Universität Ulm
Fakultät für Mathematik und Wirtschaftswissenschaften
Institut für Angewandte Informationsverarbeitung

Static Code Analysis in Multi-Threaded Environments

Dissertation

zur Erlangung des Doktorgrades Dr. rer. nat. der Fakultät für
Mathematik und Wirtschaftswissenschaften der Universität Ulm

vorgelegt von

Christian Ehrhardt

aus Mutlangen

26. Oktober 2007

Amtierender Dekan:	Prof. Dr. Frank Stehling
Gutachter:	Prof. Dr. Franz Schweiggert
	Prof. Dr. Helmut Partsch
	Prof. Dr.-Ing. Stefan Jähnichen
	Dr. Johannes Mayer
Tag der Promotion:	26. Oktober 2007

Who can understand his errors?

Cleanse thou me from secret faults.

Psalms 19.12, The Bible

Preface

The notion that software might contain errors dates back to the famous annotations of *Lady Ada Lovelace* to the description of the *Analytical Engine* designed by *Charles Babbage*[36] where she states: "Granted that the actual mechanism [of the Analytic Engine] is unerring in its processes, the cards may give it wrong orders. This is unquestionably the case; but there is much less chance of error, and likewise far less expenditure of time and labour, where operations only, and the distribution of these operations, have to be made out, than where explicit numerical results are to be attained."

Despite Lady Lovelace's assessment, errors in software (i.e. today's equivalent of cards that give wrong orders) have become a predominant concern in software development and software engineering. Nowadays, the vast majority of software engineers and hackers alike believes that all sufficiently complex software has bugs. Solutions that have been proposed to uncover bugs reach from testing on the one hand to rigorous formal proofs of correctness on the other hand.

In this work we follow an intermediate approach that tries to use sound and conservative static code analysis techniques to avoid certain classes of bugs without having to conduct a full blown formal proof of correctness. The methods developed in this work are tailored to concurrent systems where global data is shared between all threads. While this requirement is not strictly necessary for the soundness of the analysis we shall see that it greatly increases the accuracy. In this context accuracy refers to the likelihood that a problem reported by the analysis is caused by an actual bug in the software and not by an inadequacy of the analysis methods. The usefulness of the methods presented will be supported by several case studies conducted on real life software systems of significant size.

Nevertheless, I am convinced that in software engineering there is no Holy Grail that software can gain eternal bug free life from. As a corollary this Holy Grail cannot be expected to be found in this work. Thus, readers are encouraged to use the methods described in this work to supplement but not to replace existing bug prevention and software quality techniques.

Acknowledgments

This work has been developed at the Institute of Applied Information Processing at the University of Ulm. I would like to express my gratitude to all my present and former colleagues for the working atmosphere which has always been very constructive, open, and friendly. This would not have been possible without the extraordinary commitment of Prof. Dr. Franz Schweiggert, the director of the Institute, to his employees. Likewise, this work would not have been possible without his guidance and support. Thank you very much for always being there when needed.

Prof. Dr. Helmut Partsch has kindly agreed to report on this thesis which I am very grateful for.

Regarding this work, many people have given helpful suggestions and constructive criticism that has been very much appreciated. Especially, I am much obliged to Dipl. Inf. Walter Guttmann and Dr. Johannes Mayer for valuable discussions. I would also like to thank the latter for agreeing to reporting on this thesis.

Special thanks go to Dr. Andreas Borchert. He has always been open for indepth discussions and I have learned a lot from him.

Apart from those already mentioned, I would like to thank Dr. Ingo Melzer for proof reading and Dipl. Math. oec. Uta Phillip for proof reading and her love, faith, and patience.

Last but not least, I would like to thank my parents for continued and insisting encouragement and support.

Contents

Chapter 1

Introduction

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence." This famous quotation of Edsger Dijkstra[16] pinpoints a fundamental problem of software testing: No matter how many tests pass successfully and no matter how rigorous the testing process is, it is always possible that the first real test in the field triggers a bug[1] which has not been found during testing. Additionally, the software system can be influenced by external events like interrupts, input from external sources, other concurrent processes, or anything else that can make the behavior of a software system effectively non-deterministic. Thus it is possible that scenarios, that have been tested successfully before, start to fail sporadically or even systematically.

In other words: If a certain test is run and the test fails[2] we can be sure that the program has a bug. Then again, if a test passes we can only conclude that the program behaved correctly at least once for this single test case. Thus, running a single test case is practically useless from a software quality point of view. Of course, running many tests and fixing the bugs found in this process (without introducing new ones) does increase the quality of the software system being tested significantly.

Still, many interesting questions cannot be answered by testing:

Are there still bugs? This is the question we are actually interested in and testing by its very nature cannot give a definitive answer. Under some circumstances it is possible to estimate the number of bugs remaining based on statistical considerations and the rate at which bugs have been found during testing (see [39]). Such methods can never give a definitive answer though.

What is the cause of a bug? Testing does not identify the cause of a bug, it

---

[1] Some authors distinguish between some or all of the terms *anomaly*, *bug*, *defect*, *error*, *fault*, *failure*, etc[26]. In the rare cases where we need to make use of such a distinction in this work, we do so explicitly, e.g. by referring to the symptoms vs. the cause of a bug or error.

[2] i.e. the program behaves in an unexpected way

merely identifies the symptoms. Depending on its nature, manifestation of a bug's symptoms and its root cause can be far away from each other both in terms of time and in terms of source code location.

Will a successful test always succeed? The behavior of a program is not necessarily deterministic. Thus, a test that completed successfully once does not guarantee that the same test will always pass.

Are all relevant scenarios covered by tests? Even if the set of test cases is selected carefully, there is a fair chance that a certain type of bug only manifests itself under circumstances that were not tested. E.g. a bug might only show on certain hardware platforms, under heavy system load, or at a certain time of the day.

Thus, supplementary approaches to improve the quality of software are often useful. In order to find answers to the questions outlined in this section, we will consider several characteristics of bugs that significantly influence the probability that these bugs can be discovered by testing. This will allow us to identify classes of bugs that are likely to be missed by testing. This knowledge can then be used to develop specialized techniques to detect and prevent these bugs.

1.1 Software Bugs

This section is not intended to be an exhaustive classification of software bugs. It merely lists some important characteristics of bugs that influence the methods and techniques that are appropriate to prevent certain classes of bugs.

1.1.1 Possible Consequences of Bugs

Probably the most obvious question that can be asked about a certain kind of bug is "What can happen if the bug is triggered?" E.g. a missing check for a NULL-pointer will usually lead to an immediate crash once the pointer is accessed. Actually, from a developer's point of view, this consequence is not so severe because there is usually plenty of information readily available that can help to locate the bug (e.g. crash dumps and back traces). Thus if the bug is triggered at least once during testing, there is usually enough information available to locate and fix the bug.

Another very common symptom of a bug especially in multi-threaded systems is that the system hangs without processing any requests. This is worse than an immediate crash because there is much less information available to debug the cause of the problem. However, in such a case it is often possible to trigger a crash by manual intervention and the data produced by the crash can give a good clue as to why the system hangs.

However, some bugs can lead to the corruption of data without an immediate system crash or hang. Such a bug potentially can allow an attacker to take over the system or might lead to accidental deletion of business critical files. This is often much worse than an occasional system crash where normal operation can be resumed by simply restarting the system. In fact, one would likely prefer a crash (e.g. as a consequence of an assertion failure) to a compromised system or loss of important data in the scenarios outlined above.

1.1.2 Reproducibility

It is very important to know if a certain bug can easily be reproduced, i.e. there exists a well defined procedure or input that always triggers the bug. This property is more or less a prerequisite if we are trying to find the bug by testing because otherwise the bug or its symptoms might simply disappear once the test is run again.

A reproducible bug in general is relatively easy to debug because a developer can easily gather more information about the sequence of events that lead to the bug by reproducing it in a controlled environment, e.g. a debugger. Additionally, once a reproducible bug has been fixed, the steps necessary to reproduce the bug can be added to the regression test suite which effectively prevents the bug from reappearing in future versions of the software. Race conditions[3] in a multi-threaded system are a typical example of a bug that is hard to reproduce.

Bugs that are not reproducible are very rare in traditional single threaded programs. E.g. the bug reporting instructions for the GNU Compiler Collection[18] state explicitly that errors that only occur some of the time that a file is compiled should not be reported because this type of error is usually caused by faulty hardware.

1.1.3 Latency of Symptoms

Another important characteristic of a bug is the time that it takes until its symptoms become visible to the user. E.g., consider a compiler that generates wrong code. This causes a user program that is compiled with this compiler to crash. There are three important points in time with regard to this bug:

- The root cause of the bug is introduced into the source code of the compiler.
- The bug is triggered while compiling another program.
- The symptoms of the bug become visible when the user program crashes.

---

[3] I.e. abnormal behaviour of a program that is triggered by a particular, usually rare temporal sequence of events

It should not be too surprising that the introduction of a bug into the compiler goes unnoticed for quite some time. This is a common characteristic of a more subtle bug. The important point is that the fact that the bug has actually been triggered can also remain unnoticed for a potentially long time if the miscompiled user program is not executed, or not executed with input that actually makes the symptoms of the bug visible.

