Static Detection of Inevitable Errors

ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

Dissertation zur Erlangung des Doktorgrades der technischen Fakultät der Albert-Ludwigs-Universität Freiburg im Breisgau
von

Martin Schäf

Dekan
Prof. Dr. Bernd Becker

Gutachter
Prof. Dr. Andreas Podelski
Prof. Dr. Peter Thiemann

Disputation
27.07.2011

Zusammenfassung Statische Programmanalyse ist eine Technik, um bestimmte Eigenschaften eines Programms zu beweisen ohne dieses auszuführen. Implementierungen von statischer Analyse zeigen auf einer Abstraktion eines Programms, dass kein Zustand erreicht werden kann, der die betrachtete Eigenschaft verletzt. Die Abstraktion entsteht dabei durch eine Überapproximation der möglichen Ausführungen des Programms. Ist in der Abstraktion kein ungewünschter Zustand erreichbar, kann die statische Analyse garantieren, dass auch in dem ursprünglichen Programm die Eigenschaft nicht verletzt ist. Obwohl die Analyse formal zum Nachweis der Korrektheit eines Programms konzipiert ist, wird sie in der Praxis vor allem zum Finden von Fehlern eingesetzt. Statische Analyse erlaubt das Finden von Fehlern, die durch Testen nur schwierig zu finden sind.

Statische Programmanalyse zur Fehlersuche zu verwenden, hat den Nachteil, dass auch falsche Warnungen ausgegeben werden. Eine falsche Warnung tritt dann auf, wenn ein Fehler in der Abstraktion des Programms, nicht aber in dem ursprünglichen Programm gefunden wird. Falsche Warnungen entstehen durch den Präzisionsverlust bei der Abstraktion. Eine falsche Warnung kann eliminiert werden, indem man den Präzisionsverlust vermindert, z.B. durch die Angabe von Invarianten, oder durch die Verfeinerung der Abstraktion.

Eine weitere Art von durch statische Analyse gefundenen Fehlern, die als falsche Warnungen angesehen werden, sind Fehler, die auf nicht realistischen Programmeingaben basieren. Statische Analyse zieht alle möglichen Eingaben eines Programms in Betracht. Soll die statische Analyse nur eine Teilmenge der Eingaben betrachten, muss man dies durch eine Vorbedingung spezifizieren. Das Schreiben dieser Vorbedingungen verlangt Erfahrung und ist ein zeitaufwendiger Prozess, da diese Vorbedingungen zu Beginn der Entwicklung nicht bekannt sind und oft verfeinert werden müssen. Beide Arten von falsche Warnungen schränken die Benutzbarkeit von statischer Analyse in der Praxis stark ein.

Dies führt zu der zentralen Frage dieser Doktorarbeit: Ist es möglich, ein statisches Analyseverfahren zu entwickeln, das eine nichtleere Menge von relevanten Fehlern findet, aber nie falsche Warnungen ausgibt (selbst wenn der Programmierer keine Annotation hinzufügt). Der zentrale Beitrag der Doktorarbeit ist die konstruktive Antwort auf diese Herausforderung.

Wir führen den Begriff des *Doomed Program Points* ein. Doomed Pro-

gram Points weisen auf Programmfragmente hin, deren Ausführung notwendigerweise zu einem Programmabsturz führt. Das Auftreten eines Doomed Program Points wird ein Programmierer in keiner Situation als unwesentlichen Fehler abtun können. Der erste Beitrag dieser Doktorarbeit ist zu zeigen, dass der Begriff des Doomed Program Points präzise gefasst und formal definiert werden kann. Wie wir anschließend zeigen, treten Doomed Program Points typischerweise während der Entwicklung der Programme auf. Die Frage ist nun, ob Doomed Program Points erkannt werden können, automatisch und ohne falsche Warnungen zu erzeugen.

Wir stellen ein Verfahren zur statischen Programmanalyse vor, welches Doomed Program Points automatisch und präzise erkennt. Das Verfahren ist zum Einsatz durch einen Programmierer während der Entwicklung gedacht. Es benötigt keine zusätzlichen Angaben durch den Programmierer, weder in Form einer Annotation, die Invarianten spezifiziert ("assume"), noch in Form einer Annotation, die bestimmte Korrektheitskriterien explizit spezifiziert ("assert"). Es ist aber in der Lage, Spezifikationen zu berücksichtigen (d.h., Invarianten auszunutzen bzw. das Programm auf die zusätzlich angegebene Art von Fehlern zu untersuchen). Das Verfahren berechnet eine Garantie in der Form eines formalen Beweises für die Anwesenheit von Doomed Program Points auf einer Abstraktion des gegebenen Programms. Gelingt der Beweis auf der Abstraktion, existiert der Doomed Program Point auch in dem ursprünglichen Programm. Dies bedeutet, dass man die Mächtigkeit der Abstraktion nicht, wie sonst in der Statischen Programmanalyse, für die Garantie der Abwesenheit von Fehlern ausnutzt, sondern deren Anwesenheit. Annotationen können den durch die Abstraktion entstandenen Präzisionsverlust vermindern und dadurch die Anzahl der erkannten Doomed Program Points vergrößern, aber nie verkleinern. Der durch das Verfahren erbrachte Beweis eines Doomed Program Points bezieht sich auf die Gesamtmenge der Eingabewerte des Programms; er bleibt gültig selbst dann, wenn man sie auf eine nichtleere Menge von Eingabewerten einschränkt. D.h., der Programmierer kann einen Doomed Program Point nicht eliminieren (und als unwesentlichen Fehler abtun), indem er einige Eingabewerte ausschließt.

Die Frage ist, ob das oben beschriebene, fundamentale Verfahren für die Praxis anwendbar gemacht werden kann. Wir geben auch hier eine positive Antwort. Wir haben unser Verfahren unter Ausnutzung eines existierenden Frameworks für statische Analyse implementiert. Wir

beschreiben unsere Implementierung und stellen die Optimierungen vor, die wir hierfür entwickelt haben. Wir beschreiben eine vorläufige Evaluierung unserer Implementierung auf typischen Fehlern, die während der Entwicklung eines Programms auftreten. Das von uns vorgestellte Verfahren ist nützlich und so einfach in der Anwendung, dass wir hoffen, dass Programmierer es täglich in ihrer Arbeit benutzen (und sich so für die Verwendung von Verifikationswerkzeugen ködern lassen).

Abstract Static program analysis is a technique to prove properties of a program without executing its code. Tools that implement static analysis show on an abstraction of a given program that no state in this program violates a user provided property. The abstraction results from an over-approximation of the set of possible executions of the program. If no state that violates the desired property can be reached in the abstraction, static analysis can guarantee that this state will also not be reached in the original program. Even though, static analysis methods are meant to prove the correctness of a program, in practice, they are mostly used to detect errors. Static analysis offers powerful support to detect even sophisticated errors that are hard to find using testing.

The drawback of using static analysis for error detection is that they might emit false warnings. A false warning occurs, when a violation of a property is detected in the abstraction of the program that does not occur in the original program. False warnings are caused by the loss of precision during abstraction. False warnings can be eliminated if the programmer manually refines the abstraction by providing additional information (e.g., by providing invariants).

A second kind of errors that is considered a false warnings is errors that occur for unrealistic input values of a program. Static analysis considers all possible input values of a program. If the analysis is supposed to consider only certain input values, the programmer has to specify this by providing preconditions. Specifying preconditions is a time consuming process. Usually, the precondition has to be refined several times, as the scope of a method is not clear in the early stages of development. False warnings of both kinds are a severe limitation to the usability of static analysis.

This motivates the central research question of this thesis: *Is it possible to develop a static analysis that detects a non-empty set of relevant program errors but never reports false warnings (neither false warnings due to abstraction nor false warnings due to weak preconditions)?* The central contribution of this thesis is a constructive answer to this challenge.

We introduce the concept of *doomed program points.* Doomed program points indicate program fragments that inevitably crash on any possible execution of the program. A programmer can, under no circumstances, ignore the presence of a doomed program point. The first contribution of this thesis is that we show that the concept of doomed program points can be formalized. We show that doomed program points oc-

cur frequently during the coding phase of a program. This leads to the question if doomed program points can be detected without producing false warnings.

We present a static analysis that detects doomed program points automatically and precisely, i.e. without emitting false warning. The analysis does neither require user provided information in terms of annotations specifying invariants (“assume”) nor in terms of annotations specifying correctness (“assert”). However, the analysis can make use of specification to detect other kinds of errors. The analysis computes a guarantee in terms of a formal proof for the absence of doomed program points on an abstraction of the given program. That is, the power of abstraction is not used, as usually in static analysis, to guarantee the absence of errors, but to guarantee the presence of errors. Annotations can be used to reduce the loss of precision that is caused by the abstraction and thus help to increase the detection rate. The proof computed by the analysis is valid for any input value of the program and is a valid proof for any non-empty subset of input values. That is, a doomed program point can never be eliminated (i.e., ignored) by excluding some non-realistic input values.

We ask if the above mentioned analysis can be realized efficiently. We give a positive answer and present an implementation that can detect doomed program points without producing false warnings and without the need for user interaction. The implementation works without any user provided information about the pre-state of a method or the invariants of a loop. Yet, it supports specification languages to increase the detection. The implementation is based on existing and established frameworks for static analysis. We present several optimizations for this implementation and show that it is applicable in practice. Doomed program point detection is an easy-to-use but powerful analysis that can help to make the use of specification languages and static verifiers more common in todays software engineering.

Acknowledgements

Foremost I would like to thank my supervisor Andreas Podelski for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. Further I would like to thank him for all the doors he opened for me during and after the time I worked for him.

My biggest thanks go to Thomas Wies and Jochen Hoenicke, who advised through all these years with patience and providence. I'd like to thank my flat mates and colleagues Evren Ermis and Stephan Arlt for the great journey and the precious memories. Further I'd like to thank all my other colleague from Freiburg and in particular the administrative staff Marlis, Berit, and Martin for helping the department to run smoothly and for assisting me in many different ways.

I would like to thank my colleagues from MSR for the discussions we had during the time there and afterwards. In particular I would like to thank Rustan Leino, Mike Barnet, and Monika Roberts for making all this possible.

Last but not the least I would like to thank my family, thank you for the strong support and keeping faith. Thanks to my siblings and for all their help during this time. I would like to thank Viviana Segui for showing me the things in life that happen outside my screen.

Contents

List of Figures

Chapter 1

Introduction

Detecting bugs in software early is crucial to limit the costs of a software project. Most software engineers consider testing as the most common way to detect bugs in software and the best way to increase the confidence in the quality of the source code. But testing and other dynamic analysis require an executable program which is not available in the early stages of a project. Nowadays, many tools analyze source code before it is executable. Static checkers, such as type checkers, data-flow analysis, and static analysis are able to find bugs in source code before it can be executed. Many of these tools are well established and accepted. Even though the precision of these tools is limited according to Rice's Theorem, programmers trust in their output. Static checking of source code is a convenient way to detect bugs that can be applied at any stage of development. However, static checking has to deal with the tradeoff between detection rate and precision. A high detection rate always comes at the price of imprecise results, but lowering the detection rate can never completely avoid imprecision.

One approach to static checking is static program analysis. Static program analysis are techniques to prove properties of a program without executing its code. Tools that implement these techniques show on an abstraction of the given program that no state in this program violates a user provided property. The abstraction refers to the set of possible executions of the program. If no state that violates the desired property can be reached in the abstraction, static analysis can guarantee that this state will also not be reached in the original program. Even though, static analysis methods are meant to prove the correctness of a program, in practice, they are mostly used to detect errors. Static analysis offers powerful support to detect even sophisticated errors that are hard to find using testing.

The drawback of using static analysis for error detection is that they might emit false warnings. A false warning occurs when a violation of a property is detected in the abstraction of the program that does not occur in the original program. False warnings are caused by the loss of precision during abstraction. False warnings can be eliminated if the programmer manually refines the abstraction by providing additional information (e.g., by providing invariants).

A second kind of errors that are considered a false warning are errors that occur for unrealistic input values of a program. Static analysis considers all possible input values of a program. If the analysis is supposed to consider only certain input values, the programmer has to specify this by providing preconditions. Specifying preconditions is a time consuming process. Usually, the precondition has to be refined several times, as the scope of a method is not clear in the early stages of development. False warnings, of either of the two kinds, are a severe limitation to the usability of static analysis.

We hypothesize that a static analysis that is as easy to use and as convenient as, e.g. the definite assignment analysis used in modern compilers, while delivering precise and trustworthy results for a considerable larger class of errors can open a new application area for formal methods. We claim that such a formal method must (1) detect critical errors; (2) never produce false alarms; (3) work without user provided information, such as code annotations; (4) support the use of specification and analyze user provided information that is used by other verification tools.

This motivates the central research question of this thesis: *Is it possible to develop a static analysis that detects a non-empty set of relevant program errors but never reports false warnings (neither due because of imprecision nor because of too weak preconditions)?*

We introduce the concept of *doomed program points*. Doomed program points indicate program fragments that inevitably crash on any possible execution of the program. A programmer can, under no circumstances, ignore the presence of a doomed program point. We illustrate the idea of doomed program points in Section 3 and show that they are important in practice. The first contribution of this thesis is that we show that the concept of doomed program points can be formalized. We give a formal definition of doomed program points in Chapter 4. We show that doomed program points occur frequently during the coding phase of a program. This leads to the question if doomed program points can be detected without producing false warnings.

We present a static analysis that detects doomed program points in Chapter 5. The analysis works automatically and precisely, i.e. without emitting false warning. The analysis does neither require user provided information in terms of annotations specifying invariants ("assume") nor in terms of annotations specifying correctness ("assert"). However, the analysis can make use of specification to detect other kinds of errors. The analysis computes a guarantee in terms of a formal proof for the presence of doomed program points on an abstraction of the given program. That is, the power of abstraction is not used, as usually in static analysis, to guarantee the absence of errors, but to guarantee the presence of errors.

Annotations can be used to reduce the loss of precision that is caused by the abstraction and thus help to increase the detection rate. The analysis first computes an abstraction of the program as shown in Chapter 6. For this abstraction and a given fragment in the original program, we compute a formula whose validity implies that the given fragment is doomed (see Chapter 7). The proof computed by the analysis is valid for any input value of the program and is a valid proof for any non-empty subset of input values. That is, a doomed program point can never be eliminated (i.e., there is no way the programmer can dismiss the indication of a doomed program point) by excluding some non-realistic input values.

We ask if the above mentioned analysis can be realized efficiently. We give a positive answer and present an implementation that can detect doomed program points without producing false warnings and without the need for user interaction. The implementation works without any user provided information about the pre-state of a method or the invariants of a loop. Yet, it supports specification languages to increase the detection rate. The implementation is based on existing and established frameworks. We present several optimizations for this implementation in Chapter 8 and show that it is applicable in practice. In Chapter 9 we give experimental results of a prototype implementation that indicate that this approach is applicable in practice.

In general, there are more efficient ways to detect doomed program points. E.g., writing a test case can reveal any doomed program point. The contribution of this thesis is that we are able to detect errors in (fragments of) source code and that we can *guarantee* that we do not produce false warnings. We also emphasize that the proposed formal method is supposed to be applied before a program is executed and thus, it tackles a different class of bugs then testing.

Given the fact that doomed program points cannot be dismissed by restricting its context and the fact that our algorithm can detect doomed program points fully automatic, we propose to integrate the presented analysis in a software development environment.

Contribution This thesis contributes to the current research in static program analysis in several ways.

- We show that it is possible to formalize a relevant class of errors that can be detected without ever producing a false alarm.
- We show that it is possible to efficiently compute an abstraction of a program that is suitable to detect doomed program points. This abstraction preserves many of the doomed program points of the original program but never introduces new doomed program points.
- We show that the proposed static analysis can be implemented and that it is efficient enough to be applied in practice.

Chapter 2

Related work

Many different implementation of static analysis exist that differ in their complexity, the classes and amount of errors they detect, and their supposed application domain. In this chapter we give a brief overview of the field of static analysis and position our contribution in the field of existing implementations. Further we give a detailed overview of the related work the shares our motivation of finding and reporting critical bugs instead of assuring the absence of bugs.

2.1 Static Program Analysis

We next discuss the landscape of existing static analysis tools and position our approach. Static code analysis tools helping to avoid and detect errors in source code accompany programmers in their daily work and are crucial to software reliability. While all these tools have in common that they check properties of source code without executing it, their aims, abilities, and limitations differ based on their algorithmic foundation.

The most widely used approach is static type checking and data-flow analysis provided by modern compilers (e.g., [1, 21, 33]) . Many compilers reject a program if the type errors or data-flow anomalies such as the use of uninitialized variables are found.

String or pattern matching analysis are usually based on the idea of Lint (e.g. [17]). They perform a context-insensitive analysis to detect potential vulnerabilities in code which makes them scale to very large programs. On the other hand, the analysis is shallow and produces many false alarms while potentially missing real errors.

