s \rightarrow \exists s' \in Stores.x\; N\; s' \wedge s \sqsubseteq s'$$

The complete definition of $\sqcup$ is now a bit complicated:

$$(M \sqcup N) = \left\{ (x, s) \;\middle|\; \begin{array}{l} (x,t) \in M \vee (x,u) \in N \wedge \\ t = \bot \text{ if } (x,t) \notin M \wedge \\ u = \bot \text{ if } (x,u) \notin N \wedge \\ t.ref = u.ref \text{ otherwise} \wedge \\ s = t \sqcup u \end{array} \right\}$$

This just expresses that if there are both stores pointed to by the same variable that have the same reference, we have to join those stores.

With the abstract mapping in place, we will now be able to define the rest of the analysis quite easily.

6.2.4 Abstract interpretation

With the lattices in place, we now have to define how each instruction of our intermediary language that might occur in the control flow graph affects the information.

The flow going forward through the control flow graph, the analysis will work in the following way: for each incoming edge into the current instruction, we will obtain the current value of the mapping and the current value of the states. We will then compute the join (least upper bound) over all these values and will obtain the input mapping M. Using this value,

$[\![x.f := expr]\!]_{\mathcal{F}}$	M' such that for all variables $y \neq x$, $yMs \Leftrightarrow yM's$. For all s such that xMs, we get store s'. For all fields $g \neq f$, $s'(g) = s(g)$. For field f, $s'(f) = [\![expr]\!]_{\mathcal{E}}$
$[\![x := y]\!]_{\mathcal{F}}$	M' such that for all variables $z \neq x, z \neq y$, $zMs \Leftrightarrow zM's$. For x, $xM's$ if xMs or yMs. There is no s such that $yM's$.
$[\![x := new()]\!]_{\mathcal{F}}$	M' such that for all variables $z \neq x$, $zMs \Leftrightarrow zM's$. Let t denote an empty store. Let l be a unique number associated with this instruction and $t.ref = l$. Then $xM't$.
$[\![x.f := input()]\!]_{\mathcal{F}}$	M' such that for all $y \neq x$, $yMs \Leftrightarrow yM's$. For all s such that xMs, we have t with $t(g) = s(g)$ for all $g \neq f$. $t(f) =?$ and $xM't$.
$[\![invoke(M, a_1, \ldots, a_n)]\!]_{\mathcal{F}}$	Let S be the stores $\{s \mid xMs\}$ where x occurs as a parameter to the method call. Then $xM's$ if xMs and $s \notin S$. If $s \in S$, then $xM't$, where $t.ref = s.ref \wedge t(f) =?$ - the store might be changed, so we cannot say anything about it any more.

$M' = M$ for all other expressions.

We evaluate expressions as we would with normal variables. If, however, a value is not known, then the whole expression evaluates to ?.

6.2.5 Performing the analysis

With the abstract interpretation fixed, we can now almost start the analysis. What is missing, however, are sensible initial values. The analysis that we are going to perform calculates the fixpoint by going "upwards" from a bottom value. For our mapping, a bottom value would be the empty relation. It is easy to check that this relation is smaller than all other relations.

For the analysis to work, this is another requirement: That the lattice we are working on has a smallest element $\perp$. For our mapping lattice, this comes down to

$$\neg\exists x\ .\exists s\ .\ x\perp s$$

One point will receive a different initial value, and that is the intermediary language instruction that corresponds to the instruction currently executed. This state obtains the mapping representing the current state.

Chapter 7

Alternative: polyhedra for error condition

In chapter 5, we have seen how we can use logic formulas over inequalities of integers to collect error conditions. In this chapter, we will see an alternative to this approach, which, while being less flexible to express error conditions, offers better behavior when it comes to running time and can handle loops more naturally.