In the context of operating system kernels, lacking or incorrect synchronization of shared data is a common example of a bug that can have a long latency of symptoms.

These bugs tend to have severe consequences, e.g. silent corruption of data over a longer period of time, exactly because the fact that something went wrong remains unnoticed. Additionally, such a bug is usually hard to debug, because once the symptoms of the bug become visible they often have no obvious connection to the root cause of the bug.

1.1.4 Probability of Discovery

Finally, we can ask if it is likely that a certain kind of bug is discovered by a reasonably chosen set of system tests. E.g. in order to trigger a race condition it is usually necessary that the system is under a very specific work load leading to a repeated parallel execution of the affected code paths. While it is certainly possible to add tests that generate a specific work load to a test suite, it is often impossible to identify all workloads that might be problematic. Thus, it is likely that many of these bugs are not discovered by system tests.

Conversely, a missing check for a boundary case (e.g. a missing check for a NULL-pointer) is much more likely to be triggered by a system test case that includes the specific boundary case.

1.1.5 Conclusions

From the previous sections we can conclude that bugs which are hard to reproduce under lab conditions are often also likely to be missed by system tests. Bugs that are not reproducible are much more likely to happen in software systems that are multi-threaded, i.e. several flows of control are executed simultaneously and share their global data. Additionally, bugs that are specific to multi-threaded systems tend to have other characteristics that make a bug particularly severe. E.g. typical synchronization problems in multi-threaded systems often lead to silent data corruption that is not discovered immediately. In the previous sections we have seen that this makes a bug particularly expensive both with regard to potential consequences and with regard to the cost of finding and fixing the bug.

Thus multi-threaded systems offer a wide range of opportunities to apply bug prevention techniques that are different from testing.

1.1.6 Approach Used in this Work

In this work we will always assume that we are dealing with multi-threaded systems. Theoretically, the methods and algorithms presented will also work in single-threaded systems. Having said that, we optimize for multi-threaded environments, i.e. in the single threaded case the methods will tend to be less effective in finding bugs and the results produced for single threaded systems are less precise. We will concentrate our efforts on classes of bugs that are hard to find and fix by means of classical testing techniques and it is assumed that some basic knowledge about the type of bugs we are interested in exists from the beginning. Using this knowledge, we concentrate our efforts on this type of bugs and ignore other possible errors. That is, if we are interested in bugs related to synchronization we simply assume that e.g. NULL-Pointer dereferences do not occur or we conclude that a pointer that is dereferenced cannot be NULL.

In order to deal with these bugs we will develop automated analysis techniques that allow us to model the execution of a multi-threaded program conservatively yet quite accurately. Different versions of the Linux and the NetBSD kernels are used to test the effectiveness of the algorithms that are developed. This shows the suitability of the approach for real life software systems.

The algorithms presented in this work are not specific to operating system kernels, however, operating system kernels are particularly suited as test candidates for two reasons: Firstly all modern operating system kernels are multi-threaded, i.e. at least conceptually there are many control flows that operate on the same data in parallel. Secondly operating system kernels are self-contained, i.e. they do not depend on external library functions. The latter is important because a conservative analysis either needs user provided descriptions for all external library functions or all external libraries must be considered along with the program itself.

Similarly, the methods are designed to be independent of a specific programming language. Still, some definitions and design decisions are motivated by the fact that our case studies use programs written in the C programming language.

The actual semantics that must be enforced to prevent a certain bug must be given in the form of a user supplied specification. E.g. the requirement that a certain lock must be held in order to access the contents of a given structure must be provided by the user. This specification can then be checked with the help of the general analysis framework that is developed in the first part of this work. The second part illustrates the process of verification with the help of concrete examples.

1.2 Related Work

The observation that mere testing is often insufficient to prevent software bugs is not new. This section gives an overview of the solutions that have been proposed.

1.2.1 Targeted Generation of Test Cases

A lot of work has been done in the realm of testing that aims to make sure that the test suite covers all relevant cases. A classical example is the use of code coverage metrics[40] in order to make sure that the test suite covers all parts (statements, paths, etc.) of the system. However, their significance in multi-threaded systems is limited because the effects of concurrency are not accounted for.

Other approaches try to derive suitable test cases from the source code or from a formal specification even in the context of concurrent systems. E.g. in [48], an algorithm for the generation of test cases based on the formal specification is given, that is suited for concurrent systems. Tests on concurrent systems can also be conducted in a special controlled environment, where a certain temporal sequence of events can be triggered artificially. If such a testing environment is available it is possible (see e.g. [9]) to select specific sequences of events called synchronization sequences based on a formal description of the system in the form of a Petri Net.

We will not pursue this class of approaches further because the focus of this work is on supplementary analysis techniques that are not based on the execution of test cases.

1.2.2 Proof of Correctness

The classical approach to guarantee the absence of bugs is to prove the correctness of a program by means of formal methods. E.g. Hoare[23] proposed a formal calculus that can be used to reason about a program. Using this calculus, it is possible to prove certain properties of programs by means of mathematical methods. Based on the *Hoare calculus*, Dijkstra suggested *predicate transformer semantics*[17] as a means to prove the correctness of programs.

These approaches specify a calculus that can be used to describe the actual semantics of individual statements in a program formally. This description can then be used to prove theorems about the program as a whole. The ultimate goal is to express the intended semantics of the program as a theorem in the calculus and to provide a proof for this theorem. However, the calculus itself does not help in actually conducting the proof. This can be a tedious task and it cannot be fully automated. In fact some (probably most) interesting properties of a program cannot be decided automatically by a computer program in all cases. There simply is no such program. The most well known problem of this kind is most likely the so called *halting problem*[49].

Despite this, theorem provers have been able to prove substantial non trivial properties of programs in a more or less automated way ([28],[44]). However, a rigorous proof of correctness that takes all aspects even of a moderately sized program into account is not feasible with today's technology. Additionally, it is often not even clear what we should try to prove. The semantics of a function or program is usually defined in terms of an informal verbal description and in many cases it is not at all obvious how this description can be expressed within a formal calculus. Furthermore, the fact that an attempt to prove the correctness of a program fails does not necessarily mean that the program is erroneous. The reason that a proof does not succeed can either be that the techniques used to conduct the proof were not clever enough or more fundamentally that the calculus is not strong enough to express the proof.

Finally, a proof of correctness only shifts the task of checking the code to the task of checking the proof which can sometimes be difficult and error prone as well. Donald E. Knuth[29] has put it this way: “Beware of bugs in the above code; I have only proved it correct, not tried it.”

Altogether, a formal proof of correctness is virtually always desirable but it is usually not feasible to conduct such a proof for larger systems. Even if it is feasible, finding the proof is often tedious and cannot be automated.

1.2.3 Static Code Analysis

Static Analysis[41] is another approach to assess the quality of software. It does not suffer from most of the practical problems that arise in the context of correctness proofs. In static analysis, the source code is analyzed statically (i.e. without executing the code) for code patterns that are known to be erroneous or error prone. As a consequence, static analysis needs a priori knowledge of bad coding practices that should be reported as potential errors. Several tools for static analysis are available either freely or as commercial products. The types of errors that can be detected heavily depend on the tool.

One of the classical tools for static analysis is the *lint* program checker[27] for C source code. State of the art techniques for static analysis can do a lot more than lint though. E.g. Engler et. al.[19][10] suggest a technique they call meta-level compilation to perform customized static analysis on large programs. Meta-level compilation extends the compiler that is used to build object code in a way that allows the developer to specify additional semantic requirements by means of a finite automaton. State transitions are triggered by certain program constructs. If the automaton reaches a state that indicates an error a diagnostic message is given. This approach has evolved into a commercial static analysis tool called *Coverity Prevent*[24].

In [20] it is suggested that part of the rules used for static analysis that normally need to be supplied by the developer can be inferred from a larger program. E.g. a function that returns a pointer that is frequently compared to NULL upon return from the function can probably return a NULL pointer legally. If the return value of such a function is dereferenced without a check this is probably a bug in the code.

Microsoft internally uses static analysis tools called *PREfix* and *PREfast* with good results, too. However, neither the tools nor detailed reports about results seem to be available to the general public.

False Positives

Sophisticated static analysis techniques are capable of finding large amounts of errors in relatively short amounts of time. Static analysis is particularly efficient and successful in finding certain types of errors that are hard to find by means of traditional testing. As opposed to testing, static analysis usually discovers the root cause of a potential error and not just the symptoms.

However, the potential errors found by static analysis techniques can turn out to be false positives. I.e. an error is reported that cannot occur in practice. Often this is due to insufficient knowledge on the part of the static analysis tool. E.g. analysis may find a code path that would lead to a NULL pointer dereference but due to circumstances that are beyond the scope of the analysis techniques being used that code path is in fact impossible. Depending on the type of analysis that is performed, the number of false positives can comprise a substantial part of the error reports. This increases the likelihood that real errors are missed. As a consequence some static analysis tools also support the ranking of error reports by means of heuristics that prioritize error reports that are more likely to be actual errors[30].

False Negatives

Despite the fact that static analysis can find many potential errors that are hard to find with classical testing techniques, it still suffers from the same principal problem: Static analysis can find potential bugs but it cannot guarantee their absence. One of the most fundamental problems is pointer aliasing, i.e. the fact that two completely unrelated pointers can refer to the same object. In general it is very hard to decide if this can actually be the case.