Dataflow analysis uses semantic information of the program to detect vulnerabilities and errors with relatively small amount of false alarms. Two different approaches can be distinguished in dataflow analysis, both having commercial implementations on the market. Unsound dataflow analysis tools represented, e.g., by Coverity Prevent [11] or Klocwork K7 are able to detect errors in large program with only few false alarms. The good precision and its ability to scale to large program comes with the drawback that these tools cannot give guarantees as they miss real errors and report false positives.

Sound dataflow analysis tools like the Polyspace Verifier [45] can give a guarantee that certain errors do not occur. The price for this is a high false positive rate or the need for providing additional information on the intended programs behavior. A detailed survey on static analysis tools and their abilities and restrictions can be found in [15].

None of the approaches discussed above satisfies our requirement on the high degree of usability: a tool that requires no user interaction because it produces no false positives, neither false positives introduced by abstraction nor false positives introduced by insufficient precision of the specification of user input.

This motivates our work to extend the landscape of static analysis tools by our new approach. One may say that the new approach focuses on increasing usability by decreasing the error detection rate to definite errors only. The guarantee that we do not produce false warnings sets our method apart from existing approaches.

2.2 Static Bug Detection

We first want to point out that the class of errors our approach finds is subsumed by many bug detection tool and that most tools will find even more real bugs. However, the increased error detection rate comes at a price: these tools either produce a lot of noise or they require heavy user interaction (e.g., [15, 42, 46]). For instance, a set of unit tests that executes every statement in the program at least once will detect all errors related to doomed program points but one has to write or generate the test cases.

The core aspect of our algorithm is that, by proving the presence of errors, we do never produce false warnings. The importance of precision in static checking has also been stated by others. Different approaches exist to avoid false warnings or to prove the presence of errors. In the following we contrast the idea of doomed program point detection with other approaches the have a similar motivation.

Findbugs [2] which is partially pattern matching based shares our motivation that it is crucial to keep the number of false positives as low as possible. They achieve good results by focusing on certain errors that can be detected precisely and by giving categories to the reported warnings indicating the criticality of the problem. Even though they produce very good results on large programs they cannot give a guarantee that they find all errors of a certain kind or that none of the reported errors is a false alarm.

The Wasp [44] tool distinguishes between possible and definite errors. Their analysis is implemented on top of OSA. The main difference to our work is that they allow false positives in their definite error detection.

PolySpace verifier [45] that is mentioned already in the previous section has a color coding of detected bugs. It classifies program statements as red faulty if they fail on any execution which essentially means that they are doomed according to our definition. The motivation is similar to ours. Our approach allows checking program fragments in isolation without knowing anything about their reachability.

There are other approaches that prove the presence of errors. E.g., must-analysis [22] can be used to prove the presence of a bug by finding a witness. At this point we are not aware of static must-analysis and thus comparing our static approach to a dynamic approach is out of scope.

Hayes et. al. [23] proposes that a control flow path can be proven inadmissible using the weakest liberal precondition semantics with the postcondition false. From the algorithmic point of view this is exactly what we do but they restrict their analysis to single control flow paths in isolation and therefore conclude that this method is not applicable in practice.

Static verifiers such as Boogie [3, 36] provide a complex infrastructure that is used by our implementation: An intermediate verification language that abstracts away high level programming language constructs, classes to traverse the program, and classes to transform a program into a SMT formula as well as an interface to different decision procedures. Other program verifiers like Why [18] or ESC [19] provide a similar infrastructure and could serve as the basis for an implementation as well.

In [10], Cousot et. al. propose an alternative way for computing preconditions. In particular they define the strongest solution to the precondition inference problem and further its complement which is the set of states from which all runs of a program are bad in that they will end in an error. This corresponds to the idea of reachability of annotated code in [31]. This is a stronger condition than what we propose in the way that we only require that all executions passing a program fragment lead to an error instead of all executions starting in that state.

The results produced by our tool could be reproduced using full fledged automatic verifiers such as BLAST [24] by first trying to prove the program, collecting all unverified assertions, negating them and rerunning the verification. If the verifier is able to prove such a negated assertion then, the corresponding statement will fail under any circumstances. However, this would be a rather convoluted and costly way to find doomed program points. Also, tools such as BLAST are meant to be applied to the whole program, i.e., at a rather late stage of development when the errors we are targeting have probably already been fixed.

Another important problem in verification is the problem of vacuous truth, when verification succeeds even for an erroneous program because of errors in the specification. In model checking, vacuous truth and ways to detect it has been discussed in several articles (e.g., [6], [43]). The experiments in [43] show that up to 20% of the formulas pass vacuously in the first verification run. This can lead to huge problems if it is not detected early by the verification engineer.

There is only little work on the detection of vacuous truth in static program verification. In [31], Janota et. al. present a method to detect dead code in annotated programs. They point out that it is especially used to detect code that is made unreachable by the specification. Their algorithm is implemented in Boogie as well. Verification engineers often simulate this idea if they are suspicious of the proof by adding an assertion that will inevitably fail to the method. If the proof still succeeds it is certain that there is an error in the specification. Throughout this work we will show that the work in [31] is a special case of doomed program points.

Parts of this thesis have already been published as conference paper [26]. The contribution of the paper and many optimizations have been presented in an invited special issue journal [27].

Chapter 3

Motivating Examples

In the following, we present a collection of examples that demonstrate what kinds of errors our approach is able to find and, more importantly, what kinds of potential vulnerabilities it does not report.

Example 1. Our first example is given in Figure 3.1. It demonstrates a trivial, yet common error that can happen during development. There are several examples of errors of this kind, even in published code, e.g., an old version of Eclipse [29] and the BatteryClassUnload method in a reference driver of the Windows Driver Kit contained such errors.

If our algorithm identifies an error in a program, then it will report not just the statement that crashed (the symptom of the error), but also the statements that actually lead to the crash (the cause of the error). This provides additional hints to the developer that help him to fix the error. If we apply our algorithm to the example program, then it will report lines 5 and 6 as a guaranteed error. It reports line 6 because whenever the expression `*ptr` is evaluated, this will cause a null pointer dereference. It further reports line 5 because if the else branch of the conditional is taken, `ptr==0` has been evaluated to true, which guarantees the error in line 6.

Example 2. Our second example is less trivial, yet contains a common error. The procedure `getMin` in Figure 3.2 returns the minimal element of an array. For this purpose, it first sorts the array and then returns the first element. However, there is a mistake in the loop bound of the *for* loop in line 3. The loop will decrease the variable `i` until it has a negative value. This leads to an out-of-

```
1  int access(int *ptr)
2  {
3    if (ptr)
4      *ptr = 0;
5    else
6      printf("%d",*ptr);
7
8    return 0;
9  }
```

Figure 3.1: Program ACCESS

```
1  int getMin(int *a,int x) {
2    int i, j, temp;
3    for (i=x-1; i>=0; i--) {
4      for (j=1; j<=i; j++) {
5        if (a[j-1]>a[j]) {
6          temp = a[j-1];
7          a[j-1] = a[j];
8          a[j] = temp;
9        }
10     }
11   }
12   return a[i];
13 }
```

Figure 3.2: Program GETMIN

```
1  //@ requires x>10
2  void vacuous(int a,int b)
3  {
4    if (x<10) {
5        return 1/0;
6    }
7  }
```

Figure 3.3: Program VACUOUS

```
1  void entangled(int a,
2                 int b)
3  {
4      b=1;
5      if (a>0) b--;
6      b=1/b;
7      if (a<=0)
8        assert b!=1;
9  }
```

Figure 3.4: Program ENTANGLED

bounds array access in line 12. Our algorithm detects that the out-of-bounds access is inevitable. It reports lines 3 and 12 as what leads to the error. This is the only warning emitted by our algorithm. Since there is no precondition saying that array `a` is allocated and its size is given by `x`, any attempt to verify that the procedure is safe without taking into account its calling context would generate additional warnings of potential out-of-bounds errors.

Example 3. In our third example we consider the two programs VACUOUS and ENTANGLED that are shown in Figure 3.3, respectively, Figure 3.4. The two programs have both trivial errors. In the program VACUOUS taken from [31], the precondition in line 1 is a vacuous annotation that excludes all executions that may reach line 5. Thus, the division by zero error in this line would be

```
/* Sorted tree */

typedef void* T;
typedef
  struct entry
  {
    Entry left ;
    Entry right ;
    int key ;
    T data ;
  } *Entry;
```

```
void update(Entry root,
            int k, T d) {
  Entry x = root;
  while (x->key != k) {
    if (k < x->key)
      x = x->left;
    else
      x = x->right;
  }
  x->data = d;
}
```

Figure 3.5: Program UPDATE: The procedure `update` is intended to be called on an existing key

ignored by most static verifiers. Since there is no execution reaching line 5 such vacuous annotations can be detected using a reachability analysis (e.g., [31]). Similar to such an analysis our approach reports on this example that lines 4 and 5 are doomed because there is no normal terminating execution passing these two lines. In general, if a program point is not reached by any execution it is also not passed by executions that terminate normally and it is therefore doomed. However, checking whether a program point is doomed is not the same as checking whether it is not reachable by any execution. In particular, it is not the same as checking whether a program point is both not forward reachable from an initial state and not backward reachable from a potential final state of the program. This is demonstrated in the program ENTANGLED. In this program, any execution starting in a state with $a > 0$ will assign b to 0 in line 5. These executions therefore lead to a division by zero error in line 6. For any other execution, namely those starting in a state where $a \leq 0$ evaluates to true, b remains 1 and therefore the assertion in line 8 is violated. This means, no execution of the program terminates normally and, thus, all program points in the program are doomed. Yet, every program point in this program is either forward reachable or backward reachable by some execution fragment. An algorithm that checks for doomed program points needs to consider at the same time, both, the prefix and the suffix of any execution that reaches a particular program point.

Example 4. Our last example demonstrates how the user of our tool benefits from the fact that it detects guaranteed errors rather than arbitrary errors. The program fragment in Figure 3.5 is taken from a library that implements a map data structure using a sorted binary tree. The procedure `update` takes three parameters: a pointer `root` to the root of the data structure, a key `k` to an entry in the data structure, and a data value `d`. It then traverses the tree to find the entry for the given key and updates the data value associated with this key. The procedure works correctly if the calling context guarantees that there is already an entry for the given key in the data structure, which can be formalized as follows:

$$\exists y. \mathit{child}^*(\mathit{root}, y) \wedge \mathit{key}(y) = k$$

where child^* denotes the reflexive transitive closure of the child relation

$$\mathit{child}(x, y) \Leftrightarrow \mathit{left}(x) = y \vee \mathit{right}(x) = y \ \ .$$

If this precondition is violated, the execution of the procedure will result in a null pointer dereference. Note that there is no null pointer check that guards the dereference of variable `x` in the while condition at line 16. The fact that there is an entry for the given key guarantees that `x` is not null.

It is a real challenge for any error detection tool to prove that line 15 does not cause a null pointer dereference and, thus, not report this line as a potential error. For extended static checking or a modular program verifier, the user needs to specify the precondition saying that there exists an entry in the tree for the given key. However, this is not sufficient to prove the absence of a null pointer dereference. The user further needs to specify a data structure invariant that expresses the fact that the tree is sorted:

$$\forall x.\, \mathit{child}^*(\mathit{root}, x) \Rightarrow \mathit{key}(\mathit{left}(x)) \leq \mathit{key}(x) \wedge \mathit{key}(x) \leq \mathit{key}(\mathit{right}(x))$$

This information is required in the loop invariant of the while loop. Even if all necessary specifications are given, automatically proving that the loop invariant implies the absence of null pointer dereferences is still tricky. Extended static checkers use theorem provers to automate this task. The theorem prover needs to conclude from the sortedness property and the existence of an entry for the given key that this entry is located in the subtree that the while loop traverses into. Modern theorem provers still require proof hints from the user to accomplish such proofs, in particular if it requires reasoning in expressive logical theories containing, e.g., transitive closure operators [30]. All these tasks are time consuming and require the expertise of a verification engineer.

If, on the other hand, one attempts to use abstraction based program analyses to automatically infer the necessary preconditions then only the use of a very sophisticated shape analysis would leave any hope for success. However, such analyses are expensive and do not yet scale well to large programs.

In contrast, our algorithm will not report any errors, simply because there exist executions that never dereference any null pointers.

Chapter 4

Doomed Program Points

We now formally define the new class of problems that we consider in this thesis.

In order to abstract away from the details of a concrete programming language, we only assume that programs are defined over a set of states and that each program P defines a set of *initial states* $\mathit{Init}(P)$ and a set of *executions*, which are possibly infinite sequences of states that start in an initial state. A program further comes with a finite set of *program points*. Each state in an execution belongs to a unique program point. We say that an execution *passes through* a program point ℓ if one of the execution's states belongs to ℓ.

We assume that executions are partitioned into two sets: admissible executions and inadmissible executions. An execution is inadmissible if it has some undesirable behavior. For concreteness, we say an execution is inadmissible if it diverges or violates an assertion. Note that if an execution violates an assertion, it is finite, but is not properly terminating.

Definition 1. *A program point ℓ is called* doomed *if all executions that pass through ℓ are inadmissible.*

In particular, a program point is doomed if there is no execution reaching this program point (i.e., it is part of *dead code*). If a doomed program point is reached then the execution is inadmissible (i.e., it is guaranteed to diverge or to violate an assertion). Thus the existence of a doomed program point is always a witness of a programming error. Note that the doomed program point itself does not necessarily contribute to the error, e.g., in a program with only one execution, if one program point in the execution is doomed then all program points are doomed. Detecting doomed program points and extracting an error message are two separate problems. In this thesis, we are concerned with the detection of doomed program points.

Characterization of doomed program points. Following the idea of Definition 1 we propose a simple program transformation from a program P to a program P^* that allows us to check whether a given program point is passed by an admissible execution. We assume that our language has variables and assignments. For each program point ℓ, we add one auxiliary Boolean variable R_ℓ that is assigned *true* when ℓ is passed. That is, we add the assignment $R_\ell := \mathit{true}$ at each program point ℓ. We therefore call the variables R_ℓ *reachability variables.* Note that the variables R_ℓ are not explicitly initialized, i.e., they can take any Boolean value in an initial state. We further assume that the language has assertions. An assertion has a Boolean expression over program variables. If this expression evaluates to *false* in an execution then the execution is inadmissible. Without loss of generality, we assume that all admissible executions end in a unique program point. At this program point, we add the assertion **assert** $\bigwedge_\ell R_\ell$, which ensures that upon termination all reachability variables R_ℓ are set to *true*.

The transformed program P^* preserves the existence of admissible executions in the original program P because for each admissible execution of P there is always an execution of P^* where all reachability variables R_ℓ are initialized to *true*.

In program P^* we can restrict the set of admissible executions of P to those passing a program point ℓ by initializing R_ℓ to *false*. If ℓ is not passed, R_ℓ is not set to *true* and thus the assertion at the end of P is violated. That is, we can use this transformation to get all admissible executions passing a particular program point.

Using the above transformation we can now give a precise characterization of doomed program points using *weakest liberal preconditions.* The weakest liberal precondition $\mathsf{wlp}(P, S)$ of a program P and a set of states S is defined as usual, i.e., $\mathsf{wlp}(P, S)$ denotes the set of all initial states s, such that all admissible executions of P that start in s terminate in a state $s' \in S$. In particular, if $\mathsf{wlp}(P, \emptyset) = \mathit{Init}(P)$ then the program P has no admissible executions. For a program point ℓ in a program P, let $[\![R_\ell]\!]$ denote the set of all states of P^* where R_ℓ is *true*. Then

$$\ell \text{ is doomed in } P \quad \text{iff} \quad \mathit{Init}(P^*) \subseteq [\![R_\ell]\!] \cup \mathsf{wlp}(P^*, \emptyset) \tag{4.1}$$

Even though condition (4.1) gives a precise characterization of doomed program points, this does not give us a complete algorithm for detecting them, since in general the weakest liberal precondition of a program is not effectively computable. In the rest of this thesis we describe an algorithm for detecting doomed program points that uses a sound approximation of condition (4.1).

Chapter 5

Implementation

We build our analysis as a modification of a program verifier. We assume the standard program verification architecture that translates a source program into an intermediate program representation and then, from that intermediate program, generates verification conditions for an automatic satisfiability-modulo-theories (SMT) solver [3, 18, 19].

The intermediate representation is a mix of mathematical definitions and a simple imperative program notation. It provides a level of abstraction between the source-language semantics and the logical formulas, akin to the way compilers use intermediate representations as a stepping stone toward generating machine code. The translation into the intermediate language makes explicit the behavior and proof obligations stipulated by the semantics of the source language.

We apply our technique at the level of this intermediate verification language. Not only does that let us work with a simpler programming notation, but it also makes our analysis apply to all source languages that translate into the intermediate form.