Instead of formulas, we use polyhedra. A polyhedron is a disjunction of linear inequalities. If we have m variables and $\Phi(i)$ different possible fields for variable x_i, the polyhedron P consists of n linear inequalities

$$P = \bigwedge_{0 \leq i \leq n} \sum_{j=1}^{m} \sum_{k=1}^{\Phi(i)} a_{j,k} \cdot x_j.f_k \leq c_i$$

The use of polyhedra to discover relations among variables is almost as old as abstract interpretation, it was first introduced by Cousot and Halbwachs [8]. Much research has been going on to provide efficient libraries [4] and more lightweight abstractions like Octagons [13].

In this chapter we will look at polyhedra from a very high level, leaving out many of the messy details involved in obtaining a fast analysis.

7.1 Adapting the backward analysis to polyhedra

Representing error conditions as formulas is, while pretty precise, also connected to some problems that we would like to get rid of. The first one is that we are required to use a theorem prover to even decide on the order of formulas (which was defined by implication). A second problem is that two formulas might be syntactically very different and still represent the exact same error condition. As an example, take the formulas $x.f \cdot (x.f - 1) = 0$ and $x.f = 1 \vee x.f = 0$. This lack of standard form means that checking things like equality become very difficult.

With a polyhedron, we have such a standard form. Let us see how the constructs are evaluated by constructing the lattice of polyhedra for error conditions.

The first step is to define the domain again. As we have seen, a polyhedron is a conjunction of linear inequalities. These linear inequalities are on all fields of variables, exactly as with formulas.

The bottom value $\bot$ of formulas was $false$, the formula that could never be satisfied. The bottom error polyhedron is the infeasible polyhedron. For this polyhedron, there is no point (meaning valuation of the fields) that lies in it. Conversely, the top value is the polyhedron giving no restriction on any field.

An error condition P is smaller than error condition Q if every valuation of the fields that is in P is also in Q, or put otherwise, $\sqsubseteq$ is simply $\subseteq$.

Using this definition, it follows that we should use set disjunction $\cup$ for the least upper bound. This is not enough, however: Say you have the polyhedron $P : x.f \leq 5$ and $Q : 7 \leq x.f \leq 10$. The set disjunction would give $x.f \leq 5 \vee f \leq x.f \leq 10$. But this is not representable as a polyhedron, which must be convex (meaning that any two points in it can be connected by a line that also lies in the polygon).

One way to extend a set to be convex is to obtain its convex hull. What the convex hull does is that if a line between two points is not yet in the set, it adds that line. This is exactly what we will use. So $P \sqcup Q$ means $convex - hull(P \cup Q)$.

Let us now look at the abstract interpretation of the elements. Let P denote the error polyhedron after the instruction.

$[\![x.f := \epsilon]\!]_{\mathcal{P}}$	If ϵ is a linear expression, then $P' = P[x.f \to \epsilon]$. If it is not a linear expression, then $P' = P[x.f \to \nu]$ where ν is an unbound variable.
$[\![x := y]\!]_{\mathcal{P}}$	For all fields $x.f$ used in inequalities in polyhedron P, $P' = P[x.f \to y.f]$.
$[\![x := new()]\!]_{\mathcal{P}}$	For all fields $x.f$ used in the polyhedron and unbound variables $\nu_1 \dots \nu_n$, $P' = P[x.i \to \nu_i]$
$[\![x.f := input()]\!]_{\mathcal{P}}$	$P' = P[x.f \to \nu]$
$[\![[\phi]]\!]_{\mathcal{P}}$	If ϕ is expressible as polyhedron Q, $P' = P \cap Q$. $P' = P$ otherwise
$[\![\{\phi\}]\!]_{\mathcal{P}}$	If $\neg\phi$ is expressible as polyhedron Q, $P' = convex - hull(P \cup Q)$, Otherwise, $P' = \top$.

Joining different paths now requires to find the convex hull between error polyhedra. Already with these rules, we can obtain error conditions for loop free programs. As with loops, we again have to be a bit more ingenious.