Static analysis techniques are effective in finding bugs in both single-threaded and multi-threaded systems. However, existing techniques suffer from both, the possibility of false negatives and substantial amounts of false positives.

1.2.4 Partial Program Verification

Partial program verification[14] is a special form of static analysis that has been proposed as a way to prove the correctness of certain functional aspects of a program. Such a verification usually assumes that all other aspects of the program are correct or at least that bugs in other parts of the program do not affect the correctness of the verification. It is important to note that in this context the word “partial” refers to a functional part and not to a specific procedure or module.

An example of a property that is suitable for partial program verification is the use of file handles: Their use must always follow a known pattern: first a file is opened creating a valid file handle, then the file handle may be used several times, and finally the file handle is closed. Any use of a file handle that deviates from this access pattern is a bug. The idea of partial program verification is to verify that all uses of file handles follow this pattern while other potential problems and bugs are ignored.

A common approach to such a problem is to define the correct behavior, e.g. by means of a finite state machine and to show that there exists no code path that leaves the state machine in an error state. As opposed to other static analysis techniques that only try to find potential errors, a partial verification technique must make conservative assumptions about possible code paths. However, in order to decide if a certain code path is actually possible, it is often necessary to keep track of the contents of many global variables that are modified and evaluated along that code path. Otherwise it is likely that many code paths are flagged as potentially erroneous while they are in fact impossible. Having said that, keeping track of too many variables leads to an explosion of the number of possible states which makes the desired partial verification practically impossible.

In [14] heuristics that solve this problem for many practical cases are suggested and the authors are able to show that the GNU-C Compiler handles its many output files correctly according to the definition given above. However, these heuristics implicitly assume a single threaded program and it is not clear how they could be adapted to multi-threaded environments.

Seidl et. al.[46] have proposed methods based on abstract interpretation that can be used to perform partial program analysis of mutex synchronization in multi-threaded systems. However, these methods cannot handle mutexes in dynamically allocated data structures and in the examples the total number of mutexes in the system is very small (around 10).

1.2.5 Model Checking

In model checking[11] the desired properties of a system are described by a logical formula, commonly given in temporal logic[42]. It is then shown that the formal system is in fact a model of the given logical formula. The approach itself is very general but in the context of computer systems model checking has been very popular in the verification of hardware designs (see e.g. [35]).

The ideas used in model checking are to some extent similar to those used in the context of partial program verification. In fact the example of a specification for the correct use of file handles given in the previous section can be interpreted as a logical formula describing the correct behavior of (a special aspect of) the system. The process of verifying that a system adheres to that specification, shows that the system in question is a model of the formula used as the formal specification. The major difference to partial program verification is that the formal description used in model checking describes most or ideally all functional aspects of the system. This allows for improved algorithms to check if a given system is a model of the formal description. However, this also leads to more complex formal specifications and limits the applicability of model checking to systems where such a formal description can be given. Additionally, model checking also needs to deal with the state explosion problem that was already mentioned in conjunction with partial program verification.

Having said that, successful verification of a system with the help of model checking gives stronger and trustworthy results regarding the correctness of the system.

1.3 Outline of this Work

This work consists of two parts. In the first part (Chapters 2 to 5) we will develop a static analysis algorithm that yields a conservative model of the possible states that a program can be in. In the case of multi-threaded programs the result is still conservative but also relatively accurate. In the second part (Chapters 6 and 7) we will apply this algorithm to synchronization problems and present results that show that these methods can actually be applied to real life programs of significant size.

In the following we give a more detailed overview of the remainder of this work:

Program Representation Chapter 2 discusses how analyzed programs are represented. We also discuss in more detail other prerequisites that the program must have.

Basic Analysis Chapter 3 summarizes preparatory analysis techniques. Most of those techniques are also useful in the context of compilers. The results obtained by the analysis techniques described in this chapter will be helpful in subsequent chapters.

Function Body Analysis Chapter 4 discusses how a block of code, e.g. the body of a function, can be analyzed in a conservative way within a multi-threaded system. The result of the analysis is a relatively accurate conservative description of the possible code paths through the block.

Global Analysis Chapter 5 shows how the results obtained for blocks of code can be extended to a complete analysis of a self contained program.

Synchronization Chapter 6 gives an example of how the analysis results obtained in the previous chapters can be used to check that locks are used properly.

Case Study Chapter 7 shows by means of several examples that the techniques developed in this work can actually be applied to real life systems of significant size. Time measurements and examples of actual bugs that have been found are presented. We also give examples of program constructs that are not fully understood by our analysis.

Conclusions and Future Directions Finally Chapter 8 gives a summary and outlines areas where future improvements and extensions are possible.

Chapter 2

Program Representation

In this chapter, we describe requirements regarding the representation, syntax, and semantics of programs that can be analyzed with the methods developed in the subsequent chapters.

2.1 Abstract Syntax Trees

The source code of a program while easy to read is an inconvenient representation for the purpose of program analysis. Instead abstract syntax trees (AST) which allow simple access to a lot of semantic information are used. Assume for example that we are interested in the type of a variable that is used in an expression. With a pure source code representation, we would have to search the source code backwards for the declaration of a variable with that name, parse that declaration to find the name of the type, search for the declaration of that type, and parse that declaration. With an abstract syntax tree all of this information is readily available via pointers. Additionally all names of variables, functions, types etc. are replaced by pointers to their respective declaration node in the syntax tree. If necessary such a declaration node is implicitly created, e.g. for types that do not have a name or that are not explicitly declared.

Figure 2.1 shows a part of the abstract syntax tree of the following function

```
int add (int a, int b) { return a+b; }
```

that returns the sum of its two integer parameters a and b. Several things are noteworthy in this simple example:

- There is only one node for each object but this node can be referenced by multiple different nodes via pointers.
- The names of objects are only kept for debugging purposes and a lot of nodes do not have names in the first place. The semantics expressed by a tree do

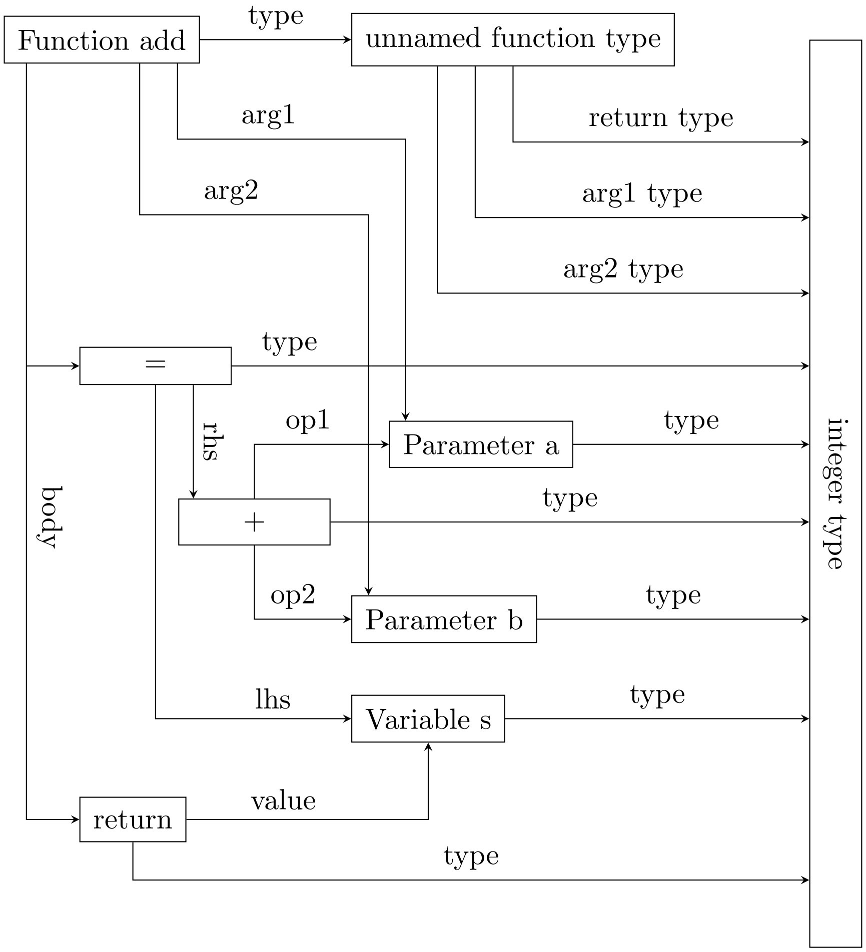

Figure 2.1: An Abstract Syntax Tree

not depend on object names. E.g. each use of a variable in an expression or elsewhere references the declaration node of the variable in the tree directly.

- Some objects are declared implicitly. Usually, such an object does not have a name. The type of the function *add* in Figure 2.1 is an example of such an object.
- The real AST contains even more pointers than those shown in Figure 2.1. E.g. a node for a variable contains a pointer to the scope of the variable which is omitted in Figure 2.1.