The verification conditions are first-order logical formulas whose validity encodes the correctness of the intermediate program. They are generated from the intermediate program using a *weakest precondition* computation. The verification conditions are in turn passed to an SMT solver, which attempts to verify or refute them. In more detail, the SMT solver negates the given formula and attempts to satisfy the negation. Failure to do so after an exhaustive exploration implies that the verification condition is valid, which means that the intermediate program is correct and thus that the source program lives up to its proof obligations. On the other hand, if the solver finds a model that seems to satisfy the negation, the model is translated back into an appropriate error message.

In our technique, we modify the interaction between the intermediate language and the SMT solver, so that the solver will in effect be searching for doomed program points in the intermediate program, corresponding to doomed program points in the source language. Let us now go into the details of the intermediate verification language.

5.1 Input Language

The syntax of our simple language is defined in Figure 5.1. A program consists of a sequence of blocks. Each block consists of a unique program point, a sequential statement, and a goto statement that connects the block with a non-empty set of successor blocks. We often identify a block and its associated program point. For a given program, we denote by $st(\ell)$ the sequential statement of the block associated with a program point ℓ. The atomic statements of our language are assignments, non-deterministic assignments (**havoc**) of program variables, assert statements, and assume statements. We do not specify the concrete syntax of expressions that are used in these statements.

$$\begin{aligned}
\mathit{Program} &::= \mathit{Block}^{+} \\
\mathit{Block} &::= \mathit{PPId}: \ \mathit{Stmt}; \ \mathbf{goto}\ \mathit{PPId}^{+} \\
\mathit{Stmt} &::= \mathit{VarId} := \mathit{Expr} \mid \mathbf{havoc}\ \mathit{VarId}^{+} \\
&\mid \mathbf{assert}\ \mathit{Expr} \mid \mathbf{assume}\ \mathit{Expr} \\
&\mid \mathit{Stmt}; \ \mathit{Stmt}
\end{aligned}$$

Figure 5.1: Simple Language

Without loss of generality, we assume that each program or program fragment considered in this thesis has one unique start block ℓ_{init} and a unique end block ℓ_{term}, i.e., each block either has a transition to other blocks or goes to ℓ_{term} which means that the program has terminated normally.

We assume that each program variable x has an associated type $\mathsf{type}(x)$ and that each type t has an associated domain $[\![t]\!]$ comprising the values that a program variable of type t can take. We assume that one of these types is the type bool, which is interpreted as the set of Booleans. We further assume a dedicated program variable pc that models the program counter and evaluates to a program point. A program state s is then a valuation of all program variables, i.e., a function mapping each program variable x to some value $s(x) \in [\![\mathsf{type}(x)]\!]$. The set of initial states $\mathit{Init}(P)$ of a program P is simply the set of all states s of P with $s(\mathsf{pc}) = \ell_{\mathsf{init}}$.

Expressions in our language are built from program variables and additional typed operators that have a fixed interpretation for all programs (e.g., Boolean connectives and arithmetic operations). The program variable pc, however, may not appear in the expressions. We require that all expressions are well-typed, i.e., each expression e has an associated type $\mathsf{type}(e)$ that can be inferred from the types of operators and program variables. We denote by $s(e) \in \mathsf{type}(e)$ the unique value denoted by an expression e in a state s. The value $s(e)$ is computed from the interpretations of operators and program variables in s.

We require that the expressions appearing in **assume** and **assert** statements have type bool. We call expressions of type bool *formulas*. We say that a state s *satisfies* a formula F, denoted by $s \models F$, if F evaluates to *true* in s. We say that F is *valid*, if F is satisfied by all program states.

Each statement st in our language gives rise to a binary transition relation $[\![st]\!]$ on program states that is recursively defined on the structure of statements, as expected:

$$\begin{aligned}
(s,s') \in [\![x := e]\!] &\overset{\text{def}}{\iff} s' = s[x := s(e)] \\
(s,s') \in [\![\textbf{havoc } x]\!] &\overset{\text{def}}{\iff} s' = s[x := v] \text{ for some } v \in [\![\mathsf{type}(x)]\!] \\
(s,s') \in [\![\textbf{assume } F]\!] &\overset{\text{def}}{\iff} s' = s \text{ and } s \models F \\
(s,s') \in [\![\textbf{assert } F]\!] &\overset{\text{def}}{\iff} s' = s \text{ and } s \models F \\
(s,s') \in [\![\textbf{goto } \ell_1,\dots,\ell_n]\!] &\overset{\text{def}}{\iff} s' = s[\mathsf{pc} := \ell_i] \text{ for some } i, 1 \le i \le n \\
(s,s') \in [\![st_1; st_2]\!] &\overset{\text{def}}{\iff} \text{exists } s_0, (s,s_0) \in [\![st_1]\!] \text{ and } (s_0,s') \in [\![st_2]\!]
\end{aligned}$$

A program gives rise to a set of executions. An execution consists of a sequence of states describing the successive execution of the program blocks starting from the initial block of the program. Formally, for a sequence of states π we denote by $\mathsf{len}(\pi)$ the length $n \in \mathbb{N}$ of π if π is finite, and the limit ordinal ω if π is infinite. Furthermore, for i with $0 \le i < \mathsf{len}(\pi)$, we denote by $\pi[i]$ the i-th state of the sequence π and if π is finite, we denote by $\mathsf{final}(\pi)$ its final state $\pi[\mathsf{len}(\pi) - 1]$. An *execution* π is then a sequence of states such that (1)

$\pi[0](\mathsf{pc}) = \ell_{\mathsf{init}}$, (2) for all i with $0 < i < \mathsf{len}(\pi)$, $(\pi[i-1], \pi[i]) \in [\![st(\pi[i-1](\mathsf{pc}))]\!]$, and (3) if π is finite then either $\mathsf{final}(\pi)(\mathsf{pc}) = \ell_{\mathsf{term}}$ or else there is no state s such that $(\mathsf{final}(\pi), s) \in [\![st(\mathsf{final}(\pi)(\mathsf{pc}))]\!]$. An execution π is *admissible* if it terminates normally, i.e., if π is finite and $\mathsf{final}(\pi)(\mathsf{pc}) = \ell_{\mathsf{term}}$. Note that a false assert statement has no continuation, and thus is the end of an execution that reaches that point. Hence, executions leading to false assert statements do not terminate normally.

Note that we can model arrays and the program's heap using function-valued program variables that map indices or memory addresses to values, following a memory model in the style of Burstall and Bornat [7,8]. The concrete representation of the heap depends on the semantics of the translated language. For a detailed discussion of such memory models for concrete programming languages see, e.g., [9,34,36,38].

We do not discuss features like procedure calls in detail because they are either eliminated by inlining the called procedure or abstracted by procedure contracts (see Section 6). A detailed discussion including procedure calls, the type system and other constructs can be found in [40].

In our implementation we use BOOGIE as input language. There are various projects translating from high level source code into BOOGIE, e.g., VCC [9], which translates `C` code to BOOGIE, or SPEC# [5] and B2BPL, which translate from `.NET` or JAVA byte code to BOOGIE, as well as many other translations of special purpose research languages. Even though it is not in scope of this work, it is important to mention that the quality of our results on real world code strongly depends on the translation from a source language to BOOGIE. Our analysis can only discover doomed program points if they are preserved by the translation. As many complex operations are over approximated by the translation, much precision is lost in this step. However, building a translation from a high level language to BOOGIE that is optimized for finding doomed program points is a separate problem and not part of this work.

5.2 Overview of the Detection Algorithm

We now give the outline of our doomed program point detection algorithm. The algorithm is implemented by the procedure Exorcise given in Figure 5.2. Exorcise takes a program as input and returns a set of doomed program points. The procedure first transforms the input program P into a program P' in loop-

```
proc Exorcise($P$ : program)
    var $P'$ : program
    var $\varphi$ : formula
    var $D$ : set of doomed program points
    begin
        $P' := \mathsf{Transform}(P)$
        $\varphi := \mathsf{wlp}(P', \mathit{false})$
        $D := \emptyset$
        for each program point $\ell$ in $P$ do
            if $\mathsf{Valid}((R_\ell = 0) \implies \varphi)$ then
                $D := D \cup \{\ell\}$
            fi
        od
        return $D$
    end
```

Figure 5.2: Algorithm for detecting doomed program points

free passive form using the subroutine Transform. A program is called *loop-free* if it has no cycles in the graph formed by its blocks and goto statements and it is called *passive* if all its blocks consist only of assume and assert statements. The transformation is sound in the sense that if a program point ℓ in P is doomed in P' then it is also doomed in P.

After the transformation, procedure Exorcise computes φ, a logical formula representation of weakest liberal precondition of P' and *false*. Then, Exorcise iterates over all program points ℓ in P and uses the subroutine Valid, which checks if $(R_\ell = 0) \implies \varphi$ and thus, if the program point ℓ is doomed. We assume that Valid is a sound test for logical validity. If the check succeeds ℓ is added to the set of doomed program points. Note that the transformation from P to P' may introduce additional program points, we therefore only check the program points from the original program P.

Chapter 6 gives details on the implementation of the subroutine Transform. In Chapter 7 we show how the formula representation of the weakest liberal precondition is computed and how the validity check Valid is realized. In Chapter 8 we discuss further optimizations of the basic algorithm.

Chapter 6

Program Transformation

The function Transform used in Figure 5.2 transforms a given Boogie program P into a loop-free and passive Boogie program $\mathsf{Transform}(P)$. A program is called *loop-free* if it has no looping control flow and it is called *passive* if all its basic blocks consist only of **assume** and **assert** statements.

The function Transform first eliminates all loops in the input program P. This transformation is described in Section 6.1. Section 6.3 describes the next step in the transformation, which augments the program with the reachability variables. The final step described in Section 6.4 is the transformation into a passive single assignment form. This transformation introduces fresh variables such that each variable in the program is written only once. The size of the resulting program $P' = \mathsf{Transform}(P)$ is linear in the size of the original program P. We will then show that one can construct a formula representing $\mathsf{wlp}(P', \mathit{false})$ whose size is linear in the size of P' and hence linear in the size of the original program P.

The resulting program P' is a sound abstraction of the program P with respect to doomed program point detection. That is, if there exists an admissible execution passing through a program point ℓ in the original program P then there also exists an admissible execution in the transformed program that passes through ℓ. However, in general the transformation loses completeness. This is due to the transformation into loop-free form: we abstract the body of each loop, which might lead to additional admissible executions that are not possible in the original program. Furthermore, the elimination of loops may change the termination behavior of the program, i.e., there might be some inadmissible executions leading to non-termination that are lost because of the transformation. Both the introduction of new admissible executions and the removal of inadmissible executions may cause that some program points that are doomed in P may no longer be doomed in P'. However, the transformation to loop-free form is the only transformation that affects completeness of our algorithm.

Many steps in the elimination of loops and the transformation into passive form are by now standard and used in many extended static checkers and program verifiers (e.g., [3, 19]). We therefore provide only a brief description of the steps known from previous work and focus on the parts that have been customized for doomed program point detection. For a more detailed discussion see, e.g., [4, 20].

6.1 Eliminating Loops

The most common techniques for dealing with loops in extended static checking are: (1) abstraction of the loop by using an inductive loop invariant (e.g., [4]) and (2) finite loop unrolling [19]. Inductive loop invariants provide a sound (and often complete [39]) technique for eliminating loops. However, the loop invariants have to be either user-provided, which makes the analysis more demanding for the user, or computed automatically using static analyses, which makes the analysis more expensive. On the other hand, finite loop unrolling is a simple and efficient technique for finding errors in programs, but it is usually not sound because loop bounds cannot be computed statically. To avoid these drawbacks, we present a technique that takes a middle-course between the two approaches above and which we refer to as *abstract loop unrolling*. In our approach we unroll the first and last iteration of each loop, but abstract all other iterations using an inductive invariant. The abstraction ensures that the approach is sound. Keeping the first and last iteration ensures that the approach can still detect most doomed program points in the surrounding code of the abstracted loop, even if the loop invariant used for the abstraction of the loop was trivial (i.e., *true*). Thus, our approach can take advantage of loop invariants that are provided by the user or a preceding static analysis, but does not rely on them for being useful in practice.

We next explain our loop elimination method in detail and then briefly sketch how it can be generalized to handle programs with procedures.

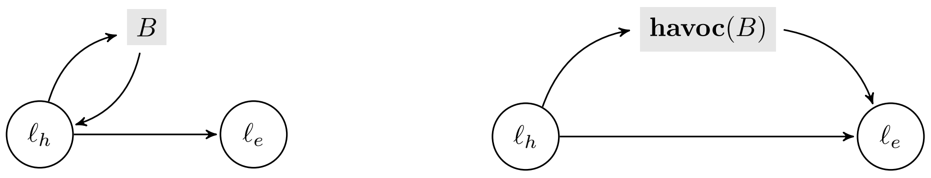


Figure 6.1: Abstraction of a loop (ℓ_h, ℓ_e, B) by adding non-deterministic assignments to the loop targets before and after the loop body. The back edges from B are replaced by edges to the loop exit.

Abstract Loop Unrolling

We now think of our program P as a control flow graph with nodes given by the blocks in P and directed edges corresponding to the **goto** statement in each block. We refer to the strongly connected components of the control flow graph as *loops*. We call maximal strongly connected components *outermost loops* and non-maximal ones *nested loops*. We assume that our programs are structured, i.e., that each loop L in the control flow graph has a unique entry point ℓ_h, which is also the unique exit point of L[1]. We call ℓ_h the *loop header* and $B \overset{\text{def}}{=} L \setminus \{\ell_h\}$ the *loop body* of the loop L. Edges from nodes inside B back to the loop header are called *back edges*. We assume without loss of generality that each loop header is a block that consists of just one **goto** statement that either goes to the first block of the loop body or to a unique block ℓ_e outside the loop. We call ℓ_e the *loop exit*. The variables that are modified by a statement in the blocks of the loop body are called *loop targets* and denoted by $\mathsf{trg}(B)$. In the rest of this section we identify each loop L with the tuple (ℓ_h, ℓ_e, B).

We can now over-approximate a loop (ℓ_h, ℓ_e, B) as shown in Figure 6.1. We denote by **havoc**(B) the program fragment that is obtained from the loop body B by replacing the statement $st(\ell)$ of every block ℓ in B with an incoming edge from ℓ_h with the statement **havoc** $\mathsf{trg}(B)$; $st(\ell)$, and likewise replacing every statement $st(\ell)$ of every block ℓ in B with an outgoing edge to ℓ_h by the statement $st(\ell)$; **havoc** $\mathsf{trg}(B)$. We can think of this transformation as eliminating loops using trivial loop invariants. In fact, if the user or some preceding analysis provides loop invariants, they can be incorporated into the transformation to increase precision (see [4]). All we need to do is assume the invariant after the **havoc** statements for the loop targets.

If we use the loop abstraction technique above without providing a precise

[1]If the program is not structured, one can first apply node splitting to obtain a control flow graph of the assumed form, see e.g., [32].

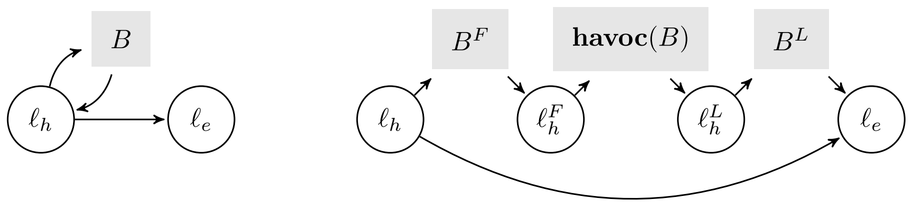


Figure 6.2: Abstract unrolling of a loop (ℓ_h, ℓ_e, B) by introducing copies of the loop body to model the first and the last iteration of the loop

loop invariant, we lose in general too much information for proving that a program point in the loop (or succeeding the loop) is doomed. In particular, we lose any information about the termination behavior of the loop. We therefore combine loop abstraction with finite loop unrolling. We refer to this technique as *abstract loop unrolling*. In our experience it is sufficient to take a look at the first and the last iteration of a loop to detect most doomed program points. Therefore, we model these iterations explicitly and use **havoc** statements for loop targets to model all intermediate iterations. As before, user provided invariants can still be incorporated into this transformation to increase precision. However, in our benchmarks discussed in Section 9 we applied abstract loop unrolling without providing loop invariants and we were still able to detect many interesting errors. Also, compared to simple loop abstraction, abstract unrolling increases the detection rate of doomed program points. In particular, it enables detection of certain cases of non-termination.