7.2 Polyhedra and loops

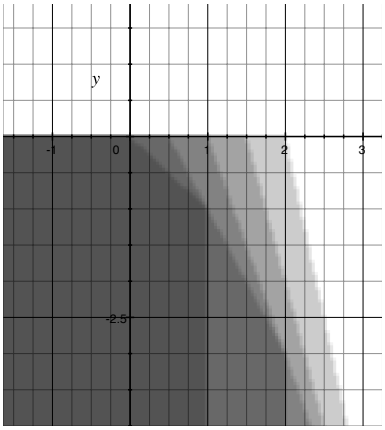
To see what happens with error polyhedra of loops, let's take a look at the sum example again that was given by figure 5-8. We start at the exit with an error condition of $x.sum \leq 0$ (for simplicity's sake we currently ignore the strict inequality).

$$x.i \leq 1 \wedge x.sum + x.i \leq 0$$

The second time around, we get

$$x.i \leq 2 \wedge x.sum + x.i \leq 0 \wedge x.sum + 2x.i \leq 1$$

While this is already a bit cleaner than what we obtained by using formulas (because we had to apply the convex hull for disjunctions), we can still go around the



x-axis: x.i. y-axis: x.sum

Figure 7-1: Successive error polyhedra for the sum program

loop a long time before we finally converge (the first few iterations can be seen in figure 7-1).

Like with formulas, we would like to overshoot to finally arrive at the polyhedron that includes all the error polyhedra generated by no matter how many iterations, $x.sum \leq 0$. In static analysis, this is done by a widening operator ∇. The rules for the widening operator were laid out in the original abstract interpretation paper [7]. Applied to the error polyhedron, this means that $P \sqcup Q \subseteq P \nabla Q$ (the error condition grows bigger when we apply the widening operator than when we just apply join) and that there is not an infinite strictly increasing sequence of $s_0 = C_0, \ldots s_n = s_{n-1} \nabla C_n, \ldots$ for every chain $C_0 \subseteq C_1 \subseteq \ldots \subseteq C_n \ldots$. This condition assures that if we apply the widening after the convex hull when dealing with loops that we will arrive at a loop error condition after a finite number of iterations. Bagnara et al. [3] give a good overview of what widening techniques can be used with polyhedra.

7.3 Checking error conditions with current state

Let's suppose we obtain the program state the same way we did for formulas. We can then express this state as a polyhedron S which only consists of equalities of the

form $x.f = c$. Checking whether errors can still be reached then amounts to checking whether the conjunction with the error polyhedron E, $S \cap E$ has feasible solutions.

7.4 Problems and advantages

The principal advantage of using polyhedra to represent the error condition is that we can use all the techniques that were used for the polyhedron domain in the last thirty years. While working with polyhedra can still be very costly, algorithms exist for combining solutions, checking feasibility and other operations that are needed. If we use formulas, it depends on the structure of the formula and the abilities of the theorem prover whether we can obtain sensible results.

Also, if we use a single polyhedron, we obtain too big error conditions very early on: Consider only the assertion $x.f = 0$. Its error condition is $x.f \neq 0$, which is already not convex. To mitigate this problem and allow a bit of non-convexity, we have to keep multiple polyhedra around, only merging them when their number becomes too big.

Nothing prevents us, however, from using polyhedra to represent a part of the error condition and general formulas for the rest.

Chapter 8

Implementation results

The techniques discussed in the last few chapters have been implemented as an extension to the Java PathFinder virtual machine. In this chapter, we will see the outline of the architecture, followed by a run of the example web server. Finally, we explore what types of programs can benefit from an analysis by our extension.

8.1 The rddsa extension architecture

The techniques outlined in the last few chapters have been implemented in the RDDSA extension to the Java PathFinder [2] virtual machine and model checker. This extension is loaded together with the Java Virtual Machine and thus gains access to the virtual machine state.

To build the control flow graph over which the extension performs the analysis, we use the Soot framework, more precisely its Jimple intermediary representation. This intermediary representation is then accessed via a bridge from Soot to Scala. The extension then translates the code to the simpler intermediary language and performs the analysis. To check the error conditions, the extension calls the CVC3 theorem prover. You can see an overview of the architecture in figure 8-1.