Despite its name, an abstract syntax tree is not really a tree in the graph theoretic sense. Additionally the tree contains semantic information, e.g. the type of each variable, expression, function, etc. Such a tree is sometimes called a *decorated abstract syntax tree*. We simply use the term *abstract syntax tree* or *AST* because we only use decorated trees and do not need to distinguish between pure and decorated trees.

```
unsigned gcd (unsigned a, unsigned b)
{
        while (a && b) {
                if (a > b) {
                        a %= b;
                } else {
                        b %= a;
                }
        }
        return a+b;
}
```

Figure 2.2: Source code for the AST in Figure 2.3

2.2 Normalized Syntax Trees

Generally there are many ways to express the same program by means of different ASTs. However, it is convenient to rewrite syntax trees into a normalized form[22][37] which only uses a restricted subset of all possible program constructs. This set is chosen in such a way that every program can be transformed into normalized form without changing its meaning. In this section we briefly describe all programming constructs that are allowed in the normalized form. By only allowing a limited number of these constructs, we will be able to describe analysis methods in a more compact form.

Before we continue with a more formal description of the program constructs it might be helpful to consider an example. The function *gcd* in Figure 2.2 calculates the greatest common divisor of its parameters *a* and *b*. Figure 2.3 shows a normalized abstract syntax tree for this function. For the sake of clarity, only some of the nodes in the AST are shown. A node for a variable or parameter in the AST is represented by multiple boxes to avoid intersecting arrows. To emphasize this, the respective boxes are surrounded by a dashed frame.

Looking at the tree itself we can see that the **while** loop has been replaced with an **if** statement and **goto**s. Several temporary variables were added for intermediate results of more complicated expressions. Only plain variables are used in the condition of an if statement. If necessary, the condition is calculated in advance and the result is stored in a temporary variable. In the following we describe the elements of a normalized syntax tree in more detail.

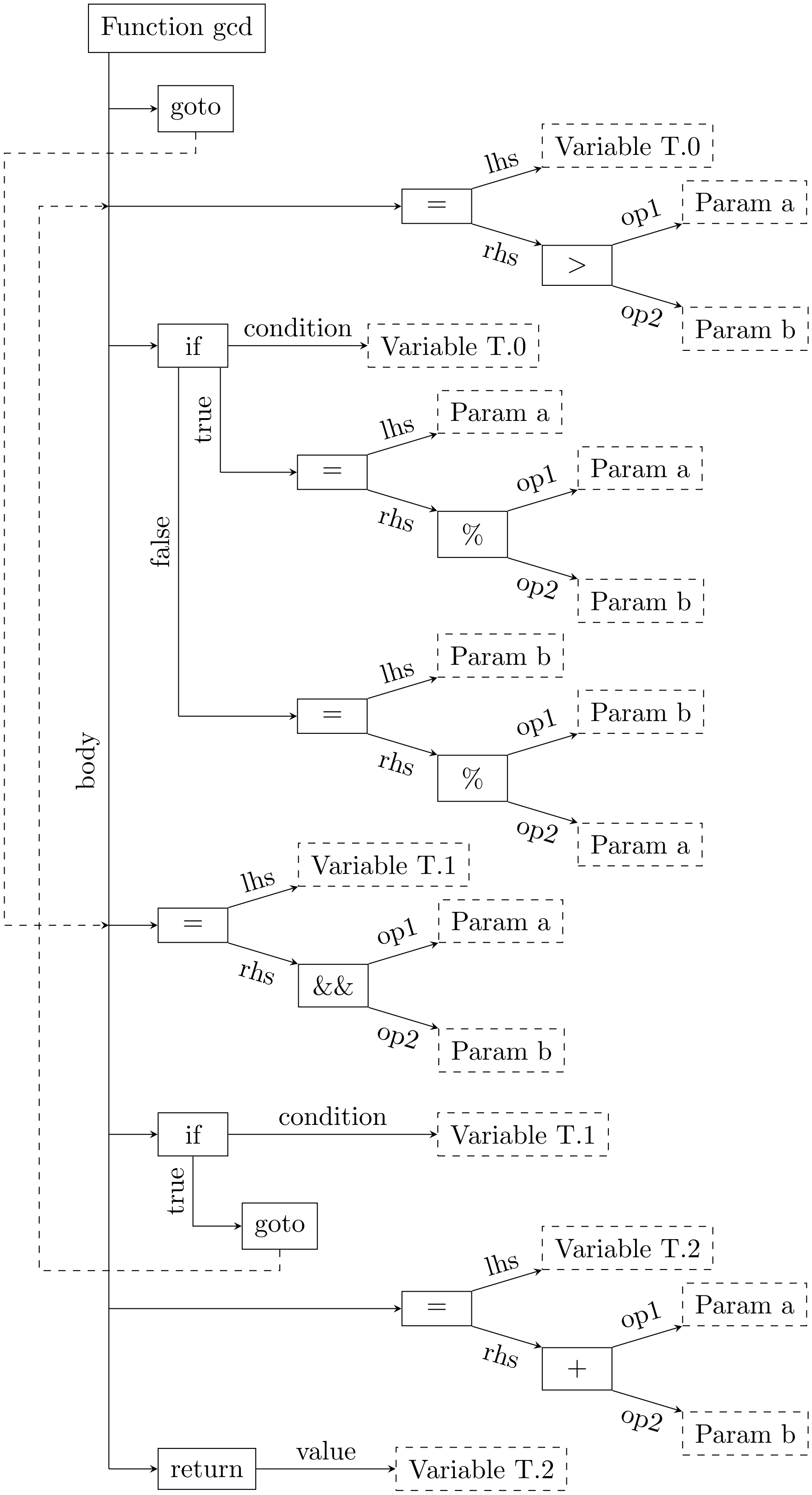

Figure 2.3: Simplified normalized AST for the code in Figure 2.2

2.2.1 Declarations

Each declaration describes a new program object (type, variable, function, etc.) which may or may not have a name. Every declaration specifies the source file and line number of the declaration. Whenever a node in an AST refers to a previously declared program object it references the declaration node in the AST directly (e.g. by means of a pointer), instead of the name of the declared object. The following paragraphs explain which elements must be contained in this description of an object being declared depending on the kind of that object.

Type Declarations

The description of a type may be a simple type (e.g. **int**, **char**, or **float**) another existing type, a pointer type, an array type, or a record type. A pointer type description solely consists of the type of the memory object that the pointer points to. An array type description consists of an integer constant which specifies the number of elements in the array as well as the type of a single element. Finally, a record type description is a list of record field declarations.

Record Field Declarations

A record field declaration always has a name that specifies the name of the record field. The description of the field is given by the type of the field and the type of the surrounding record. The latter restriction ensures that a single record field declaration cannot be used as a field in different unrelated records.

Variable Declarations

The declaration of a variable is described by the type of the variable and optionally by the scope that the declaration is contained in. The scope if present is the compound statement that the variable is declared in. A global variable does not have a scope. Instead global variables can also have an initial value.

Formal Parameter Declarations

A formal parameter of a function is a special case of a variable declaration. Its scope is always the function itself and it must always have a name. Formal parameter declarations can only appear in the argument list of a function declaration.

Function Declarations

The description of a function declaration includes a list of formal parameter declarations and the return type of the function. For non-prototype declarations the description also includes a compound statement that contains the body of the function. Nested functions are not allowed.

2.2.2 Expressions

Mathematical Expressions

All the usual arithmetic, logical, bitwise, and comparison operators can be used to build expressions with one or two operands depending on the operator. However, only variables, formal parameters, and constants are allowed as operands. Furthermore, the use of mathematical expressions themselves is limited to the right hand side of an assignment. Basically, this means that the results of these expressions are always stored in a possibly artificial variable before use.

Component Reference

Given a pointer to a variable of record type and a field declaration, a component reference yields the value of the specified field in that record.

Array Reference

Given a pointer to a variable of array type and an integer value, an array reference yields the value of the array element at the index specified by the integer.

Address-Of Expressions

The address-of operator can be applied to variables, formal parameters, function declarations, component references, and array references. It yields the address of the object given as its argument. In the latter two cases this is the address of the record field or array element. Note, that it is not possible to apply the address-of operator to a label.

Dereference Expressions

A dereference operation can only be applied to a pointer pointing to a scalar type. Dereferencing a pointer to a record or array type is not allowed. Individual record fields or array elements can be accessed via component or array references. However, the array or record as a whole can never be the result of an expression.

2.2.3 Simple Statements

Labels

Labels themselves are statements without effect. They are embedded in a list of statements to mark a place where control can be transfered to. The scope of a label is the surrounding procedure, i.e. control can only be transfered to a label from within the same procedure.

Goto Statements

A goto statement specifies the label where control should be transfered to. Note that the argument of a goto statement is always a label given by its declaration. It is not possible to specify a calculated address or the contents of a variable as the target of a jump.

Assignments

If the left hand side of an assignment is a variable or a formal parameter, the right hand side may be any expression or a call to a function that has a return value. The left hand side of an assignment may also be a dereference, an array reference, or a component reference. In this case the right hand side can only be a variable. I.e. at least one side of an assignment is always a variable or a formal function parameter. Arrays or records as a whole cannot be assigned to each other.

2.2.4 Return Statement

The return statement terminates the execution of the current function and returns control to the calling function. If the function returns a value, a local variable containing the return value should be specified. Conceptually, returning a value from a function is equivalent to an assignment to a variable in the calling procedure.