The idea of abstract loop unrolling is illustrated in Figure 6.2. Given a loop (ℓ_h, ℓ_e, B), we first compute the sound loop abstraction described above to model an arbitrary number of intermediate loop iterations. Then we introduce two additional copies of the loop head and loop body: ℓ_h^F and B^F, respectively, ℓ_h^L and B^L. The blocks ℓ_h^F and B^F model the first iteration of the loop, while ℓ_h^L and B^L model the last iteration. We assume that for every program point ℓ, the program points ℓ^L and ℓ^F are fresh program points that do not already appear in the program. The blocks in B^F are then obtained from B by replacing every program point ℓ of a block in B by ℓ^F and changing all **goto** statements accordingly. The blocks B^L and the block of ℓ_h^L are obtained from B, respectively, ℓ_h in a corresponding manner. Finally, incoming and outgoing edges to the loop heads $\ell_h, \ell_h^F, \ell_h^L$ in the three copies of the loop body are changed as illustrated in Figure 6.2. Note that we apply unrolling only to outermost loops in the control flow graph. Nested loops are eliminated without unrolling using

```
void altBit(int i)
{
  int a[10];

  while (i<=10)
  {
    if (i%2==0) {
        a[i] = 1;
    } else {
        a[i] = 0;
    }
    i++;
  }
}
```

$$
\begin{array}{rl}
\ell_{\mathsf{init}} : & \textbf{goto } \ell_1; \\
\ell_1 : & \textbf{goto } \ell_2, \ell_6; \\
\ell_2 : & \textbf{assume } i \leq 10; \\
 & \textbf{goto } \ell_3, \ell_4; \\
\ell_3 : & \textbf{assume } i\%2 = 0; \\
 & \textbf{assert } 0 \leq i < 10; \\
 & a[i] := 1; \\
 & \textbf{goto}\, \ell_5; \\
\ell_4 : & \textbf{assume } i\%2 \neq 0; \\
 & \textbf{assert } 0 \leq i < 10; \\
 & a[i] := 0; \\
 & \textbf{goto } \ell_5; \\
\ell_5 : & i := i + 1; \\
 & \textbf{goto } \ell_1; \\
\ell_6 : & \textbf{assume } \neg(i \leq 10); \\
 & \textbf{goto } \ell_{\mathsf{term}};
\end{array}
$$

Figure 6.3: Program ALTBIT with a doomed program point

the simpler technique shown in Figure 6.1. This ensures that the size of the resulting program is linear in the size of the input program.

We demonstrate abstract loop unrolling on the program ALTBIT given in Figure 6.3. The figure shows the program in C syntax and a corresponding Boogie program. The Boogie program contains assertions that guarantee all array accesses to be inside the array boundaries. The program that is obtained from our abstract loop unrolling is given in Figure 6.4. In the program ALTBIT there is an error in the last iteration of the loop. In this iteration, the value of `i` will be 10 and, thus, the array access `a[i]` is outside the bounds of array `a`.

During the last iteration of the loop, in block ℓ_2^L, we assume $i \leq 10$; next we assert $i < 10$ in ℓ_3^L and ℓ_4^L; we increment i by one in ℓ_5^L; and we assume $i > 10$ in ℓ_6. This is a contradiction and thus, there is no terminating execution passing the last iteration. We can conclude that the loop will access the array with an index out of bounds when leaving the loop, i.e., program point ℓ_5 is doomed.

Note that unrolling the first and last iteration of the loop does not help to find more doomed program points inside a loop body but helps to discover more doomed program points outside the loop by providing more information about the loop's behavior to the analysis of the surrounding program points. For instance, the doomed program point ℓ_5 in the program shown in Figure 6.3 could not be detected using loop abstraction without unrolling.

```
$\ell_{\mathsf{init}}$ : goto $\ell_1$;
$\ell_1$ : goto $\ell_2^F, \ell_6$;

$\ell_2^F$ : assume $i \leq 10$;
      goto $\ell_3^F, \ell_4^F$;
$\ell_3^F$ : assume $i\%2 = 0$;
      assert $0 \leq i < 10$;
      $a[i] := 1$;
      goto $\ell_5^F$;
$\ell_4^F$ : assume $i\%2 \neq 0$;
      assert $0 \leq i < 10$;
      $a[i] := 0$;
      goto $\ell_5^F$;
$\ell_5^F$ : $i := i + 1$;
      goto $\ell_2$;

$\ell_2$ : havoc $a, i$;
     assume $i \leq 10$;
     goto $\ell_3, \ell_4$;
$\ell_3$ : assume $i\%2 = 0$;
     assert $0 \leq i < 10$;
     $a[i] := 1$;
     goto $\ell_5$;
$\ell_4$ : assume $i\%2 \neq 0$;
     assert $0 \leq i < 10$;
     $a[i] := 0$;
     goto $\ell_5$;
$\ell_5$ : $i := i + 1$;
     havoc $a, i$;
     goto $\ell_2^L$;

$\ell_2^L$ : assume $i \leq 10$;
      goto $\ell_3^L, \ell_4^L$;
$\ell_3^L$ : assume $i\%2 = 0$;
      assert $0 \leq i < 10$;
      $a[i] := 1$;
      goto $\ell_5^L$;
$\ell_4^L$ : assume $i\%2 \neq 0$;
      assert $0 \leq i < 10$;
      $a[i] := 0$;
      goto $\ell_5^L$;
$\ell_5^L$ : $i := i + 1$;
      goto $\ell_6$;

$\ell_6$ : assume $\neg(i \leq 10)$;
     goto $\ell_{\mathsf{term}}$;
```

Figure 6.4: Abstract loop unrolling applied to program AltBit

By adding copies of the loop body for the last iteration, we potentially introduce dead code into the program that might introduce additional doomed program points that are not present in the original program. This can be seen in the program shown in Figure 6.4: the assume statements at program points ℓ_6 and ℓ_2^L enforce that in any execution going through the blocks modeling the final iteration of the loop we have $i = 10$. Hence, there is no admissible execution going through program point ℓ_4^L, since i is even. Thus ℓ_4^L is doomed in the transformed program. However, recall that in our algorithm we only check whether the program points that are already present in the original program are doomed, not those that have been added in the transformation. We will now show that the transformation is indeed sound.

Let $\mathsf{AbsUnroll}(P)$ be the program that results from applying abstract loop unrolling to a program P. The transformation preserves the existence of admissible executions and is thus sound for the detection of doomed program points. In particular, if an admissible execution π iterates only once through an outermost loop $L = (\ell_h, \ell_e, B)$ in the original program P then we can obtain an admissible execution π' for $\mathsf{AbsUnroll}(P)$ by simple duplication of the single iteration of L in π. The havoc statements in **havoc**(B) ensure that the duplicated iterations connect to a proper execution of $\mathsf{AbsUnroll}(P)$. For the explicit construction of the execution π' see the proof of the following proposition.

Proposition 1. *Let P be a program and ℓ a program point in P. If ℓ is doomed in $\mathsf{AbsUnroll}(P)$ then ℓ is doomed in P.*

Proof. We prove the contraposition: let π be an admissible execution of P passing through ℓ in P. We have to show that there exists an admissible execution π' of $P' = \mathsf{AbsUnroll}(P)$ that passes through ℓ.

In the following, for a sequence of states π and $i, j \in \mathbb{N}$, we denote by $\pi[i, j]$ the sequence $\pi[i] \dots \pi[j]$ if $i \leq j$ and ϵ otherwise. Furthermore, we define π^F to be the sequence obtained from π by mapping for all $i \in \mathbb{N}$, the state $s_i = \pi[i]$ to the state $\pi^F[i] = s_i[\mathsf{pc} := (s_i(\mathsf{pc}))^F]$. We define π^L correspondingly. Finally, we say that an execution π iterates $k > 0$ times through a loop (ℓ_h, ℓ_e, B) of P if there exist $i_0, \dots, i_k \in \mathbb{N}$ such that $0 \leq i_0 < \dots < i_k < \mathsf{len}(\pi)$ and the following conditions hold: (1) for all j with $0 \leq j < k$, $\pi[i_j](\mathsf{pc}) = \ell_h$ and for all i with $i_j < i < i_{j+1}$, $\pi[i](\mathsf{pc}) \neq \ell_h$, (2) $\pi[i_k](\mathsf{pc}) = \ell_h, \pi[i_k + 1](\mathsf{pc}) = \ell_e$ and for all i with $i_0 < i < i_k$, $\pi[i](\mathsf{pc}) \neq \ell_e$, and (3) for all $i < i_0$, if $\pi[i](\mathsf{pc}) = \ell_h$ then there exists j with $i < j < i_0$ and $\pi[j](\mathsf{pc}) = \ell_e$.

We can now construct π' from π by removing or duplicating iterations of loops in π such that π' iterates through all outermost loops exactly three times and through all nested loops exactly one time, while preserving reachability of program point ℓ. Formally, we construct π' from π recursively as follows: define $\pi_0 = \pi$. Then given π_n for some $n \geq 0$, if there is no outermost loop of P through which π_n iterates $k \neq 3$ times and no nested loop through which π_n iterates more than 1 time, define $\pi' = \pi_n$. Otherwise, choose the first loop $L = (\ell_h, \ell_e, B)$ of P through which π_n iterates k times, where L is either an outermost loop and $k \neq 3$ or L is a nested loop and $k > 1$. Let $i_0, \dots, i_k$ be the corresponding indices of the iterations in π_n. Further, let j be the index of the first of these iteration that goes through program point ℓ, if such an iteration exists, and $k - 1$ otherwise, i.e.

$$j \overset{\text{def}}{=} \min(\{k - 1\} \cup \{j \mid 0 \leq j < k \land \exists i.\, i_j \leq i < i_{j+1} \land \pi_n[i](\mathsf{pc}) = \ell\}).$$

Now, if L is an outermost loop we obtain π_{n+1} from π_n by keeping only the first iteration, the j-th iteration, and the last iteration of L:

$$\pi_{n+1} \stackrel{\text{def}}{=} \pi_n[0,i_0](\pi_n[i_0{+}1,i_1])^F\pi_n[i_j{+}1,i_{j+1}{-}1](\pi_n[i_{k-1},i_k{-}1])^L\pi_n[i_k{+}1,\mathsf{len}(\pi_n)]$$

Note that if $k < 3$ then $i_o = i_j$ or $i_j = i_{k-1}$ (or both), i.e., π_{n+1} is obtained from π_n by introducing additional loop iterations via duplication.

If on the other hand L is a nested loop then π_{n+1} is obtained from π_n by keeping only the j-th iteration of L:

$$\pi_{n+1} \stackrel{\text{def}}{=} \pi_n[0,i_0]\pi_n[i_j+1,i_{j+1}-1]\pi_n[i_k+1,\mathsf{len}(\pi_n)]$$

Clearly, π' is well-defined, since π is finite.

The non-deterministic assignments of the loop targets in every abstracted loop body in P' ensure that every sequence of states that is an execution of the corresponding original loop body in P is also an execution of the abstracted loop body in P'. Using this fact we can prove that π' is an execution of P'. Finally, by construction we have for all $n \geq 0$ that $\mathsf{final}(\pi_n)(\mathsf{pc}) = \ell_{\mathsf{term}}$ and $\pi_n[i](\mathsf{pc}) = \ell$ for some i, $0 \leq i \leq \mathsf{len}(\pi_n)$. Hence, π' is admissible and passes through ℓ. ∎

6.2 Eliminating Procedure Calls

While our simple programming language from Section 5 does not support procedures, we do support them in our implementation. In the folowing, we briefly sketch how we handle procedures.

In the Boogie language [3,36] a program with procedures consists of a set of procedure declarations. A procedure declaration specifies the formal parameters of the procedure and the procedure body. The procedure body consists of a set of blocks as in Section 5 where the statements constituting the blocks are extended by procedure calls and return statements. Figure 6.5 shows an example of a program with procedures.

The analysis of a program with procedures proceeds as follows. For each procedure declaration we first eliminate all procedure calls in the blocks of the procedure body. This effectively leaves us with a simple program for every procedure declaration. We then analyze each of these simple programs in isolation using the algorithm given in Figure 5.2.

```
procedure $pow$ ($x$ : int, $m$ : int)          procedure $main$ () {
  returns $y$ {                                   $\ell_0$ : call $z := pow\ (2, 3)$;
  $\ell_0$ : goto $\ell_2, \ell_3$;                       return;
  $\ell_1$ : assume $m > 0$;                      }
      call $y := pow\ (x, m - 1)$;
      $y := y \cdot x$;
      goto $\ell_3$;
  $\ell_2$ : assume $m \leq 0$;
      $y := 1$;
      goto $\ell_3$;
  $\ell_3$ : return;
}
```

Figure 6.5: Program with procedures

Abstract Inlining. Similar to abstract loop unrolling, we employ a combination of inlining and contract-based abstraction to eliminate procedure calls from the blocks of each procedure declaration.

We inline procedure calls in a straightforward manner. We replace the call statement by a copy of the procedure body and a set of statements that assign the parameters of the call statement to the formal parameters of the procedure. Procedure calls in the inlined procedure or recursive calls are abstracted using trivial procedure contracts (i.e., with pre and postcondition *true*).

For contract-based abstraction, the global variables modified by the called procedure and the variables receiving the return values are assigned non-deterministic values by **havoc** statements. As for abstract loop unrolling, the precision of the abstraction can be improved if either the user or a preceding analysis provides actual contracts for the abstracted procedure.

Using abstract inlining, we can, e.g., detect if a procedure is called with illegal arguments that lead to abnormal program termination, i.e., no admissible execution is passing the procedure call. Figure 6.6 shows the result of applying abstract inlining to the procedures in Figure 6.5.

The soundness argument for abstract inlining is similar to the soundness argument for abstract loop unrolling. Given an admissible execution π of the original program P that executes a procedure inlined in the transformed program, then π is also an execution of the transformed program provided the inlined procedure has no further procedure calls that are executed by π. If the inlined procedure has itself calls to other procedures then we can obtain an ad-

```
**procedure** $pow$ ($x$ : int, $m$ : int)
  **returns** $y$ {
   $\ell_0$ : **goto** $\ell_2, \ell_3$;
   $\ell_1$ : **assume** $m > 0$;
        $x_r := x$;
        $m_r := m - 1$;
        **goto** $\ell_0^R$;

   $\ell_0^R$ : **goto** $\ell_2^R, \ell_3^R$;
   $\ell_1^R$ : **assume** $m_r > 0$;
        **havoc** $y_r$;
        $y_r := y_r \cdot x_r$;
        **goto** $\ell_3^R$;
   $\ell_2^R$ : **assume** $m_r \leq 0$;
        $y_r := 1$;
        **goto** $\ell_3^R$;
   $\ell_3^R$ : **goto** $\ell_1^C$;

   $\ell_1^C$ : $y := y_r$;
        $y := y * x$;
        **goto** $\ell_3$;
   $\ell_2$ : **assume** $m \leq 0$;
        $y := 1$;
        **goto** $\ell_3$;
   $\ell_3$ : **return**;
}
```

```
**procedure** $main$ () {
   $\ell_0$ : $x_r := 2$;
        $m_r := 3$;
        **goto** $\ell_0^R$;

   $\ell_0^R$ : **goto** $\ell_2^R, \ell_3^R$;
   $\ell_1^R$ : **assume** $m_r > 0$;
        **havoc** $y_r$;
        $y_r := y_r \cdot x_r$;
        **goto** $\ell_3^R$;
   $\ell_2^R$ : **assume** $m_r \leq 0$;
        $y_r := 1$;
        **goto** $\ell_3^R$;
   $\ell_3^R$ : **goto** $\ell_0^C$;

   $\ell_0^C$ : $z := y_r$;
        **return**;
}
```

Figure 6.6: Abstract inlining applied to the program in Figure 6.5

missible execution from π simply by cutting out the parts of π that correspond to the execution of these nested calls. The havoc statements in the inlined procedures ensure that the resulting sequence of states is indeed an admissible execution of the transformed program. As for abstract loop unrolling, the abstract inlining transformation is in general not complete for detecting doomed program points. Some inadmissible executions, such as executions that execute a nonterminating procedure, are not preserved by the transformation.

We point out that (abstract) inlining can only be used to check if the call statement is doomed. It is not used to detect if the called method contains a doomed program point (this is not possible, because we cannot know all calling contexts of a method). Hence, when inlining we can only report a doomed program point if any path through the inlined method is doomed for the given input values.

In practice we can control the trade-off between efficiency and precision of the analysis by using heuristics that decide when to inline a procedure and when to abstract it. For instance, depending on the size of a called procedure one may decide to recursively inline calls in the body of the called procedure or not inline the called procedure at all and immediately abstract the call. In our experiments we experienced that calls to non-trivial methods are unlikely to be proven doomed because, e.g. they have some admissible executions (due to their complexity) or due to e.g., imprecise heap over-approximation or a loop in the called function it is not possible to prove that all executions of the method body are inadmissible in the considered calling context. In any case, using contract-based abstraction always looses precision. However, if the called function itself cannot be analyzed precisely (e.g., because it is recursive or contains a loop) inlining is imprecise as well.

6.3 Introducing Reachability Variables

We next explain in detail how a program P is translated to a program P^* with reachability variables.