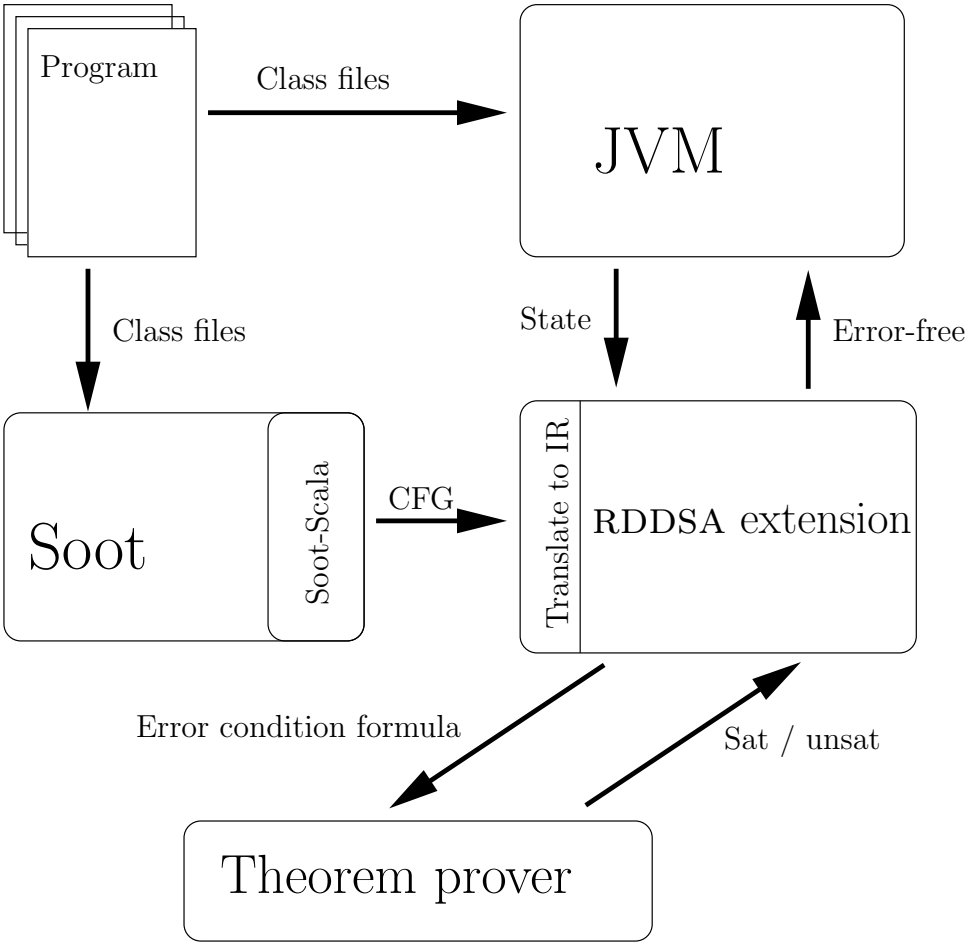


Figure 8-1: RDDSA system architecture

8.2 A typical run

In the beginning of this thesis, we have seen an example web server, whose code was given in figure 2-1. First off, we write a configuration file where we disable safe mode. We run the web server inside the Java PathFinder virtual machine, which then starts the analysis.

We need some way to monitor the status of the application, so that we can tell whether the application is still prone to crash. The RDDSA extension provides a visual indicator in the form of a button: If the button is red, then errors might still occur in the current program, but when it turns green, then the current execution will not crash anymore.

After some time, the analysis comes up with the following main loop error condition:

$$
\begin{aligned}
&(\exists ?r.f.(?r.f \neq 0 \wedge \exists ?r\_other.f.(?r\_other.f = 0 \wedge s.safe\_mode \neq 1) \wedge s.safe\_mode \neq 1) \vee \\
&\exists ?r.f.(s.safe\_mode \neq 1 \wedge \exists ?r\_other.f.(?r\_other.f \neq 0 \wedge \exists ?r\_other\_other.f. \\
&(?r\_other\_other.f = 0 \wedge s.safe\_mode \neq 1) \wedge s.safe\_mode \neq 1) \wedge ?r.f \neq 0) \vee \\
&\exists ?r.f.(s.safe\_mode \neq 1 \wedge \exists ?r\_other.f.(s.safe\_mode \neq 1 \wedge \exists ?r\_other\_other.f. \\
&(?r\_other\_other.f \neq 0 \wedge \exists ?r\_other\_other\_other.f. \\
&(?r\_other\_other\_other.f = 0 \wedge s.safe\_mode \neq 1) \wedge s.safe\_mode \neq 1) \wedge \\
&?r\_other.f \neq 0) \wedge ?r.f \neq 0))
\end{aligned}
$$

which really boils down to $s.safe\_mode \neq 1 \wedge \exists r\_f.r\_f = 0$. Because we have set the safe mode to 0, this error condition can still be satisfied. You can see, however, that if we set safe mode to 1, the error condition becomes unsatisfiable. By changing the web server and running it again, we obtain the green button indicating us that the server will not crash from now on (see figure 8-2).

8.3 Evaluation

We tested the algorithm on a selection of small examples.

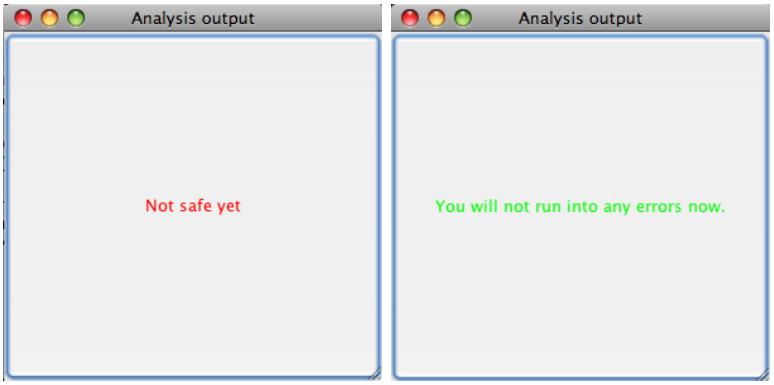


Figure 8-2: Analysis results

Web Server The web server that we presented as introductory example

McCarthy91 Is a nontrivial recursive program that returns 91 for $n \leq 101$.

We found the error condition $n > 100$ for the assertion $n == 91$, which is the best we could expect with induction-iteration.

NumberGuess Asserting properties about numbers that were guessed by binary search

In this example, the theorem prover was not able to prove that a number that was found via binary search was positive. However, it deduced the property before the last guess was made. This shows that even if a property cannot be proved by the theorem prover, a later state might still provide guarantees.

Collatz Shows that unrelated error conditions get propagated as well as error conditions depending on loop conditions.

Noninteractive Performs a lengthy nontrivial calculation. Here, it can be established early on that the program will not crash, but the calculation takes longer.

8.4 Analyzable programs

We tested the RDDSA extension on various programs with and without interaction and with or without loops. We found out that these techniques worked best on

long-running programs that may be interactive. Due to the flat heap representation, however, even simple assertions on the structure of the heap cannot be guaranteed to hold. There is, however, the possibility to split up the error condition in different parts: Say your error condition was $o.next = o \vee o.content > 0$. We could then split up the error condition in the part that was analysable and the part that isn't. As soon as we can say that the simple part of the error condition is not satisfiable anymore, we can then launch a second, more complete analysis checking the more complex properties.

Chapter 9

Related work

While the ideas and algorithms I used for obtaining error conditions have been around for a long time, there is still much research going on when it comes to finding weakest preconditions, so the first chapter will deal with the relations of my work to this. Only with the increase in computing power and the development of multi-core processors, dynamically checking program properties became feasible. The last section will outline related work there.

9.1 Relation to weakest precondition inference

The whole backward analysis that we did was based on propagating error conditions upwards. Normally, the opposite is done: we propagate a formula that expresses the allowed states. I have already mentioned how Xu and Reps [20] use a similar technique to infer formulas for loops in programs given as machine code.