2.2.5 Function Calls

The function that should be called is either specified explicitly by a function declaration or implicitly by a variable that contains a pointer to the function. The actual parameters are given as a list of variables. No expressions or constants are allowed. A function with a return value is normally used on the right hand side of an assignment. However, a function call can be used in statement context if the function does not return a value or if the return value can be ignored.

2.2.6 Aggregate Statements

Compound Statement

A compound statement is used to build a block of one or more statements. It may also introduce one or more additional variable declarations that are only valid within the scope of this compound statement.

Conditional Statements

A conditional statement (also known as an if statement) consists of a variable and two compound statements. If the value of the variable is true, the first compound statement is executed, otherwise the second compound statement is executed. The statement list in each of these compound statements is allowed to be empty. The actual branch condition must be calculated in advance and sophisticated expressions that might require boolean short circuiting must be rewritten using consecutive conditional statements. Note that the last action in both compound statements is implicitly a control transfer to the statement immediately following the if statement.

Switch Statement

While a **switch** statement can in theory be rewritten by means of nested if statements, this can lead to a rather complicated nested block structure which may make program analysis unnecessarily difficult. Thus, we allow **switch** statements in a normalized source tree. The switch condition must be a variable of integral type. The body is a *single* compound statement interspersed with label statements. Control is transferred to one of these labels dependent on the value of the switch condition. This results in the fall through semantics known from C, a **break** statement can be rewritten as a jump to the statement that immediately follows the **switch** statement. Every **switch** statement must have a default branch that is taken if the condition variable has a value that does not appear explicitly in conjunction with a case label.

2.3 Single Static Assignment Form

Single static assignment form, often also referred to as SSA form, takes normalization of abstract syntax trees one step further. In SSA form each variable appears exactly once on the left hand side of an assignment. If the original program contains multiple assignments to a variable, each one assigns to a different version of that variable. Typically, versions are distinguished by a serial number that is appended to the variable name. At each use of the variable we need a way to decide which version should be used. This problem is solved by adding so called Φ-nodes and even more versions of the variable. If it is possible at some place in the program that two versions $x.i$ and $x.j$ of a variable x are valid, we generate a new version $x.n$ and add an Assignment $x.n = \Phi(x.i, x.j)$. I.e. the Φ-function is a reminder that at this point of the program two different versions of the variable x are joined. Consider the example in Figure 2.4: We can see that each assignment to x generates its own version.

Original Program

```
if (a) {
        x = 3;
} else {
        x = 4;
}
print (x);
```

SSA Form

```
if (a) {
        x.1 = 3;
} else {
        x.2 = 4;
}
x.3 = Φ(x.1, x.2);
print (x.3);
```

Figure 2.4: Transformation into SSA Form

After the if statement there are two versions of x, namely $x.1$ and $x.2$, which might be valid depending on the branch that is taken. This is expressed by the new version $x.3$. The arguments of the Φ-function list the versions of x that flow together at this point. It is important to note that SSA form is in fact a property of a single variable. It is possible to rewrite a program such that only a certain subset of variables has the SSA property. For example it is common to bring only variables declared locally to a function into SSA form.

If we want to transform a program given in SSA form back into a normalized parse tree, we have to replace each Φ-Function with normal source code. In the example in Figure 2.4 this can be done by adding the assignments $x.3 = x.1$ and $x.3 = x.2$ to the end of the true and the false branch of the if statement, respectively. For a detailed description of SSA form see for example[3].

There are efficient algorithms to convert programs to SSA form and back[13]. Hence, we will assume that our program is given by a normalized parse tree as defined in the previous section or as a normalized parse tree transformed into SSA form depending on what is more convenient for the task at hand.

2.4 Linking

Up to this point we silently assumed that each program object such as a type or a variable corresponds to a single declaration in the parse tree. Generally, this is true for a single translation unit and its resulting parse tree. However, a program usually consists of many source files that are compiled separately. The results of these compilations are then combined into the final executable by a process called linking. Basically, linking ensures that different references to the same program object actually refer to one and only one instance of the object even if its declaration has been imported into multiple translation units. It is convenient to make the following distinction that is motivated by the C programming language:

Source File refers to a single file that is used during compilation. In the context of the C programming language this usually means a `.c` or a `.h` file.

Translation Unit refers to all the source code that the compiler considers while translating a single file. This includes definitions of types, function prototypes, etc. in interface definitions that might be stored in different source files.

While we normally deal with one parse tree at a time which is derived from a single translation unit we have to keep in mind that an object mentioned in one source file or translation unit can be used in other translation units as well. For example a global variable may be accessed from different translation units and this might influence the results of our analysis.

As we do not actually create a final executable, our linking process, i.e. the way linking is done, slightly differs from that of a normal linker: We generate a global list of all program objects that are used in more than one translation unit and assign a unique ID to each of these objects. Objects that can be used in multiple translation units are external variables and functions, type declarations, and record field declarations.

2.4.1 External Variables and Functions

Linking of functions and variables is relatively simple: There can only be one global exported variable or function with a given name and all exported declarations of a function or variable with that name refer to the same object. Other non-exported or local declarations do not need any linking at all. Thus we associate all global exported declarations for a given name with each other. As stated above, this means that each linked function or variable is given its own globally unique ID. While doing this we also check that each function is implemented exactly once throughout the whole program, i.e. there is exactly one non-prototype function declaration for that function name. Likewise there can be at most one exported variable declaration with an initialization for each exported variable name.

2.4.2 Type Declarations

With field and type declarations the situation is a bit more difficult because the type information is normally lost in the translation process. Consequently an ordinary linker has no need to care about type information and thus there is no general rule that allows us to decide which type declarations with a given name refer to the same type. The situation is complicated even more by the possibility of incomplete or unnamed types. Incomplete types may have a declaration and it is possible to declare and use pointers to that type while the layout of the type need not be available in the translation unit at all. Furthermore it is possible that two types in different translation units have the same name but different and possibly unrelated definitions. Thus it is in general not possible to link type declarations reliably.

There are several possible solutions to this problem: Some programming languages (e.g. Oberon[51]) allow stricter type checking across translation unit boundaries. To some extend it is also possible to infer or even define type equality by their use[47]. We take a simple heuristic approach that works well for reasonably written programs. Two type declarations in different translation units are considered to be equal if the following conditions are met:

- The type has global scope. Type declarations with local scope cannot be used outside of their scope anyway and a local scope cannot cross translation unit boundaries.
- Both type declarations declare a type with the same name.
- Both declarations have the same source location, i.e. they come from the same source file albeit being declared in different translation units. This basically means that there is one and only one physical definition in some header file or definition module and this definition is imported by each translation unit that wants to use the type.

These rules work reasonably well and the possibility that two type declarations are linked while they in fact declare different types can safely be ignored. However, there is a reasonable chance that two type declarations in different translation units refer to the same type but this fact is not detected by the heuristic. A function prototype that returns a pointer to some record like this

```
struct page * newpage();
```

is an example where this can happen accidentally in C. This piece of code will compile just fine even if there is no previous declaration of `struct page` provided that no pointer to `struct page` is dereferenced in this translation unit. However, in this example `struct page` is implicitly a forward declaration of a type defined elsewhere, probably in a header file that should be included. If the full definition of `struct page` is not available in this translation unit, the heuristic given above will not link the real declaration to the implicit forward declaration.

Consequently we have to be careful not to rely on linking of type declarations. If really necessary, it is still reasonable to add linking of types, that works correctly with the heuristic above, to the prerequisites for an analysis. However, most of the time it is sufficient if we restrict this prerequisite to the linking of field declarations as described in the following section.

2.4.3 Structure Field Declarations

As we have seen, linking of type declarations is error prone and does not work well in general. However, we can use almost the same rules for the linking of field declarations and we will see that we can expect this to work reliably for reasonably well written programs. Two field declarations are linked if

- both field names are identical and
- both field declarations have the same source location.

First of all note that a field declaration in a normalized AST must have a name. Again it is difficult to write a program where two linked field declarations do not refer to the same record field and the probability that this happens accidentally can be safely ignored. Conversely, having two field declarations that should be linked but are not linked if we use this rule would require two field lists at different source locations for the surrounding record type within a single program. This situation is very rare in practice and would be considered an error by most people. We only consider programs where linking of field declarations as described above works reliably.

2.5 Semantic Restrictions

In this section we discuss semantic restrictions for the programs we analyze. Unless otherwise stated a violation of these restrictions leads to a program that is not well formed for the purpose of our analysis, i.e. a violation of these restrictions invalidates the analysis.

2.5.1 Complete Normalized ASTs

One prerequisite is that our program is (or can be) represented as a set of normalized abstract syntax trees as described in Section 2.2. All ASTs that comprise the program must be available and linking must be possible without any conflicts. This means that the program cannot use external libraries that are not available for analysis. Furthermore there must not remain any references to undefined functions or variables after linking. The fact that a program can be represented in the form of an abstract syntax tree also means that the program is at least syntactically well formed according to the specification of the programming language it is written in.

Most of this is a pretty obvious. However, the fact that no external libraries are allowed is a relatively strong requirement. To mitigate this, it is generally possible to specify the effect of external functions by hand for the purpose of our analysis. The analysis algorithms will blindly trust such a description provided by the user. Naturally, this can influence the correctness of the analysis and it is up to the user to assess the consequences of possible errors in user provided descriptions.