We add a set of auxiliary variables and assignments to our program P that allow us to restrict the set of admissible executions of P to those that are passing a certain program point ℓ. The actual transformation is straightforward. For each block ℓ in the program, we introduce a new Boolean reachability variable R_ℓ. We can think of R_ℓ as a static uninitialized variable. Now, we replace the statement of $st(\ell)$ of the block by the statement $R_\ell := \mathit{true}; st(\ell)$ that sets the reachability variable to *true* whenever the block ℓ is executed. Finally we add the following assert statement at the end of the last block of the program:

$$\textbf{assert}(\bigwedge_{\ell \in P} R_\ell)$$

Clearly the transformation from P to P^* preserves all admissible executions.

Proposition 2. *For all programs P and program points ℓ in P, ℓ is doomed in P if and only if ℓ is doomed in P^*.*

Proof. Let X be the set of all program variables of program P. Now let π be an admissible execution of P passing through ℓ. Then map the states of π to

```
$\ell_{\mathsf{init}}$ : $R_{\ell_{\mathsf{init}}} := \mathit{true}$;
        goto $\ell_1$;
$\ell_1$ : goto $\ell_2^F, \ell_6$;

$\ell_2^F$ : $R_{\ell_2^F} := \mathit{true}$;
        assume $i \leq 10$;
        goto $\ell_3^F, \ell_4^F$;
$\ell_3^F$ : $R_{\ell_3^F} := \mathit{true}$;
        assume $i \% 2 = 0$;
        assert $0 \leq i < 10$;
        $a[i] := 1$;
        goto $\ell_5^F$;
$\ell_4^F$ : $R_{\ell_4^F} := \mathit{true}$;
        assume $i \% 2 \neq 0$;
        assert $0 \leq i < 10$;
        $a[i] := 0$;
        goto $\ell_5^F$;
$\ell_5^F$ : $R_{\ell_5^F} := \mathit{true}$;
        $i := i + 1$;
        goto $\ell_2$;

$\ell_2$ : $R_{\ell_2} := \mathit{true}$;
        havoc $a, i$;
        assume $i \leq 10$;
        goto $\ell_3, \ell_4$;
$\ell_3$ : $R_{\ell_3} := \mathit{true}$;
        assume $i \% 2 = 0$;
        assert $0 \leq i < 10$;
        $a[i] := 1$;
        goto $\ell_5$;
$\ell_4$ : $R_{\ell_4} := \mathit{true}$;
        assume $i \% 2 \neq 0$;
        assert $0 \leq i < 10$;
        $a[i] := 0$;
        goto $\ell_5$;
$\ell_5$ : $R_{\ell_5} := \mathit{true}$;
        $i := i + 1$;
        havoc $a, i$;
        goto $\ell_2^L$;

$\ell_2^L$ : $R_{\ell_2^L} := \mathit{true}$;
        assume $i \leq 10$;
        goto $\ell_3^L, \ell_4^L$;
$\ell_3^L$ : $R_{\ell_3^L} := \mathit{true}$;
        assume $i \% 2 = 0$;
        assert $0 \leq i < 10$;
        $a[i] := 1$;
        goto $\ell_5^L$;
$\ell_4^L$ : $R_{\ell_4^L} := \mathit{true}$;
        assume $i \% 2 \neq 0$;
        assert $0 \leq i < 10$;
        $a[i] := 0$;
        goto $\ell_5^L$;
$\ell_5^L$ : $R_{\ell_5^L} := \mathit{true}$;
        $i := i + 1$;
        goto $\ell_6$;

$\ell_6$ : $R_{\ell_6} := \mathit{true}$;
        assume $\neg(i \leq 10)$;
        assert$(\bigwedge_{\ell \in P} R_\ell)$
        goto $\ell_{\mathsf{term}}$;
```

Figure 6.7: Loop-free program AltBit with reachability variables

a sequence of states π' of P^* as follows: for all i with $0 \leq i < \mathsf{len}(\pi)$, define $\pi'[i](x) = \pi[i](x)$ for all $x \in X$ and $\pi'[i](R_\ell) = \mathit{true}$ for all $\ell \in P$. Then π' is an admissible execution of P^*, since P^* behaves as P and only assigns value *true* to variables in $\mathcal{R}$. Thus, the additional assertion $\bigwedge_{\ell \in P} R_\ell$ in P^* is not violated by π'. Furthermore, π' passes through ℓ, since π does. For proving the other direction, let π' be an admissible execution of P^* passing through ℓ. Then we immediately obtain an admissible execution π of P that passes through ℓ by projecting all states in π' onto the variables X. ∎

Figure 6.7 shows the loop-free program from our running example augmented with reachability variables. For each block, one boolean reachability variable is introduced and assigned to *true* at the beginning of this block. The block labeled with ℓ_6 further is extended by an assertion that all reachability variables are *true*. This allows us to make all executions diverge that do not pass a certain program point ℓ_i, simply by initializing the corresponding reachability variable R_{ℓ_i} to *false*.

6.4 Program Passification

After introducing the reachability variables, we transform the program into passive form. This is done by applying a *single assignment* transformation [12] where auxiliary variables are introduced to ensure that each program variable is assigned at most once per execution path [20]. The general idea is to replace each read of a variable by the auxiliary variable that represents its value at that point in the program, and to introduce a new auxiliary variable for every write. For example, an assignment $x := x + 1$ may be transformed into $x_{k+1} := x_k + 1$, where k is some sequence number (see [4, 20, 35] for details). Second, since no assignment of an auxiliary variable is preceded by a use of that variable, we can replace each assignment $x_k := e$ by an assume statement $\textbf{assume}(x_k = e)$.

Let $\mathsf{Passify}(P)$ be the result of applying the single assignment transformation to a program P. The following proposition states soundness of the transformation. Its proof follows a similar argument then stated in [20, Theorem 1].

Proposition 3. *For all loop-free programs P and program point ℓ in P, ℓ is doomed in P if and only if ℓ is doomed in $\mathsf{Passify}(P)$.*

The function $\mathsf{Transform}$ is now defined as the composition of the transformations described in the previous sections: $\mathsf{Transform}(P) \overset{\text{def}}{=} \mathsf{Passify}(\mathsf{AbsUnroll}(P)^*)$. Soundness of $\mathsf{Transform}$ follows from Propositions 1, 2, and 3.

Proposition 4. *For any program P and program point ℓ in P, if ℓ is doomed in $\mathsf{Transform}(P)$ then ℓ is doomed in P.*

The passification is the final step of our transformation. Starting with the initial program in Figure 6.3 we first generated a corresponding loop free program in Figure 6.4 and added reachability variables in Figure 6.7. In this program, there is no looping control-flow left and thus, by adding auxiliary variables, we can transform the program in a way that each variable is only assigned once on each control path. I.e., we can create a program without state changes where assignments can be replaced by assumptions.

```
$\ell_0$ : $R_{\ell_0} = true$;
     goto $\ell_1$;
$\ell_1$ : goto $\ell_2^F, \ell_{tmp}$;
```

```
$\ell_2^F$ : assume $R_{\ell_2^F} = true$;
      assume $i_0 \leq 10$;
      goto $\ell_3^F, \ell_4^F$;
$\ell_3^F$ : assume $R_{\ell_3^F} = true$;
      assume $R_{\ell_4^F} = R^{init}_{\ell_4^F}$;
      assume $i_0 \% 2 = 0$;
      assert $0 \leq i_0 < 10$;
      assume $a_0[i_0] = 1$;
      goto $\ell_5^F$;
$\ell_4^F$ : assume $R_{\ell_4^F} = true$;
      assume $R_{\ell_3^F} = R^{init}_{\ell_3^F}$;
      assume $i_0 \% 2 \neq 0$;
      assert $0 \leq i_0 < 10$;
      assume $a_0[i_0] = 0$;
      goto $\ell_5^F$;
$\ell_5^F$ : assume $R_{\ell_5^F} = true$;
      assume $i_1 = i_0 + 1$;
      goto $\ell_2$;
```

```
$\ell_2$ : assume $R_{\ell_2} = true$;
     assume $i_2 \leq 10$;
     goto $\ell_3, \ell_4$;
$\ell_3$ : assume $R_{\ell_3} = true$;
     assume $R_{\ell_4} = R^{init}_{\ell_4}$;
     assume $i_2 \% 2 = 0$;
     assert $0 \leq i_2 < 10$;
     assume $a_1[i_2] = 1$;
     goto $\ell_5$;
$\ell_4$ : assume $R_{\ell_4} = true$;
     assume $R_{\ell_3} = R^{init}_{\ell_3}$;
     assume $i_2 \% 2 \neq 0$;
     assert $0 \leq i_2 < 10$;
     assume $a_1[i_2] = 0$;
     goto $\ell_5$;
$\ell_5$ : assume $R_{\ell_5} = true$;
     assume $i_3 = i_2 + 1$;
     goto $\ell_2^L$;
```

```
$\ell_2^L$ : $R_{\ell_2^L} = true$;
      assume $i_4 \leq 10$;
      goto $\ell_3^L, \ell_4^L$;
$\ell_3^L$ : $R_{\ell_3^L} = true$;
      assume $R_{\ell_4^L} = R^{init}_{\ell_4^L}$;
      assume $i_4 \% 2 = 0$;
      assert $0 \leq i_4 < 10$;
      $a_3[i_4] := 1$;
      goto $\ell_5^L$;
$\ell_4^L$ : $R_{\ell_4^L} = true$;
      assume $R_{\ell_3^L} = R^{init}_{\ell_3^L}$;
      assume $i_4 \% 2 \neq 0$;
      assert $0 \leq i_4 < 10$;
      $a_3[i_4] := 0$;
      goto $\ell_5^L$;
$\ell_5^L$ : $R_{\ell_5^L} = true$;
      $i_5 := i_4 + 1$;
      goto $\ell_6$;
```

```
$\ell_6$ : assume $R_{\ell_6} = true$;
     assume $\neg(i_5 \leq 10)$;
     assert$(\bigwedge_{\ell \in P} R_\ell)$
     goto $\ell_{\mathsf{term}}$;
```

```
$\ell_{tmp}$ : assume $R_{\ell_2} = R^{init}_{\ell_2} \wedge R_{\ell_2^F} = R^{init}_{\ell_2^F} \wedge R_{\ell_2^L} = R^{init}_{\ell_2^L}$;
       assume $R_{\ell_3} = R^{init}_{\ell_3} \wedge R_{\ell_3^F} = R^{init}_{\ell_3^F} \wedge R_{\ell_3^L} = R^{init}_{\ell_3^L}$;
       assume $R_{\ell_4} = R^{init}_{\ell_4} \wedge R_{\ell_4^F} = R^{init}_{\ell_4^F} \wedge R_{\ell_4^L} = R^{init}_{\ell_4^L}$;
       assume $R_{\ell_5} = R^{init}_{\ell_5} \wedge R_{\ell_5^F} = R^{init}_{\ell_5^F} \wedge R_{\ell_5^L} = R^{init}_{\ell_5^L}$;
       assume $i_5 = i_0$;
       assume $a_4 = a_0$;
       goto $\ell_6$;
```

Figure 6.8: The program AltBit after transformation into passive form

Figure 6.8 shows the passive version of our initial program AltBit. Notice that a new program point ℓ_{tmp} has been added. This program point is needed to express that the variables modified by the loop-body remain unchanged if the loop is not entered. This, in particular, affects the reachability variables of the loop body. It explicitly states that, e.g. the reachability variable R_{ℓ_5} associated with the program point ℓ_5 keeps its initial value $R_{\ell_5}^{init}$ (which intentionally is not specified) instead of being set to *true*. If we would initialize $R_{\ell_5}^{init}$ to *false*, any execution that does not pass the program point ℓ_5 - and therefore assigns R_{ℓ_5} to *true* - will violate the assertion at ℓ_6. The combination of reachability variables and single assignment is a simple and efficient way the make all executions inadmissible that do not pass a certain program point.

Chapter 7

Detecting Doomed Program Points

In the previous chapter we described how to transform a program P into a loop-free passive program $\mathsf{Transform}(P)$ with reachability variables. We now explain how to detect doomed program points in the transformed program and, by soundness of the transformation, in the original program P.

7.1 Computing Weakest Liberal Preconditions

Recall condition (4.1) from Chapter 4, which characterizes doomed program points in terms of weakest liberal preconditions. We use this observation to reduce the problem of detecting doomed program points in the transformed program to the problem of checking validity of logical formulas that express weakest liberal preconditions.

Let P be a program in loop-free passive form and let F be a formula over the program variables in P. Using Dijkstra's predicate transformer semantics of programs [14], we can compute a formula $\mathsf{wlp}(P, F)$ that denotes the weakest liberal precondition of program P and formula F. First, for each block $\ell : S$ in

P we recursively define the formula $\mathsf{wlp}(S, F)$, as follows:

$$\begin{aligned}
\mathsf{wlp}(\mathbf{goto}\, \ell_1, \dots, \ell_n, F) &= \mathsf{wlp}(st(\ell_1), F) \wedge \dots \wedge \mathsf{wlp}(st(\ell_n), F) \\
\mathsf{wlp}(\mathbf{assume}\, E, F) &= E \implies F \\
\mathsf{wlp}(\mathbf{assert}\, E, F) &= E \implies F \\
\mathsf{wlp}(S_1; S_2, F) &= \mathsf{wlp}(S_1, \mathsf{wlp}(S_2, F))
\end{aligned}$$

Note that $\mathsf{wlp}(S, F)$ is well-defined since the program P is in loop-free passive form. The weakest liberal precondition $\mathsf{wlp}(P, F)$ of program P and formula F is then simply given by the formula $\mathsf{wlp}(st(\ell_{\mathsf{init}}), F)$ where ℓ_{init} is the program point of the start block of program P.

Given the semantics of statements defined in Chapter 5, we can easily prove that $\mathsf{wlp}(P, F)$ has the intended meaning.

Lemma 1. *Let P be a program in loop-free passive form, F a formula over program variables of P, and s an initial state of P. Then $s \models \mathsf{wlp}(P, F)$ iff for all admissible executions π of P starting in s, $\mathsf{final}(\pi) \models F$.*

With Lemma 1 we can now prove that weakest liberal preconditions can be used to detect doomed program points in transformed programs.

Proposition 1. *Let P be a program and ℓ a program point in P. Then ℓ is doomed in $\mathsf{Transform}(P)$ iff the formula $R_\ell \vee \mathsf{wlp}(\mathsf{Transform}(P), \mathit{false})$ is valid.*

Proof. For a program point ℓ in P, let R'_ℓ be the variable used in the passification step of $\mathsf{Transform}$ to represent the value of the reachability variable R_ℓ after the update $R_\ell := \mathit{true}$ in block ℓ, i.e., each block ℓ in $\mathsf{Transform}(P)$ contains the statement $\mathbf{assume}(R_\ell)$ and the assert statement at the end of $\mathsf{Transform}(P)$ is of the form $\mathbf{assert}(\bigwedge_{\ell \in P} R'_\ell)$. Furthermore, on all paths in the CFG of $\mathsf{Transform}(P)$ that do not pass through block ℓ there is some block containing the assume statement $\mathbf{assume}(R'_\ell \equiv R_\ell)$.

For proving the right-to-left direction, assume that ℓ is not doomed in $\mathsf{Transform}(P)$. Then there exists an admissible execution π of $\mathsf{Transform}(P)$ passing through ℓ. Define the sequence of states π' as follows: for all i, $1 \leq i \leq \mathsf{len}(\pi)$, and program variables x of $\mathsf{Transform}(P)$, if $x \neq R_\ell$ then $\pi'[i](x) = \pi[i](x)$, and $\pi'[i](R_\ell) = \mathit{false}$. Since π passes through ℓ, the variable R_ℓ does not appear in any of the blocks $st(\pi[i](\mathsf{pc}))$, for all $1 \leq i \leq \mathsf{len}(\pi)$. Thus, π' is still an admissible execution of $\mathsf{Transform}(P)$. By definition of π' we have

$\pi'[0] \not\models R_\ell$. Furthermore, since $\mathsf{final}(\pi') \not\models \mathit{false}$, it follows from Lemma 1 that $\pi'[0] \not\models \mathsf{wlp}(\mathsf{Transform}(P), \mathit{false})$.