To infer weakest preconditions, more work can be offloaded to the theorem prover than we do here. Gulwani, Srivastava and Venkatesan [10] encode the formulas at different program cut-points and solve for the weakest liberal precondition at method entry.

With their Snugglebug tool, Chandra, Fink and Sridharan [6] presented techniques for efficient call graph generation by initially skipping over calls and then inserting method calls according to the constraints obtained. They also presented generalization

of procedure summaries in which they separate the part of the formula that is actually changed by the method from the rest.

Because we used a very simple heap structure, we did not have to resort to modeling field accesses specially. Normally, this can be done by using uninterpreted functions. The paper on Invariant synthesis for combined theories [5] explores how to find such invariants if uninterpreted functions are involved.

The idea of combining concrete execution with forward symbolic execution was also used in a model checking context (also using Java PathFinder) to generate test cases by Păsăreanu et al [14].

9.2 Relation to dynamically deployed analysis

I already used current program state together with a heap structure analysis based on rules associated with bytecode instructions. This led to the specialization project Dynamically Deployed Static Analysis in Java [9], which provided many of the questions that this thesis answers. While working on heap structure, I could not reuse results of the analysis for later checks.

CrystalBall [21] uses dynamically obtained state of a distributed system together with state exploration to predict future errors. Because distributed system allow for multiple legal executions, this information is then used to steer execution away from these errors.

Chapter 10

Conclusions

In the preceding pages, you have seen techniques to combine program analysis (namely backward propagation of error conditions) with program state. These techniques were then implemented in a tool that can sometimes guarantee absence of certain errors for individual program runs while it is not possible to prove this absence for all executions.

10.1 Future work

The technique that we presented worked very well with the nice little examples, but real big programs break many of our assumptions. Multiple threads may share information in static class fields, exceptions get thrown and later catched when consistency properties of linked data structures are not met. So there is work ahead specifying the analysis for all these cases, with the goal that in the end, we can give certain guarantees about most of the programs that we run. One first step in this direction is to see how we can integrate reasoning about linked data structures, using the results that we have obtained with forward analysis in the past work.

A second interesting problem is how to use the information that we have obtained. Assuring the user of correctness of a run can in itself be useful, but we could also use the information to make future execution faster (by eliminating dynamic checks that we have now proved to hold) and use it in just-in-time compilation.

Finally, we could still drastically optimize the analysis, which would render it practical to run it always in parallel with programs.

Bibliography

[1] Cvc3 home page, http://www.cs.nyu.edu/acsys/cvc3/. 26

[2] Java pathfinder homepage, http://javapathfinder.sourceforge.net. 2, 51

[3] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. *Sci. Comput. Program.*, 58(1-2):28–56, 2005. 48

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008. 45

[5] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Proceedings of the Eighth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007, Nice, January 14-16)*, LNCS 4349, pages 378–394. Springer-Verlag, Berlin, 2007. 58

[6] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. *SIGPLAN Not.*, 44(6):363–374, 2009. 57

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. 2, 38, 48

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY. 45

[9] Sebastian Gfeller. Dynamically deployed static analysis for java. Specialization project report, January 2009. 38, 58

[10] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI '09: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 120–135, Berlin, Heidelberg, 2009. Springer-Verlag. 57

[11] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, New York, NY, USA, 1991. ACM. 38

[12] Ondřej Lhoták. Soot-scala interface. Personal communication, May 2009. 13

[13] Antoine Miné. The octagon abstract domain. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 310, Washington, DC, USA, 2001. IEEE Computer Society. 45

[14] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26, New York, NY, USA, 2008. ACM. 58

[15] Sun. The java^{TM}virtual machine specification. 13, 14

[16] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 264–280, New York, NY, USA, 2000. ACM. 28

[17] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 132–143, New York, NY, USA, 1977. ACM. 25

[18] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999. 13

[19] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998. 13

[20] Zhichen Xu, Barton P. Miller, and Thomas W. Reps. Safety checking of machine code. In *PLDI*, pages 70–82, 2000. 25, 57

[21] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. Technical report, 2008. 58