2.5.2 Addresses of Objects

We assume that there are only two ways to get the address of a memory object: The first possibility is that the address is explicitly calculated by the use of the address-of operator. For a variable this is the only way to obtain its address. The second possibility to obtain an address is to allocate the object from the global heap by calling a special allocation function. In this case, the allocation function returns the address of the newly allocated object. Note that once the address of an object is known, the pointer value can be passed around arbitrarily.

We also assume that it is not possible to calculate the offset of a record field inside its surrounding record type, i.e. an individual record field can only be addressed if there exists at least one address-of operation whose argument is a component reference using that specific record field declaration. This restriction on addresses is a prerequisite.

We say that a variable or a record field is *unaddressed* if there exists no address-of operator that is applied to that variable or record field (see Section 3.2).

2.5.3 Effective Type and Aliasing

The effective type of a variable or formal parameter is its declared type. The effective type of a heap object is the type that was used for the last write operation to that object. We require that each memory object must have a uniquely defined effective type at any point in the program. Read accesses to a memory object with a certain effective type must not be done through a pointer to another (incompatible) type. An informal description of this requirement is that a read access to an object via a certain type can only read a value that was previously stored into the same object via the same type. As a corollary the value of an object cannot change between two reads without a write operation to that object with a type that is compatible with the second read.

One of the immediate consequences of this restriction is that memory functions that operate on blocks of memory (such as `memcpy` or `memmove` in C) cannot be used.

However, if these functions are used to copy aggregate objects of compatible type, this can be rewritten as a sequence of assignment operations for the elements of the aggregate (recursively if necessary). The other common block operations can similarly be rewritten as a sequence of basic operations without changing semantics. Other uses of these functions are rare and if they occur the user must manually rewrite them to clarify the semantics of the copying operation.

Many programming languages already require that each memory object has a single effective type for its entire life time. However, the strictness of the type rules and their checking differs. E.g. in Oberon, all memory objects have a well defined effective type and violations of the type rules lead to compile or runtime errors[51]. The C programming language[25] also imposes type rules and aliasing restrictions on a conforming program[1] that are much stricter than most C programmers are aware of. However, the language specification does not require that these restriction are checked during runtime. Instead it is assumed that the programmer obeys the rules and violations lead to undefined behavior. Consequently, an optimizing compiler is free to make optimizations based on the aliasing rules and programs that do not adhere to them are likely to break silently right away, with some future compiler version, or due to an apparently unrelated change elsewhere that triggers more optimizations. As we will primarily focus on programs written in the C language we have a closer look at the aliasing restrictions imposed by this language and their relation to our type rules as specified above. Consider the example in Figure 2.5. With most recent

```
typedef struct { int i; } integer;
typedef struct { short a,b; } short2;
void f (integer * i, short2 * s) {
        s->a = 3;
        s->b = 4;
        i->i = 5;
        printf ("%d␣%d␣%d", s->a, s->b, i->i);
}
```

Figure 2.5: C99 Aliasing violation

C compilers[2] an optimized version of this function will output `3 4 5` even if `i` and `s` point to the same location. This is because the compiler may rightfully assume that a pointer to an `integer` object cannot point to an object with effective type `short2`. Thus the actual semantics of this function are not uniquely defined by the source code alone. As stated above we will not consider such programs later on.

---

[1] See [25], sections 6.5(6) and 6.5(7)

[2] E.g. **Sun C 5.8 2005/10/13** (`cc -xO4 -xalias_level=std`) and **gcc version 3.4.3** (`gcc -O2`).

Unfortunately, the C99 Standard[25] allows an exception to these type rules: A character pointer is allowed to alias anything. One of the main reasons for this exception are the block memory functions that can be rewritten as described above. We will still assume that writes through a character pointer cannot change the contents of objects with another effective type. This means that places in a C program where pointers of another type are cast to character pointers need careful manual checking. However, the risk of actual errors caused by violations of this assumption is very low in the analysis of reasonably well written programs.

It is important to note that the considerations in this section do not necessarily require accurate inter module linking of types as discussed in Section 2.4.2.

2.5.4 Field Declarations and Unions

We assume that linking of field declarations works reliably as previously described in Section 2.4.3. Unions are not allowed in the normalized abstract syntax trees as defined in Section 2.2, i.e. each field declaration is part of exactly one record type. If we want to analyze programs containing unions, these can be treated like records. This will lead to additional possibilities for aliasing among the fields of a union that are not taken into account by the analysis but the overall results will still be mostly accurate.

The reliable linking of field declarations together with the program wide rules for effective types give us a very nice property of record fields at least as long as they are unaddressed within the whole program: The value written to a record component via a component reference can only be changed by a store via a component reference using the same field declaration. Additionally, the base vectors used by the two component references must be equal.

2.5.5 Integer Model

We assume that all integer and pointer types follow a simplified integer model in which each integer variable can have a value that reaches from $-\infty$ to $+\infty$. Even in this model, no variable can actually have an infinite value. However, if we describe the range of values that a variable can have, we will use ranges that include $-\infty$ or $+\infty$ whenever we do not know a lower or upper bound, respectively. E.g. if we state that the value of a variable is in the range $[-\infty, -1]$, this means that the variable can have any negative value that can be represented in the range of values permitted by the type of the variable. Should the variable have an unsigned type, no values would actually be possible. It is not a priori clear that arithmetic operations always yield the same results in our integer model as the respective operations on real integer types. Special care must be taken in the following two cases:

An overflow occurs in an arithmetic operation: We could in theory rewrite all critical arithmetic operations to get around this problem. In this case the result of a critical operation would be adjusted to the possible range of values by a bit-wise "and" or a "modulo" operation. However, we will not follow this approach. Instead we either consider the possibility of an overflow separately for each operation or simply treat the result as unknown. While treating the result of arithmetic operations as undefined may seem to be too imprecise, we shall see that this approach in fact has a lot of nice side effects.

An integer type conversion occurs: The strategies for dealing with this case are basically the same as with arithmetic operations. However, for conversion operations it is often easier to prove that the conversion is safe.

Having said that, many interesting operations can safely be performed in our simplified integer model:

- Assignments where both sides have the same type
- Assignments where the type of the left hand side is a superset of the type of the right hand side
- Comparisons of two variables of the same type
- Comparisons of an integer variable with an integer constant.
- Passing a variable as an actual parameter in a function call to a formal parameter of the same type. The same applies to the return value of a function.

All of these operations when performed in our integer model will yield the same results as the real program operation. In a few cases there is a possibility of imprecision, i.e. our simplified model may allow additional values that are impossible in reality.

Floating point arithmetic is allowed, but the analysis will not track the value of floating point variables beyond equality and inequality.

2.5.6 Multi-Threading

All programs that we consider are assumed to be multi-threaded, i.e. there are multiple flows of control. All of these control flows are assumed to share the same global variables and a single global heap. Furthermore, all control flows execute the same code base and all possible entry points are known. This means that each thread has a top level function that it executes and the thread dies once this function would return. Termination of a thread can also be caused by an assertion failure, an external event, or an explicit call to the function `exit`. In its life time a thread can call arbitrary other functions and it can be put to sleep by voluntary actions or by external events. A thread that should never terminate can run an infinite loop in its main function.

As indicated previously, the methods developed in this work can also by applied to single threaded systems. However, they are optimized for the multi-threaded case and the results obtained for a single threaded program will be disappointing.

Thread and Processor Local Data

Any thread can have its own blocks of thread local data, i.e. data that can only be accessed by the thread itself. One obvious example of such a local data block of a thread is its stack. The stack need not be the only thread local block of data though. Additionally, there may also be blocks of data that are local to each processor. Examples of data that is typically local to each processor are performance counters, interrupt data structures, and scheduling information.

We will assume that thread and processor local data is physically protected from other threads and processors respectively. In reality this is often achieved by a combination of checks done by the compiler and discipline exercised by the programmer.

2.5.7 Volatility of Global Data

One important consequence of the previous section on multi-threading is that global data (all global variables and the heap) is volatile. In other words, global data can be changed by another control flow at any point in time unless special additional care is taken. Depending on the processor architecture, reading the value of an integer variable at the same time as a write to that variable can return a partially written value, i.e. neither the old nor the new value of that variable.

This means that it is neither necessary nor useful to keep track of arbitrary global data during program analysis. Let us consider the code fragment in Figure 2.6 derived from the random number generator of the Linux kernel version 2.6.6. At first sight this appears to be reasonable code: `random_state` is a pointer to a record that

```
struct entropy_store *random_state;
/* ... */
int random_get (char * buf, int size) {
        spin_lock (&random_state->lock, flags);
        /* Copy size bytes from random_state->pool to buf */
        spin_unlock (&random_state->lock, flags);
}
```

Figure 2.6: Volatility of global data

includes a mutex lock and random data that is protected by that lock. The real code even ensures that interrupts are disabled during access to the random data in `random_state->pool`. However, the problem is that there is no protection for the global pointer `random_state` itself, i.e. it is entirely unclear whether the call to `spin_lock`, the call to `spin_unlock`, and the intermediate copying operation all use the same pointer value from `random_state`. In fact there used to be a feature[3] that allowed the super user to change the size of the pool of random data. This was done by allocating a totally new record and the value of `random_state` was replaced with a pointer to that newly allocated record. This assignment could happen between the lock and the unlock operation in the code fragment in Figure 2.6. The result is unpredictable behavior reaching from silent data corruption and unexpected segmentation faults to complete hangs of the whole system.