For proving the left-to-right direction, assume that s is a state of $\mathsf{Transform}(P)$ with $s \not\models R_\ell$ and $s \not\models \mathsf{wlp}(\mathsf{Transform}(P), \mathit{false})$. Define s_0 such that s_0 agrees with s on the values of all program variables, except that $s_0(\mathsf{pc}) = \ell_{\mathsf{init}}$. Thus, s_0 is an initial state of $\mathsf{Transform}(P)$, and we still have $s_0 \not\models R_\ell$ and $s_0 \not\models \mathsf{wlp}(\mathsf{Transform}(P), \mathit{false})$. Then from Lemma 1 follows that there is an admissible execution π of $\mathsf{Transform}(P)$ starting in s_0. If π was not passing through ℓ then there would be some i with $1 \leq i \leq \mathsf{len}(\pi)$ such that the block $\pi[i](\mathsf{pc})$ contained the assume statement $\mathbf{assume}(R'_\ell \equiv R_\ell)$. Since $s_0 \not\models R_\ell$ and π starts in s_0 we would thus have for all i, $1 \leq i \leq \mathsf{len}(\pi)$, $\pi[i] \not\models R'_\ell$. In particular, $\mathsf{final}(\pi)$ would violate the assert statement $\mathbf{assert}(\bigwedge_{\ell \in P} R'_\ell)$ and, hence, π would not be admissible. It follows that π must pass through ℓ, i.e., ℓ is not doomed in $\mathsf{Transform}(P)$.

From Propositions 1 and 4 now follows the soundness of algorithm Exorcise.

Theorem 1. *The algorithm* Exorcise *is sound, i.e., for all programs P and program points ℓ of P, if $\ell \in \mathsf{Exorcise}(P)$ then ℓ is doomed in P.*

7.2 Block Variables and Incremental Checking

The definition of the formula $\mathsf{wlp}(P, F)$ that we chose in the previous section is a rather naive one. While this naive definition simplifies reasoning about the correctness of our algorithm, it is impractical due to redundancies in the formula representation.

Note that each block ℓ in program P may have multiple predecessor blocks (i.e., blocks with a **goto** statement to block ℓ), due to join points in the control-flow graph. The weakest precondition of each such predecessor block contains the formula $\mathsf{wlp}(st(\ell), F)$ as a subformula, which leads to duplication of subformulas in the final formula $\mathsf{wlp}(P, F)$. In fact, the number of duplications of a subformula $\mathsf{wlp}(st(\ell), F)$ in the formula $\mathsf{wlp}(P, F)$ is equal to the number of control-flow paths in program P that visit block ℓ. The number of such paths can be exponential in the number of blocks in the program, leading to an exponential explosion of the size of the formulas that is given to the SMT solver.

We can avoid duplication of the subformulas generated for each block by using the idea of *block variables* [35]. For this purpose we introduce an auxiliary

Boolean variable B_ℓ for each program point ℓ that replaces all occurrences of subformulas $\mathsf{wlp}(st(\ell), \mathit{false})$. This means, instead of the formula $\mathsf{wlp}(P, \mathit{false})$, we build the formula

$$\begin{aligned} F_{Bdef} : & \bigwedge_{\ell \in P} \left(B_\ell \equiv \mathsf{wlp}(st'(\ell), B_{\ell_1} \wedge \cdots \wedge B_{\ell_n} \right) \\ & \wedge \neg B_{Term} \end{aligned}$$

Here $\ell_1, \ldots, \ell_n$ are the program points occurring in the final **goto** statement of block ℓ. The statement $st'(\ell)$ refers to the statement of block ℓ where the final **goto** statement is replaced by the statement **assume** *true*.

The SMT solver now checks for each program point ℓ in P, whether the verification condition $F_\ell : (R_\ell \vee F_{Bdef}) \implies B_{\ell_{\mathsf{init}}}$ is valid or equivalently, whether $\neg R_\ell \wedge F_{Bdef} \wedge B_{\ell_{\mathsf{init}}}$ is unsatisfiable. Note that we have to check the validity of F_ℓ for each program point ℓ separately. However, we can reuse the formula F_{Bdef} that occurs in each such check. Most SMT solvers support incremental queries, which means that they can reuse the learned clauses from this formula between the separate checks.

7.3 Example

For our running example from the previous chapter we can generate a weakest liberal precondition representation straight forward from the passive version of the program AltBit given in Figure 6.8. As the program only contains **assume** and **assert** statements, we can apply the predicate transformer semantics from Section 7.1 to obtain the formula representation of each basic block and then use the optimization from Section 7.2 to obtain the formula given in Figure 7.1 that can be sent to the theorem prover. Notice that in this formula, which is unsatisfiable if ℓ is doomed, the only reference to ℓ is the last conjunct. Therefore, the rest of the formula can be precomputed and stored in the theorem prover (e.g., it can be pushed on the axioms stack in $Z3$).

In this example program, the last iteration of the WHILE-loop, where i reaches 10, always violates the bounds of the array a. In the formula from Figure 7.1 we can see the contradiction between ℓ_3^L, ℓ_4^L, ℓ_5^L, and ℓ_6 which can be simplified to $(0 \leq i_4 < 10) \wedge (i_5 = i + 4 + 1) \wedge (\neg(i_5 \leq 10))$. The weakest liberal precondition representations of the basic blocks in the loop body trivially evaluate to *true*.

$$
\begin{aligned}
& B_{\ell_0} \equiv \neg(R_{\ell_{\mathsf{init}}} \wedge \neg B_{\ell_1}) \\
\wedge \quad & B_{\ell_1} \equiv \ell_2^F \wedge B_{\ell_{tmp}} \\
\wedge \quad & B_{\ell_2^F} \equiv \neg(R_{\ell_2^F} \wedge (i_0 \leq 10) \wedge \neg(B_{\ell_3^F} \wedge B_{\ell_4^F})) \\
\wedge \quad & B_{\ell_3^F} \equiv \neg(R_{\ell_3^F} \wedge (i_0\%2 = 0) \wedge (0 \leq i_0 < 10) \wedge (a_0[i_0] = 1) \wedge \neg B_{\ell_5^F}) \\
\wedge \quad & B_{\ell_4^F} \equiv \neg(R_{\ell_4^F} \wedge (i_0\%2 \neq 0) \wedge (0 \leq i_0 < 10) \wedge (a_0[i_0] = 0) \wedge \neg B_{\ell_5^F}) \\
\wedge \quad & B_{\ell_5^F} \equiv \neg(R_{\ell_5^F} \wedge (i_1 = i_0 + 1) \wedge \neg B_{\ell_2}) \\
\\
\wedge \quad & B_{\ell_2} \equiv \neg(R_{\ell_2} \wedge (i_2 \leq 10) \wedge \neg(B_{\ell_3} \wedge B_{\ell_{4)}})) \\
\wedge \quad & B_{\ell_3} \equiv \neg(R_{\ell_3} \wedge (i_2\%2 = 0) \wedge (0 \leq i_2 < 10) \wedge (a_1[i_2] = 1) \wedge \neg B_{\ell_5}) \\
\wedge \quad & B_{\ell_4} \equiv \neg(R_{\ell_4} \wedge (i_2\%2 \neq 0) \wedge (0 \leq i_2 < 10) \wedge (a_1[i_2] = 0) \wedge \neg B_{\ell_5}) \\
\wedge \quad & B_{\ell_5} \equiv \neg(R_{\ell_5} \wedge (i_3 = i_2 + 1) \wedge \neg B_{\ell_2^L}) \\
\\
\wedge \quad & B_{\ell_2^L} \equiv \neg(R_{\ell_2^L} \wedge (i_4 \leq 10) \wedge \neg(B_{\ell_3^L} \wedge B_{\ell_4^L})) \\
\wedge \quad & B_{\ell_3^L} \equiv \neg(R_{\ell_3^L} \wedge (i_4\%2 = 0) \wedge (0 \leq i_4 < 10) \wedge (a_3[i_4] = 1) \wedge \neg B_{\ell_5^L})) \\
\wedge \quad & B_{\ell_4^L} \equiv \neg(R_{\ell_4^L} \wedge (i_4\%2 \neq 0) \wedge (0 \leq i_4 < 10) \wedge (a_3[i_4] = 0) \wedge \neg B_{\ell_5^L}) \\
\wedge \quad & B_{\ell_5^L} \equiv \neg(R_{\ell_5^L} \wedge (i_5 = i_4 + 1) \wedge \neg B_{\ell_6}) \\
\\
\wedge \quad & B_{\ell_{tmp}} \equiv \neg(\textstyle\bigwedge_{2 \leq i \leq 5}(R_{\ell_i^F} = R^{init}_{\ell_i^F} \wedge R_{\ell_i} = R^{init}_{\ell_i} \wedge R_{\ell_i^L} = R^{init}_{\ell_i^L}) \\
& \qquad \wedge (i_5 = i_0) \wedge (a_4 = a_0) \wedge \neg B_{\ell_6}) \\
\\
\wedge \quad & B_{\ell_6} \equiv \neg(R_{\ell_6} \wedge (\neg(i_5 \leq 10)) \wedge \textstyle\bigwedge_{\ell \in P} R_\ell \wedge \neg \mathit{false}) \\
\\
\wedge \quad & (\neg B_{\ell_0}) \\
\wedge \quad & (\neg R_\ell)
\end{aligned}
$$

Figure 7.1: The formula sent to the theorem prover to check if the program point ℓ is doomed in the program AltBit. If the theorem prover proves the formula to be unsatisfiable, we report that ℓ is doomed.

That is, our formula can be reduced to the simple formula:

$$\Big(R_{\ell_0} \wedge \bigwedge_{2\leq i\leq 5}(R_{\ell_i^F} = R_{\ell_i^F}^{init} \wedge R_{\ell_i} = R_{\ell_i}^{init} \wedge R_{\ell_i^L} = R_{\ell_i^L}^{init}) \wedge (i_5 = i_0)$$
$$\wedge(a_4 = a_0) \wedge (R_{\ell_6} \wedge (\neg(i_5 \leq 10)) \wedge \bigwedge_{\ell\in P} R_\ell \wedge \neg \mathit{false})\Big) \wedge (\neg R_\ell^{init})$$

which is unsatisfiable if R_ℓ is chosen to be the reachability variable of the program point ℓ_2,ℓ_3,ℓ_4, or ℓ_5 and therefore these program points are proven to be doomed by our algorithm.

Error Message. The quality of feedback is the most crucial aspect of any static analysis tool. It has to provide information that can help to increase the quality of the program and the productivity of the programmer. Doomed program point analysis takes an extreme position because it only detects a small class of errors but for those it can prove their existence. However, returning a set of program points that cannot be passed by normal terminating executions is not yet a valuable information. The program point might be doomed because preceding commands fail on any execution and thus the program point itself does not indicate which statement actually fails. A program point might also be doomed because it is not passed by any execution (i.e., it is dead code). Maybe a program point is doomed because it is an intentional program abortion (e.g. **assert** *false*). All these cases are doomed program points according to our definition, however, for a user it is important to get additional information on why a program point is doomed. E.g., in our running example the contradiction between $(0 \leq i_4 < 10) \wedge (i_5 = i + 4 + 1) \wedge (\neg(i_5 \leq 10))$ would be a valuable information. Unfortunately, we cannot use the unsatisfiable core of the formula to isolate the contradicting clauses, because, in particular, if we consider the structure of our formula $\neg F_{B_{def}} \wedge \neg R_\ell$, the negated reachability variable R_ℓ of the block that we analyze is of course the smallest unsatisfiable core of the formula. In fact the we are looking for the unsatisfiable core of the trace formula of each trace passing through the doomed program point (according to the definition of doomed program points we can be sure that an unsatisfiable core exists because each trace formula is infeasible). We are only interested in the contradicting statements on paths that contain the program point ℓ. We make use of the fact that each passive statement also is a literal in the formula sent to the theorem prover and thus in some trace formula. We can identify the statements that are part of the unsatisfiable cores of the trace formulas by randomly removing statements from the passive program that correspond to a statement in the original program (i.e., all statements that do not relate to reachability variables) and checking if the program is still doomed. If removing a statement adds admissible executions to the program, we know, that this statement is part of the unsatisfiable core of some trace formula and that it is on a path that contains ℓ (otherwise there cannot be an admissible execution of this path because the assertion related that all reachability variables are one at the end of the path would be violated).

This approach allows us to identify those statements that contradict on the paths that contain the program point ℓ. These statements are a more compact representation of the problem and can help to further interpret the problem. E.g., if all returned statements are **assume** statements, it is certain that the doomed program point refers to unreachable code after the last **assume**.

Such an approach of identifying statements that are reported to the user is very primitive but suitable for our purpose as, if a doomed program point is detected, the computation time to generate an error message is not that relevant.

For a real implementation of doomed program point detection, it is important to further distinguish doomed program points (e.g., dead code, intentional abortion, etc.). In this thesis, however, we only focus on the efficient detection of doomed program points and leave any post processing of the result for the future work.

Chapter 8

Optimizations

We have presented an algorithm that detects whether a given program point is doomed by calling a theorem prover. In practice we are not just interested in checking whether one particular program point is doomed, but we want to compute all (detectable) doomed program points in a program. To compute the set of all doomed program points our algorithm Exorcise simply iterates over all program points and checks for each one whether it is doomed or not. We can do better and exploit dependencies between different program points in the control flow graph of the program that allow us to reduce the total number of theorem prover calls.

The algorithm Exorcise$^+$ shown in Figure 8.1 exploits such dependencies. Like the original procedure from Figure 5.2, Exorcise$^+$ iterates over the set of program points in P, which are stored in a work set N. While N is non-empty it chooses a program point $\ell \in N$ and checks whether it is doomed. If ℓ is doomed then it computes a set of program points $\mathsf{Doomed}(P', \ell, N)$, which are the program points in N that are doomed in P' under the assumption that ℓ is doomed. This set is then removed from the work set and added to the set of doomed program points. Similarly, if ℓ is not doomed then the algorithm computes the set $\mathsf{Absolved}(P', \ell, N)$ of program points in N that are not doomed in P' under the assumption that ℓ is not doomed, and removes them from the work set in one go. In the following we describe how the subroutines Doomed and Absolved are implemented.

```
proc Exorcise$^+$($P$ : program)
    var $P'$ : program
    var $\varphi$ : formula
    var $D$ : set of doomed program points
    var $N$ : set of program points
    var $A$ : set of program points
    begin
        $P' :=$ Transform($P$)
        $\varphi :=$ wlp($P'$, $false$)
        $D := \emptyset$
        $N :=$ program points in $P$
        while $N \neq \emptyset$ do
            choose $\ell$ from N
            if Valid($R_\ell \vee \varphi$) then
                $A :=$ Doomed($P', \ell, N$)
                $D := D \cup A$
            else
                $A :=$ Absolved($P', \ell, N$)
            fi
            $N := N \setminus A$
        od
        return $D$
    end
```

Figure 8.1: Optimized version of algorithm Exorcise given in Figure 5.2

8.1 Control Dependency Graph

Let ℓ and ℓ' be program points in a program P such that the set of admissible executions passing ℓ' is a super set of those passing ℓ. Then if ℓ' is doomed in P, so is ℓ and vice versa, if ℓ is not doomed then neither is ℓ'. Our optimization is based on the simple insight that the control flow graph of a program gives us some information about the relations between the sets of admissible executions passing through different program points.

In the following we fix a program P. We say that a path in the control-flow graph of P is *complete*, if it starts in the initial block ℓ_{init} and ends in the final block ℓ_{term}. Note that a path in the control flow graph of P is not to be confused with an execution of P. A path in the control flow graph ignores the semantics of statements in the blocks of the program. Now let ℓ and ℓ' be two program points in P. We say ℓ' is *control-dependent* on ℓ, written $\ell \preceq \ell'$, if every complete path in the control flow graph that passes through ℓ' also passes through ℓ.

Lemma 1. *For any program P and program points ℓ and ℓ' in P, if $\ell \preceq \ell'$ then every admissible execution passing through ℓ' also passes through ℓ.*

The relation $\preceq$ is a reflexive and transitive relation on program points and, hence, the relation $\simeq \overset{\text{def}}{=} \preceq \cap \preceq^{-1}$ is an equivalence relation. For a program point ℓ, we denote by

$$\mathsf{T}(\ell) \overset{\text{def}}{=} \{\ell' \in P \mid \ell' \preceq \ell\}$$

the set of all program points in P on which ℓ is control dependent and we denote by $[\ell]$ the equivalence class of ℓ under the relation $\simeq$. We lift the relation $\preceq$ from program points to the quotient under the equivalence relation $\simeq$, i.e., we have $[\ell] \preceq [\ell']$ if and only if $\mathsf{T}(\ell) \subseteq \mathsf{T}(\ell')$.