Multi-Threading and Effective Types

The rules for aliasing and effective types as defined in Section 2.5.3 assume that there is only a single control flow that can access memory. We will only consider programs where the programmer obeys the rules defined for effective types even across multiple control flows. This means that any store to a memory location at any time defines the effective type of the memory location for all control flows. Reading a memory location is only allowed via the current effective type of that memory location.

Memory Allocation Functions

The consideration about volatility of global data does not apply to memory returned from a memory allocation function. The return value of a memory allocation function must always point to unaliased memory on the heap, i.e. the pointer returned from the allocation function is the only way to access that object. It is possible to specify a list of memory allocation functions. Analysis will trust this list and assume that these functions actually return unaliased memory at all times.

2.6 Summary

We will only consider programs given in a certain form for analysis. The properties of the programs we consider are:

Completeness The complete program is available in the form of an abstract syntax tree.

---

[3] Removed in more recent kernel versions

Object Addresses The address of an object can only be obtained via an address-of operator or via a pointer to the object that is returned by a memory allocation function.

Multi-Threading We will assume multi-threaded programs. Unless other guarantees are made, all global data is subject to change at any time.

Strict Effective Types and Aliasing rules Rules for effective types and aliasing must hold across control flows. The most important consequence is that a value stored into an unaddressed record field can only be changed if some control flow executes an assignment to the same record field.

It is always possible to describe the semantics of some program construct manually, e.g. if it would otherwise violate the semantic restrictions given in this chapter or if the source code of a function is not available. But then, violations of these program restrictions will not render our analysis completely useless. They do leave additional room for errors and imprecision though.

Chapter 3

Basic Analysis

In this chapter we have a closer look at several basic analysis techniques. These techniques are mostly well known or more or less straightforward. They are detailed here because their results will be needed later.

From now on analysis is conservative, i.e. we do not “guess” but only draw conclusions that are known to be true based on assumptions detailed in the previous chapter.

3.1 AST Generation and Object Names

The first analysis step is the compilation of each translation unit into a normalized abstract syntax tree (see Section 2.2). A modified version of gcc can be used to generate a text representation of this AST. In this process each translation unit gets a unique translation unit ID (TID). Additionally, each node of the abstract syntax tree produced from a translation unit is assigned a node ID (NID) that is unique within the translation unit. This way each AST node in the whole program can be identified uniquely by its TID and NID.

Beyond that, some nodes (e.g. global variables and exported functions) get a global link ID (LID) according to the rules described in Section 2.4. A program object that does not have a LID is only referenced in a single translation unit, i.e. each program object can be uniquely identified by its LID if it has one or otherwise by its TID/NID pair. However, it is important to keep in mind that different TID/NID pairs can refer to the same program object. In this case the TID/NID-pair must be translated into a globally valid LID whenever we want to refer to that program object outside of the current translation unit. Translating a LID back to a local NID of a given translation unit is only possible if that translation unit actually references the program object associated with the LID.

3.2 Addressed Variables and Fields

An interesting property of any object is whether there might be a pointer to that object. If this is the case, any pointer dereference even in an other module or thread might affect the value stored in that object provided that the effective type rules (see Section 2.5.3) do not say otherwise.

The restrictions given in Section 2.5.2 leave only limited ways to obtain the address of an object. For variables and formal parameters the only way to obtain their address is by means of the address-of operator. If this does not happen for a certain variable within the entire program, we can safely assume that there is no pointer pointing to that variable. We will refer to such a variable as an *unaddressed variable*. Similarly, if a field declaration is never used as part of the argument of an address-of operator, we will refer to this field as *unaddressed*.

Variables that are both local to a single control flow (all local variables of a function and thread local data) and unaddressed can only be accessed by an explicit reference to the variable in the same thread. For function local unaddressed variables, the access is limited to the current instance of the function. Global unaddressed variables can only be accessed via an explicit reference to their name. As opposed to local variables this can happen from any thread. Similar considerations apply to field declarations: An unaddressed record field can only be accessed via a component reference with its field declaration. As with global variables, this is possible from all control flows.

We iterate through the ASTs of all translation units and inspect all address-of operations in order to determine whether an object is addressed. If the argument of the address-of operator is a variable declaration, a parameter declaration, or a component reference, that declaration (the field declaration is used in case of a component reference) is marked as addressed in this translation unit. If the declaration is referenced in more than one translation unit, it has a global LID, which must also be marked as addressed. Finally, we have to go through all translation units a second time and mark all declarations as addressed that are linked to a global LID which is addressed. A declaration that is not marked as addressed in this process is unaddressed.

3.3 Additional Return Values

Some functions can conceptually return more than one value. In this case, the address of the variable that should receive the additional return value is passed as a parameter. A lot of these functions can easily be detected. For this purpose we iterate through all function definitions and find parameters of pointer type. If such a parameter is only used in an indirect reference on the left hand side of an assignment, we have identified a function parameter that can be used to return an additional value from the function. If we find such a parameter, we keep track of its index in the parameter list and the function it is a parameter of. This information is useful in two ways:

- If the address of a variable v is passed to one of these parameters of a function f we fully understand the way in which the address is used. It can only be used to clobber[1] the variable v during this specific call of the function f. Consequently, we can still treat the variable as unaddressed. However, special care must be taken to account for the effect of the function f on the variable, e.g. by treating the function call as an assignment to the variable with an unknown right hand side.
- Subsequent analysis can sometimes treat the function as if it would return several values at once. One of these values is assigned to v. An assignment to v via the pointer passed as a function parameter specifies the secondary return value received by v without actually returning from the function. In case of multiple statements that specify a particular secondary return value, the last write wins.

Special care with regard to aliasing must be taken if we want to make use of the information about secondary return values in the analysis of a function. During analysis of a function itself, we can always treat the memory locations pointed to by the pointers in the parameter list identified above as local variables of the current function. This is possible because the only operations performed on these variables are assignments and the value is not read inside the function itself. However, the information about the value stored in these memory locations is only accurate if the respective pointer is the only way to access the memory location from within the function. The calling function must ensure that this is actually the case before relying on this information. E.g. if the calling function passes the address of a global variable to the function with additional return values, the called function can access the global variable directly and via the pointer that is passed as a parameter, i.e. the pointer is *not* the only way to access the respective memory location from the called function. Thus, the calling function must not make use of information about this additional return value.

Fortunately, using information about the respective additional return value in the calling function is safe if the address of an otherwise unaddressed local variable is passed and the same variable is not used simultaneously for another additional return value in the same function call. This usage is safe because there are exactly two ways to access the memory location of the local variable during the function call: A direct access via the name of the variable which is only possible in the calling function and an indirect access via the pointer which is only possible in the called function.

---

[1] modify in an unspecified way

3.4 Basic Blocks and the Control Flow Graph

Informally, a basic block is a sequence of statements in a function that are always executed in the given order. For the purpose of this section, execution of a conditional statement or a switch statement consists of the evaluation of the condition and a control transfer as appropriate. Execution of the statements in each branch is not considered to be part of the execution of a conditional or a switch statement. Similarly execution of a compound statement only consists of an implicit control transfer to the first statement inside the compound statement. With this notion of execution there is always exactly one statement that is being executed. The following definitions define a basic block more formally:

Direct Domination Given two different statements s_1 and s_2 we say that s_1 *directly dominates* s_2 if s_1 is always executed immediately before s_2 and s_2 is always executed immediately after s_1.

Basic Block A sequence of statements $s_0, s_1, \ldots, s_n$ in a function forms a *basic block* if for any $k = 1, 2, \ldots, n$ the statement s_{k-1} directly dominates s_k.

Maximal Basic Block A *maximal basic block* is a basic block that cannot be extended by additional statements without losing the basic block property.

Each function body can be decomposed into disjoint maximal basic blocks. At the end of a basic block there is either a return statement or control is transferred to the beginning of another basic block. This may be a fall through, i.e. an implicit transfer to the next statement in source code order, a control transfer via goto, or a branch as part of a conditional or switch statement. In the latter case there are obviously several basic blocks where control can potentially be transfered to. If we add a directed edge from a basic block A to another basic block B if control can be transfered to the start of basic block B immediately after the execution of A, the result is a so-called *control flow graph*[2] or *CFG* of the function. Note that A and B might be equal, i.e. a CFG can contain loops. Additionally, it is often useful to add two artificial basic blocks ENTRY and EXIT to the control flow graph. ENTRY has a single outgoing edge to the first basic block of the function and EXIT has incoming edges from all basic blocks that end in a return statement. The example given in Figure 3.1 shows a function and its associated CFG. Each node of the graph in that figure represents a maximal basic block.