We can compute the sets $\mathsf{T}(\ell)$ and thus the equivalence classes efficiently by computing the dominator and post dominator trees of the control flow graph [37, 41]. For two program points ℓ and ℓ', ℓ' is a *dominator* of ℓ if every path in the control flow graph from the initial block to ℓ passes through ℓ' and dually ℓ' is a *post-dominator* of ℓ if every path from ℓ to the final block passes through ℓ'. We denote by $\mathsf{Dom}(\ell)$ the set of all dominators of a program point ℓ and by $\mathsf{Dom}_{\mathsf{post}}(\ell)$ its post-dominators. Then we have

$$\mathsf{T}(\ell) = \mathsf{Dom}(\ell) \cup \mathsf{Dom}_{\mathsf{post}}(\ell) \enspace .$$

```
int i = 0;
if (k != 0) {
      i = 3;
} else {
      i = 5;
}
i++;
return;
```

$$
\begin{array}{l}
a : i := 0; \\
\quad \textbf{goto}\, b; \\
b : \textbf{goto}\, c, d \\
c : \textbf{assume}\, k \neq 0; \\
\quad i := 3; \\
\quad \textbf{goto}\, e; \\
d : \textbf{assume}\, k = 0; \\
\quad i := 5; \\
\quad \textbf{goto}\, e; \\
e : i = i + 1; \\
\quad \textbf{goto}\, f; \\
f :
\end{array}
$$

Figure 8.2: Program PathProg

Figure 8.3 shows the equivalence classes under the relation $\simeq$ and the dominator and post-dominator tree for the program PathProg given in Figure 8.2. In program PathProg we have $\mathsf{T}(a) = \mathsf{T}(b) = \mathsf{T}(e) = \mathsf{T}(f) = \{a, b, e, f\}$, $\mathsf{T}(c) = \{a, b, c, e, f\}$, and $\mathsf{T}(d) = \{a, b, d, e, f\}$.

Once the sets $\mathsf{T}(\ell)$ have been computed, we compute the directed acyclic graph that corresponds to the Hasse diagram of the relation $\preceq$ on the quotient, i.e., the nodes in this graph are the equivalence classes and the edges are given by the transitive reduction of the relation $\preceq$. We call this graph the *control dependency graph*. Figure 8.4 shows the control dependency graph of program PathProg.

8.2 Avoiding Redundant Theorem Prover Calls

We can now use the control dependency graph of a program to avoid redundant theorem prover calls in the algorithm $\mathsf{Exorcise}^{+}$. If two program points ℓ and ℓ' are in the same equivalence class then it follows from Lemma 1 that ℓ is doomed if and only if ℓ' is doomed. More generally, if $[\ell]$ is transitively reachable from $[\ell']$ in the control dependency graph then ℓ' is doomed if ℓ is doomed. This leads us to the following definitions of the functions Doomed and Absolved:

$$
\begin{aligned}
\mathsf{Absolved}(P, \ell, N) &\stackrel{\text{def}}{=} \{\ell' \in P \mid [\ell] \preceq [\ell']\} \cap N \\
\mathsf{Doomed}(P, \ell, N) &\stackrel{\text{def}}{=} \{\ell' \in P \mid [\ell'] \preceq [\ell]\} \cap N
\end{aligned}
$$

We also use the control dependency graph to determine the order in which

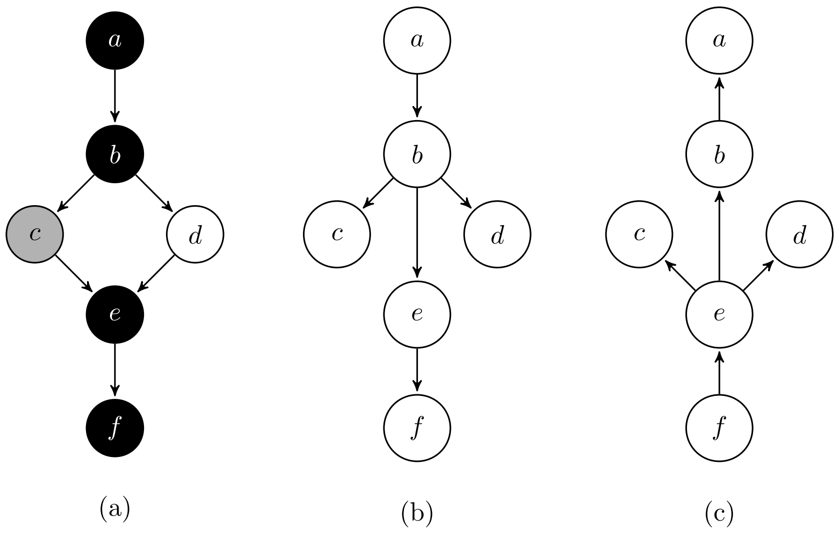


Figure 8.3: The control flow graph (a) dominator tree (b) and post-dominator tree (c) for program PathProg. The colors of the nodes in the control flow graph indicate the equivalence classes under relation $\simeq$.

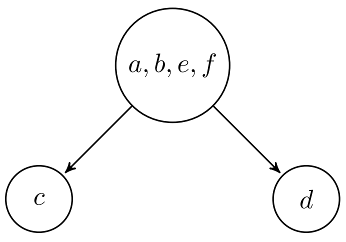


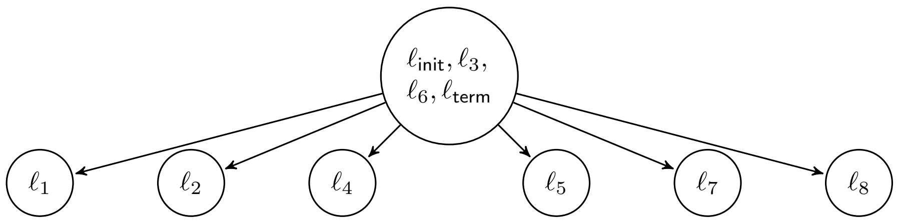
Figure 8.4: Control dependency graph of program PathProg

```
int a, b, c, x;

void diamonds(){
  if (a != 0) { x++; }
  else { x--; }
  if (b != 0) { x++; }
  else { x--; }
  if (c != 0) { x++; }
  else { x--; }
}
```

```
$\ell_{\mathsf{init}}$ : goto $\ell_1, \ell_2$;
$\ell_1$ : assume $a \neq 0$;
     $x := x + 1$;
     goto $\ell_3$;
$\ell_2$ : assume $a = 0$;
     $x := x - 1$;
     goto $\ell_3$;
$\ell_3$ : goto $\ell_4, \ell_5$;
$\ell_4$ : assume $b \neq 0$;
     $x := x + 1$;
     goto $\ell_6$;
$\ell_5$ : assume $b = 0$;
     $x := x - 1$;
     goto $\ell_6$;
$\ell_6$ : goto $\ell_7, \ell_8$;
$\ell_7$ : assume $c \neq 0$;
     $x := x + 1$;
     goto $\ell_{\mathsf{term}}$;
$\ell_8$ : assume $c = 0$;
     $x := x - 1$;
     goto $\ell_{\mathsf{term}}$;
```

Figure 8.5: Program DIAMONDS



$\ell_{\mathsf{init}}, \ell_3, \ell_6, \ell_{\mathsf{term}}$
ℓ_1
ℓ_2
ℓ_4
ℓ_5
ℓ_7
ℓ_8

Figure 8.6: Control dependency graph of program DIAMONDS

the program points are checked in algorithm Exorcise$^+$. To guarantee that a given program does not contain any doomed program points, it is sufficient to show that none of the leaf nodes in the control dependency graph is doomed. Since we expect the number of doomed program points to be small, we explore the control flow graph by starting with the equivalence classes at the leafs.

We illustrate the effect of our optimizations on the program shown in Figure 8.5. When we use the vanilla algorithm Exorcise to analyze the program DIAMONDS then we will have 10 calls to the theorem prover, one for each program point. The control dependency graph of program DIAMONDS is shown in Figure 8.6. It consists of seven nodes: one is given by the equivalence class $\{\ell_{\mathsf{init}}, \ell_3, \ell_6, \ell_{\mathsf{term}}\}$ and each of the other equivalence classes contains one of the remaining program points. The equivalence classes containing only one program point are the leaves of the control dependency graph. Applying algorithm Exorcise$^+$ to program DIAMONDS will therefore only require six theorem prover calls to prove that none of the program points is doomed.

Further Optimization. We can further improve the function Absolved by using the output produced by the theorem prover. When the theorem prover fails to prove that a program point ℓ is doomed, it emits a countermodel for the input formula. The countermodel encodes an admissible execution of the program that passes through ℓ. From the countermodel we can extract all program points $\mathsf{CE}(\ell)$ that are visited by this admissible execution. The countermodel witnesses that all program points in $\mathsf{CE}(\ell)$ are not doomed. Hence, we can remove them from the work set. For our example program DIAMONDS this further reduces the number of required theorem prover calls to at most four: if we, e.g., check the leaf node ℓ_4 of the control dependency graph in Figure 8.6, the prover will produce a countermodel corresponding to a normal terminating execution of the program that may pass, e.g., through the program points $\ell_{\mathsf{init}}, \ell_1, \ell_3, \ell_4, \ell_6, \ell_7$ and ℓ_{term}. In this case, we thus know that the leaf nodes ℓ_1 and ℓ_7 cannot be doomed either. Every other possible countermodel that the prover may produce will also eliminate at least two other leaf nodes in addition to ℓ_4.

Chapter 9

Evaluation

We implemented our algorithm Exorcise$^+$ including the optimizations presented in this thesis as part of the Boogie program verifier [3]. The implementation is publicly available for download[1]. We added a switch to Boogie which allows us to use it for doomed program point detection instead of verification. Figure 9.1 gives an overview of the architecture of Boogie. The Boogie program verifier takes a program in a specific language called *Boogie* [13] as input. This language is similar to our simple language presented in Chapter 5. Boogie provides some high level commands like specification statements for describing preconditions, postconditions, and invariants, as well as some high level constructs such as loops. However, these constructs can easily be translated to the simpler commands used throughout this thesis. Programs in that language are usually created using other tools such as Spec# that translates C# programs with special annotations into the Boogie language while adding some assertions for memory safety and other potential run-time errors. Further these tools provide axioms about the type system, data structures, and object orientation. Similar translators exist for Java and C programs as well as for some research programming languages. Boogie then transforms the input program into a loop-free program, applies a single assignment transformation and finally generates a verification condition using weakest precondition semantics [35]. The validity of this verification condition implies that all executions of the original program terminate normally.

In order to use Boogie for doomed program point detection, we have to

[1]`http://boogie.codeplex.com`

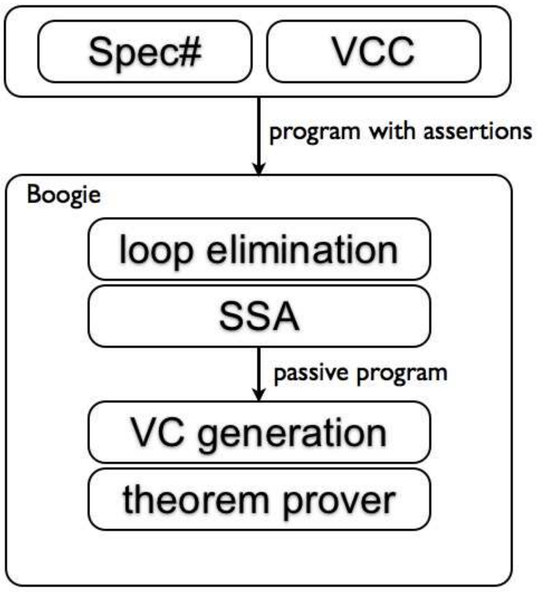


Figure 9.1: Overview of the Boogie program verifier.

modify it in a way that it proves that all executions passing a program point terminate abnormally, instead. This can be achieved by applying minor modifications to the verification condition generation and the verification procedure. First, we modify the loop elimination and program transformation used in Boogie. For a given program the loop elimination in Boogie removes the back edge of each loop and adds an **assume**(*false*) statement to the end of the loop body [35]. For the purpose of verification, this is a sound abstraction but obviously this abstraction cannot be used for doomed program point detection. We replace this loop elimination by the abstract loop unrolling presented in Section 6.1. For (recursive) method calls we use the reuse the transformation that is implemented in Boogie. We replace the function call by a non-deterministic assignment to the variables modified by the method and add an (in our case trivial) assertion of the postcondition. This is a very coarse approximation. Later on, we argue why it is a good tradeoff between precision and speed for doomed program point detection. We then augment the loop-free program with reachability variables (see Section 6.3). The reachability variables allow us to make all executions inadmissible that do not pass a certain program point simply by assuming that the reachability variable of this program point is initially zero. Thus, for this transformed program we only have to prove that all it's executions terminate abnormally (in contrast to verification, where we prove that all executions terminate normally). We next apply a single assignment transformation to the loop-free program. This transformation is the same for verification and for doomed program point detection (see Section 6.4 or [35]). For the resulting passive program we use predicate transformer semantics to compute a forumla representing the weakest liberal precondition. We can reuse the weakest precondition computation implemented in Boogie by simply changing all **assert** statements into **assume** and changing the postcondition from *true* to *false*. Boogie already implements the optimized formula generation described in Section 7.1. For program verification, Boogie generates one verification condition using the weakest precondition, sends it to the theorem prover and returns the result to the user (either a message that verification succeeded or a counterexample). For doomed program point analysis we replace the weakest precondition transformation by a weakest-liberal precondition transformation (which can be done by treating assertions as assumptions). We push the verification condition on the theorem prover stack and check if initializing one of the introduced reachability variables to zero reveals a doomed program point. We pick the reachability variables based on the optimizations presented in Chapter 8. For each of them, we send a query to the theorem prover, if assigning the variable to zero implies the verification condition.

Finally, we return a list of doomed program points. This list only indicates which statements cannot be reached on normal terminating executions. It does not identify errors or distinguish between the violation of an assertion or a reachability problem. We can, e.g. use the approach sketched in Section 7.3 to identify statements in the source program that can be blamed for the error and be used to generate an error message. For our benchmark in the following Section, we only compute doomed program points without generating error messages.

In the following we present an evaluation of our implementation. We conduct two experiments. First, we compare the detection rate of our algorithm to other error detection tools. Then we evaluate the performance of our implementation and the effect of our optimizations.

```
void diamond(int a, int b,
             int c, int y) {
   x = y;
   if (a != 0) { x++; }
   else { x--; }
   if (b != 0) { x++; }
   else { x--; }
   if (c != 0) { x++; }
   else { x--; }
   assert x==y+3;
}
```

Figure 9.2: DiamondsErr

9.1 Detection Rate

We compare our implementation of Exorcise$^+$ to Findbugs [28] and Boogie [36] with the /smoke option. When the /smoke option is enabled, Boogie uses static reachability analysis [31] to identify parts of a seemingly correct Boogie program that are unreachable. In particular, smoke testing can detect code fragments that are vacuously correct due to errors in the specification. We apply all tools to the example programs given throughout this thesis and to programs from the Findbugs null pointer micro benchmark [28]. For Findbugs we use Java versions of the programs. For Exorcise$^+$ and Boogie we use handwritten Boogie programs for the examples from this thesis and a Boogie program generated by Spec# for the Findbugs micro benchmark. While the programs are all small (only a few lines of code), they cover a variety of common errors in programs and sources for potential false positives.

The result of our comparison is shown in Table 9.1. For all benchmark programs our analysis can correctly determines whether the program has an error, i.e., all errors in the examples actually cause some program point to be doomed. Findbugs produces false negatives on three programs with errors. Boogie detects all errors but also produces 5 false positives.

In particular, this benchmark shows that in the case that both Boogie and Exorcise$^+$ do not detect errors, Exorcise$^+$ is always faster.

The C# version of the Findbugs benchmarks and the benchmarks from this thesis are provided in two separate input files to Boogie (both when used with our algorithm and when used with smoke testing). Boogie is restarted for each

Program	error?	Exorcise$^+$		Findbugs [28]	Boogie [3]	
		t (ms)	result	result	t (ms)	result
fp1	no	168	true neg	true neg	756	true neg
tp1	yes	76	true pos	false neg	35	true pos
fp2	no	71	true neg	true neg	159	true neg
tp2	yes	68	true pos	false neg	26	true pos
fp3	no	81	true neg	true neg	185	true neg
tp3	yes	90	true pos	true pos	57	true pos
tp4	yes	75	true pos	true pos	38	true pos
fp4	no	71	true neg	true neg	129	true neg
tp5	yes	62	true pos	true pos	25	true pos
tp6	yes	63	true pos	true pos	20	true pos
Vacuous	yes	108	true pos	true pos	267	true pos
Entangled	yes	15	true pos	true pos	3	true pos
Access	yes	12	true pos	true pos	1	true pos
GetMin	yes	38	true pos	false neg	27	true pos 3 false pos
Update	no	24	true neg	true neg	11	2 false pos
DiamondsErr	yes	15	true pos	true pos	3	true pos
Diamonds	no	16	true neg	true neg	32	true neg
Total Time		1.1 s		3s	1.87 s	

Table 9.1: Comparison of our algorithm with Findbugs and Boogie with smoke testing enabled. The columns list the analyzed program, whether it contains an error, and the running time and result for each tool. The result can either be "true positive" if an error is found, "true negative" if no error is reported on correct programs, "false positive" if a non-existing error is reported, or "false negative" if an existing error is overlooked. Programs fp$i$ and tp$i$ come from the Findbugs null pointer micro benchmark and the remaining programs from this thesis.

Program	LOC	#blocks	#checked blocks			total time (s)		
			All	Incr	None	All	Incr	None
Tree	7366	551	84	551	551	17.46	113.20	209.86
RB Tree	6981	435	137	453	453	111.63	279.00	298.25
RwLock	7485	181	30	181	181	507.65	579.20	705.65
NestedLock Pattern	8391	252	54	252	252	63.90	554.46	524.58

Table 9.2: Performance benchmarks on four C programs. The columns show the name of each program, the total number of lines of code of the Boogie program, total number of basic blocks, the number of blocks that were checked during the analysis and the total running time of the analysis. We differentiate between three different runs of the analysis on each program: one without any optimizations (None), one with incremental theorem prover calls (Incr), and one with incremental checking and the optimizations discussed in Chapter 8 (All).

input file. The initialization time of Boogie accounts for a small increase in the running times for the programs fp1 and `Vacuous`, which happen to be the first programs in the input files. The running times are given for the sake of completeness only. The programs we consider here are too small to make a meaningful comparison between the tools. Experiments with larger examples indicate that Findbugs tends to have better running time than both Exorcise$^+$ and Boogie with smoke testing. However, our approach still scales reasonably well, as we shall see in the next experiment.

9.2 Performance Benchmarks

We next evaluate the performance of our implementation. We apply our algorithm to four C program fragments taken from an operating system kernel. The program fragments are translated to Boogie using VCC [16]. The programs implement two different tree data structures, a reader-writer lock [25], and a nested lock pattern implementation. The programs are partially annotated with specifications.

Table 9.2 shows the results of our experiments. We can see that our prototype implementation already scales quite well to larger programs. We analyzed each program without optimizations, only with incremental checking enabled, and with all optimizations enabled. Our optimizations could significantly improve the running time of the analysis. Only for the reader-writer lock implementation (RwLock) the effect of our optimization was hardly visible. A more detailed analysis of this example revealed that 90% of the computations time is spent by proving the correctness of two loop invariants that are annotated in the program. Thus our optimizations only affected the remaining 10%.

Incremental checking turned out to be slower on the NestedLock Pattern example than the naive approach. We found out that in certain cases the theorem prover gets slowed down by useless instantiations of axioms that are introduced in the translation to the BOOGIE program in order to formalize the sementics of C. In principle, we could circumvent this problem by restarting the prover if we detect that incremental checking slows down.

Our implementation detected a doomed program point in an early version of the tree implementations which was caused by a wrong data structure invariant. We also detected a few doomed program points caused by intensional program abortions (i.e., **assert** *false* statements) in all four programs. We believe that the latter should not actually be reported to the user. However, this is only a problem of presentation. As we expected, it is unlikely to detect actual doomed program points in legacy code as this would indicate that some parts of the code have never been executed.

9.3 Discussion

In general, we think that our implementation for doomed program point detection is useful already. However, in the current workflow of the Boogie program verifier, there is only little application for it. Boogie is used to prove correctness. If the verification engineer wants to be sure that proof of the program is not valid because of vacuous specifications she can use the *Smoke* test, that is build into Boogie [31] which results in a higher error detection rate but some false positives.

Doomed program point detection is meant to be run during compile time and support the error detection methods of the compiler. Our experiments indicate that it scales good on the method level. As our analysis does not claim to be complete, we believe that there is still a lot of room for improvements related to performance. In the very first place a simpler memory and type model could safe a lot of time. Programs translated from high-level programming languages such as C or C# to our intermediate language carry a prelude of several thousand lines with them that describe the type system and the memory model. This prelude is designed for proving correctness which requires a much higher precision than error detection. We plan to rerun our experiments with a slightly simpler prelude.

The abstract loop unrolling generates three copies of the loop body. One simulating the first, the last, and an arbitrary number of iterations. In our experiments the doomed program points found in loops were always caused by statements in the last iteration. The size of the formula can be reduced significantly by just using a non-deterministic assignment and the last iteration of the loop body. Again, this is a decision between precision and performance, and we will perform some empirical analysis to identify the most suitable abstraction for doomed program point detection. We could further improve the performance or detection rate by either using a simplified loop unrolling or a more sophisticated loop elimination (e.g., by computing invariants to further strengthen the abstraction). However, this is a matter of optimization and exceeds the scope of this work. Closely related to this, we found that the inlining of function calls is of little help for doomed program point detection. Our only experiments where we detected doomed program points using inlining were small functions that perform numeric computations without loops or nested function calls. For our future work, we plan to replace the inlining be some kind of summary computation.

Overall, our experiments show that doomed program point detection is only useful for small program fragments as doomed program points tend to be a local problem. This supports our initial motivation that this analysis should be used during development as an extension to existing compiler checks. For legacy code it is very unlikely to find a statement that cannot be passed by a normal terminating execution as this would indicate that this statement has never been tested. We argue that, if we can prove that a statement in a program fragment has no normal terminating executions, we can prove it for any larger fragment as well. Increasing the size of the fragment increases the size of the verification condition but also helps to reduce the set of possible executions passing the statement and thus makes it easier to prove that it is doomed. However, in a real program, increasing the size of the fragment will at some point not help to reduce the executions passing a statement (e.g., because the program fragment already contains abstraction, such as non-deterministic assignments, such that a further extension of the scope will not reduce the number of possible executions of that fragment). Finding the right size of the scope means finding the sweet spot between performance and precision. From the experiments we can see that the method scales up to large methods but does not find sophisticated errors (as those are usually not doomed program points).

The experiments indicate that our implementation is fast and precise for small program fragments so that we conclude that the presented implementation is a useful tool for error detection during compile time.

Chapter 10

Conclusion and future work

This thesis is motivated by the idea that in practice program verification is rather considered as a way to detect bugs than to prove correctness. Achieving correctness with respect to (eventually informally) given requirements is a very hard problem but the verification procedure itself and the automated tools supporting this process can help to identify previously unspotted errors. Using verification techniques to prove the presence of errors instead of their absence promises powerful tools that can be applied by a broad audience. Precise static detection of errors as performed by todays compilers is maybe one of the most powerful and widely used approaches towards better software quality. Therefore, we believe that sound static error detection using software verification is a new and promising research direction.

The main contribution of this work is the new idea of doomed program points. We argued that doomed program points constitute an interesting class of program errors that should be detected at compile time. We showed that such a detection algorithm can be realized efficiently in practice an that the algorithm can easily be integrated in existing extended static checkers and program verifiers, assuming they provide the infrastructure for generating verification conditions and automatic theorem provers to check them. We believe that our idea can now be adopted and extended by many others. We see a huge potential in this work. It provides a formal method that is applicable by every programmer without prior knowledge of how to formally specify the correctness of a program. Yet, given that our technique can be integrated into full-fledged program verifiers such as Spec#, the programmer can directly benefit from additional specifications that she puts into the program, as such annotations gradually

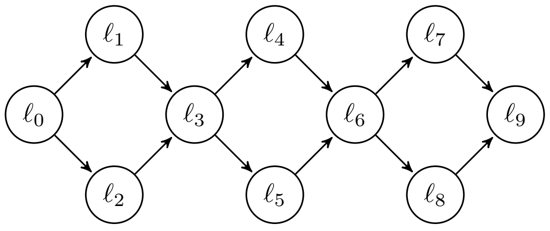


Figure 10.1: CFG for Diamond

extend the range of errors that our algorithm can detect - without producing noise. Our technique can therefore provide a smooth learning curve towards the use of full program verification and allow static verifiers to be integrated in current IDEs.

Our experiments show, that our approach is not only useful to detect error in code in a very convenient way, it can also be used to detect errors in user provided specifications and thus is very useful to any verification engineer working with a static verifier.

Maybe the best reason to use our approach is that there is no argument against it: our method is fully automatic and it remains invisible to the user as long as no doomed program point is found. If a warning is emitted, then this is a definite indication that the program is incorrect.

Future work We see much room for further improvements of our method. For instance, we want to optimize our detection algorithm by developing specialized techniques for finding *correct* executions, so that error verification conditions are quickly recognized as invalid. Doomed program points are sparse; i.e., almost all generated error verification conditions are not valid in practice (this is in contrast with the usual verification conditions, for correctness). Every programmer's experience confirms the intuition that it is easier to find a correct execution (for a program fragment that has no guaranteed error) than to find an incorrect one (for a program fragment that may lead to an error). This gives an interesting potential for further optimization.

Doomed program point detection and testing. One of the most promising aspects of doomed program point analysis is that, if we cannot prove that a program point is doomed, the theorem prover provides us with a counterexample that includes initial values of the program variables that can lead to an admissible execution. Unfortunately, since the algorithm is not complete, the values might not correspond to a possible execution in the original program. However, we believe that there are several ways to exploit this feature.

Doomed program point detection can be combined with testing to analyze program fragments that are not reached by a certain number of test cases. For these fragments doomed program point analysis can check if there are admissible executions passing this program fragment (and possible be extended in a way that it returns a test case for one of these executions by parsing the model produced by the theorem prover) or give an error message that shows, why there are no admissible executions passing the fragment. If an admissible execution passing this fragment exists, our implementation will fail to prove it doomed and thus, the theorem prover will provide a model with a valuation of the program variables the can reach this fragment. In particular in combination with random testing this approach looks promising as the counterexample provided by the prover can help to seed the random test case generation and thus lead to a higher or faster test coverage of the examined program. In some way, this approach is dual to the idea of may-must analysis where static may analysis is used to prove properties of the program and dynamic must analysis is used to refine the static analysis while our approach uses dynamic analysis to detect bugs and applies static doomed program point detection on fragments to provide seed values for the test case generation.

Another application is to check for several doomed program points at the same time. That is, instead of asking if there is no admissible execution passing through a program point ℓ we ask, e.g. if all executions passing through several program points ℓ_1 to ℓ_n are inadmissible. This is only a very small change in our implementation but allows us to check much more. We illustrate this using the control flow graph of the DIAMOND program given in Figure 10.1. E.g. the program point ℓ_5 is doomed if all 4 possible executions passing the point are inadmissible. If we want to focus on one particular execution we can e.g. check if the program points conjunction of ℓ_1, ℓ_5 and ℓ_7 is doomed. Or we could check ℓ_1 and ℓ_5 to see if the two executions that are passing through both of them are inadmissible. Using this approach we can easily focus our analysis on a certain set of executions by checking several program points at the same time. This approach allows finding more sophisticated errors in a program. However, a heuristic has to be found to select interesting program points. Otherwise, the analysis may report executions that are intended to be inadmissible (e.g. conditionals which are mutual excluding each other) and checking each control flow path in isolation would be too costly (see [23]). Further, this approach can be used to generate test cases that reach certain points in the program which can be used to direct the test case generation in a particular way.

Error message construction. In this thesis, we have presented a brute force implementation that extract the contradiction statements in our formula and reports the corresponding statements in the program back to the user. This can be further investigated. Essentially an error message for a doomed program point can be seen as an explanation (or compact representation) of the prove of a safety property (i.e., the property that some location is never reached). This is an invariant problem and we plan to use existing techniques to infer invariants to investigate if they are suitable as a basis for error messages for doomed program points. In contrast to verification where the goal is to find the strongest invariant, we are interested in finding the weakest invariant for which the property still holds.

Extended error detection. Static verifiers like Boogie compute the weak precondition of a program which describes all pre-states, such that the execution of the program ends in a normal terminating state when started in such a prestate. The complement of this is a condition on the initial states such that all executions cannot terminate normally. Using this condition as a precondition for the program guarantees that the program has no normal terminating executions and thus is doomed. We can now prove this program doomed and a technique to identify the contradicting statements (e.g., the brute force implementation from Section 7.3). If we can identify a statement contradicting with the artificially added precondition, we know that this statement in someway contributes to the abnormal termination of the program. However, to make use of this feature there has to be a clear notion of error messages for doomed program points, which then can serve as a basis for general error detection.

In the algorithm presented throughout this thesis, we require that, *for all* valuations of the input variables, the execution of the program either terminates abnormally or does not pass the considered program point. Instead, we could distinguish between *angelic* and *demonic* variables, where angelic variables, just as in the presented approach, try to pick a valuation that allows normal termination and demonic variables try to pick an initial value the leads to abnormal termination. Of course this sacrifices soundness but allows to focus on certain variables and thus can be used, e.g. for robustness analysis of methods.

We conclude that doomed program points are a valuable extension to the current state of research in static error detection. The presented algorithm can support even an unexperienced programmer as it does not require knowledge about formal methods, yet it allow a verification engineer to detect contradictions in complex invariants. Further, observing the tendency that formal methods are more and more used for error detection rather than for proving correctness we believe that doomed program point detection is a valuable contribution to the community and can play an important role in the evolving research on combinations of static and dynamic analysis.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Workshop on Program Analysis for Software Tools and Engineering, PASTE*, pages 1–8. ACM, 2007.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[4] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Workshop on Program Analysis for Software Tools and Engineering, PASTE'05*, pages 82–87. ACM, 2005.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[6] Ilan Beer, Shoham Ben-david, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in actl formulas. In *FMSD*, pages 279–290. Springer-Verlag, 1997.

[7] Richard Bornat. Proving pointer programs in hoare logic, 2000.

[8] Rod M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Learning*, 7, 1972.

[9] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, February 2009.

[10] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Contract precondition inference from intermittent assertions on collections. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI*, 2011.

[11] Coverity. Coverity Prevent$^{\text{TM}}$User's Manual 2.4, 2006.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[13] Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[14] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[15] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electr. Notes Theor. Comput. Sci.*, 217:5–21, 2008.

[16] Markus Dahlweid Ernie Cohen, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics, TPHOLs'09*, pages 23–42, 2009.

[17] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[18] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification, CAV'07*, pages 173–177, 2007.

[19] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation, PLDI'02*, pages 234–245. ACM, 2002.

[20] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'01*, pages 193–205. ACM, 2001.

[21] Nicu G. Fruja. The correctness of the definite assignment analysis in c#. *Journal of Object Technology*, 3(9):29–52, 2004.

[22] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Annual ACM Symposium on the Principles of Programming Languages, POPL*, pages 43–56. ACM Press, 2010.

[23] Ian J. Hayes, Colin J. Fidge, and Karl Lermer. Semantic characterisation of dead control-flow paths. *IEE Proceedings - Software*, 148(6):175–186, 2001.

[24] Thomas A. Henzinger, Ranjit Jhala, Rubak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Model Checking Software, 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.

[25] Mark A. Hillebrand and Dirk C. Leinenbach. Formal verification of a reader-writer lock implementation in c. *Electron. Notes Theor. Comput. Sci.*, 254:123–141, 2009.

[26] Jochen Hoenicke, K. Rustan Leino, Andreas Podelski, Martin Schäf, and Thomas Wies. It's doomed; we can prove it. In *FM '09: Proceedings of the 2nd World Congress on Formal Methods*, pages 338–353, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] Jochen Hoenicke, K. Rustan Leino, Andreas Podelski, Martin Schäf, and Thomas Wies. Doomed program points. *to appear in Formal Methods in System Design*, 2010.

[28] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Workshop on Program Analysis for Software Tools and Engineering, PASTE*, pages 9–14. ACM, 2007.

[29] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *ACM SIGSOFT Software Engineering Notes*, 31(1):13–19, 2006.

[30] Neil Immerman, Alexander Rabinovich, Thomas Reps, Mooly Sagiv, and Greta Yorsh. The Boundary Between Decidability and Undecidability for Transitive-Closure Logics. In *Computer Science Logic, CSL'04*, pages 160–174, 2004.

[31] Mikoláš Janota, Radu Grigore, and Michal Moskal. Reachability analysis for annotated code. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 23–30, New York, NY, USA, 2007. ACM.

[32] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 19(6):1031–1052, 1997.

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

[34] Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[35] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters, IPL*, 93(6):281–288, 2005.

[36] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, June 2008. Available at http://research.microsoft.com/~leino/papers.html.

[37] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[38] David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 1(2):226–244, 1979.

[39] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, 1995.

[40] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 11(4):517–561, 1989.

[41] Reese T. Prosser. Applications of Boolean matrices to the analysis of flow diagrams. In *IRE-AIEE-ACM'59 (Eastern)*, pages 133–138. ACM Press, 1959.

[42] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *International Symposium on Software Reliability Engineering, ISSRE*, pages 245–256, 2004.

[43] Marko Samer and Helmut Veith. On the notion of vacuous truth. In *LPAR'07: Proceedings of the 14th international conference on Logic for programming, artificial intelligence and reasoning*, pages 2–14, Berlin, Heidelberg, 2007. Springer-Verlag.

[44] Vladimir I. Shelekhov and Sergey V. Kuksenko. On the practical static checker of semantic run-time errors. In *Asia Pacific Software Engineering Conference, APSEC*, page 434, 1999.

[45] Polyspace Technologies. PolySpace for C. Documentation, 2004.

[46] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *ACM SIGSOFT Software Engineering Notes*, 29(6):97–106, 2004.