In order to calculate the maximal basic blocks of a function and its CFG, we first determine for each statement all possible statements where control may be transferred to. We call such a statement in the control flow a *next statement*. Note, that this may be different from the *successor* of a statement within a statement list in an AST or in the text representation of a function.

```
int f (int x,int y)
{
    int t;
loop:
    if (x < y) {
        t = x;
        x = y;
        y = t;
    } else {
        t = x - y;
        x = t;
    }
    if (x != y)
        goto loop;
    return x;
}
```

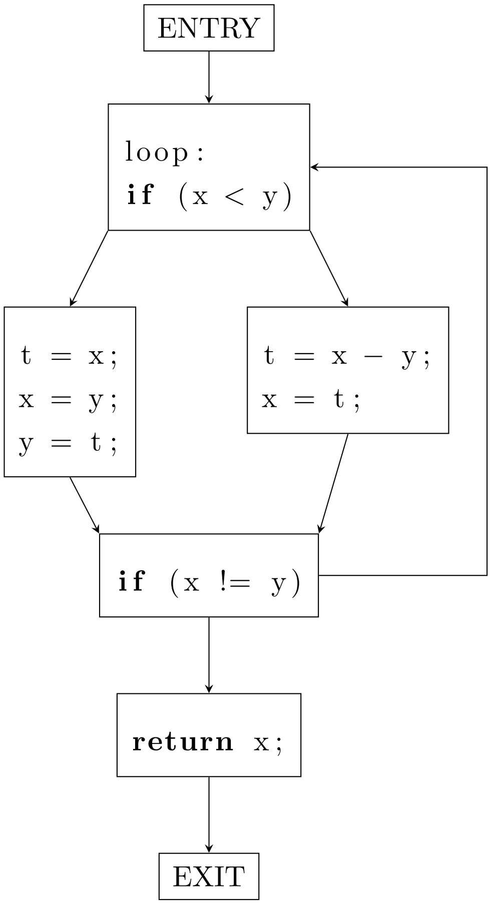

Figure 3.1: A Function and its Control Flow Graph

Once we know all possible next statements for each statement, it is easy to calculate all possible previous statements for each statement. At this point, we can recursively eliminate unreachable statements, i.e. statements that are not the ENTRY block and do not have any previous statement. Now we can determine a decomposition of the function into maximal basic blocks: First, we treat each statement as a single basic block. The next statements of a statement define the directed edges of this preliminary control flow graph. The resulting graph has all properties of the CFG except that the basic blocks used as its nodes are not necessarily maximal. Then, as long as there are two distinct basic blocks A and B (both different from the ENTRY and EXIT block), such that there is an edge from A to B which is the only outgoing edge of A and the only incoming edge of B these blocks can be combined into one. This is done by concatenating the statements in A and B. The edge between A and B is removed. Of course the order of the statements in the resulting basic block is preserved. Thus, we do not lose any information in the process of combining basic blocks. This step is repeated until no more basic blocks can be combined. The result is the final CFG having nodes that are maximal basic blocks.

More information on control flow graphs and their calculation can be found in most textbooks on compiler design such as [1].

3.5 Live Ranges

Informally, the *live range* of a variable is that part of a program where the value of the variable must be preserved. The control flow information gathered in the previous section will turn out to be useful in the calculation of live ranges. Live range detection has important applications in the context of register allocation and thus has been studied in detail (see e.g. [8] and [6]). In the following we only consider live ranges of unaddressed local (non static) variables. The live range of global variables is assumed to be the whole program and the live range of local addressed variables is the surrounding compound statement. For a local unaddressed variable, we first determine all statements where the variable's value is changed (definitions) and all statements that contain a read access to the variable (uses). It is possible that a single statement acts as both a definition and a use for the same variable. A compound statement is treated as a definition for those variables that are declared locally to it. Once we know all definitions and uses of a variable, we ask for each statement whether the variable is live, i.e. if we need to preserve its value, immediately before and immediately after the statement.

More formally, we can say that a variable is live if there is at least one code path starting at the current point in the program that uses the value currently stored in the variable. An assignment to a variable obviously erases the value currently stored in it. Thus, we only have to consider each code path up to the next statement that assigns a value to the variable. Furthermore, all code paths end if the current function returns because we only consider live ranges of local variables. This leads to the following observations:

- A variable is live after a statement s if the variable is live before at least one next statement of s in the control flow graph.
- No variable is live immediately before the EXIT block in the control flow graph.
- The following rules can be used to determine if a variable is live before a statement s other than the EXIT block:
    - The variable is live before s if the statement is a use of the variable.
    - Otherwise if s is a definition (but not a use) of the variable the variable is *not* live before s because its value is about to be changed.
    - Finally, if the variable is neither defined nor used in s it is live before s if it has been found to be live after s.

These conditions are sufficient to determine the set of variables that are live before and after each statement. This is done bottom up, i.e. we start at the EXIT block and proceed to the ENTRY block using the rules given above. For a given statement s we use the following definitions in the description of the algorithm:

- $\mathit{CFG\_next}(s)$ is the set of next statements of s in the control flow graph.
- $\mathit{live\_before}(s)$ is the set of variables that are live immediately before s.
- $\mathit{live\_after}(s)$ is the set of variables that are live immediately after s.
- $\mathit{defs}(s)$ is the set of variables that are defined in s.
- $\mathit{uses}(s)$ is the set of variables that are used in s.

Using these definitions we get the following data flow equations for all statements s:

$$\begin{aligned} \mathit{live\_after}(s) &= \bigcup_{s_{next} \in \mathit{CFG\_next}(s)} \mathit{live\_before}(s_{next}) \\ \mathit{live\_before}(s) &= (\mathit{live\_after}(s) \setminus \mathit{defs}(s)) \cup \mathit{uses}(s) \\ \mathit{live\_before}(\mathit{EXIT}) &= \emptyset \end{aligned}$$

With these equations we can use the algorithm given in Figure 3.2 in order to calculate $\mathit{live\_before}(s)$ and $\mathit{live\_after}(s)$ for all statements s of a function. We say that a variable *dies* at a statement s if it is in $\mathit{live\_before}(s)$ but not in $\mathit{live\_after}(s)$. The value of a dying variable can safely be ignored in any further analysis.

```
S := Set of all Statements except ENTRY and EXIT
$\mathit{live\_before}(\mathit{EXIT})$ := $\emptyset$
foreach $s \in S$ {
        $\mathit{live\_before}(s)$ := $\emptyset$
        $\mathit{live\_after}(s)$ := $\emptyset$
}
while anything changes {
    foreach $s \in S$ {
        foreach $n \in \mathit{CFG\_next}(s)$ {
            $\mathit{live\_after}(s)$ := $\mathit{live\_after}(s) \cup \mathit{live\_before}(n)$
        }
        $\mathit{live\_before}(s)$ := $(\mathit{live\_after}(s) \setminus \mathit{defs}(s)) \cup \mathit{uses}(s)$
    }
}
```

Figure 3.2: Live Range Calculation

3.6 Function Pointers

A calculated call is a call expression where the function being called is not given explicitly by name. Instead the function address is given by a pointer variable with unknown contents. One of the very first steps towards inter procedural analysis is the resolution of calculated calls. For each call that is not to an explicitly named function we need to find the set of functions that can actually be called at that point in the code. Analysis will then assume that any of these functions can actually be called.

In the following we give an algorithm that calculates this set of functions for each calculated call in almost all practically relevant cases. The algorithm will also give a list of those functions that might be called by some calculated call, but the algorithm was unable to give a restrictive set of possible call sites. These must either be resolved by hand or we must assume that any calculated call can be to one of those functions. The algorithm is a special case of points-to analysis (see e.g. [4]) that is adapted to our requirements. In particular, points-to analysis described in this chapter is restricted to functions and pointers to functions. We only rely on the provisions made in Chapter 2 and we show a way to perform this kind of points-to analysis across module boundaries.

3.6.1 Preliminaries

The fundamental idea behind the algorithm is that the only way to calculate the address of a function is by taking its address via an address-of operator. I.e. if we go through all normalized ASTs of the program and search for all address-of operators that are applied to function declarations we get a complete list of all functions that can potentially be called at a calculated call. Additionally, we find all of the following program objects if they can contain function pointers according to their type:

- Global unaddressed variables
- Local unaddressed variables
- Unaddressed function parameters
- Unaddressed record fields
- Arrays that are
    - local to the current translation unit (static in C)
    - not local to a function, and
    - only used inside an array reference.

For the remainder of this section, we treat an array as if it was a single variable, i.e. an access to the array always means an access to an arbitrary element of the array.

All of these program objects have the favorable property that we can find all locations where their content is accessed. We call these objects function pointer objects for the remainder of this chapter.

3.6.2 Inclusion Graph

Given a function pointer object A, we define $FP(A)$ to be the set of functions that the function pointer given by A can point to. Whenever we have a place where a function pointer object A is assigned to another function pointer object B, we make the conservative assumption that $FP(A)$ is a subset of $FP(B)$. Additionally, if a function address is assigned to an object A, we must assume that $FP(A)$ contains that function. Aside from normal assignments, passing a function pointer object A to a function as an argument counts as an assignment of A to the function pointer object corresponding to the function parameter. We will assume for the moment that in this case we can uniquely determine the function being called, i.e. the call is not a calculated call. With this assumption the result of this step can be modeled as an inclusion graph where the nodes are the sets $FP(X)$ and there is a directed edge from $FP(B)$ to $FP(A)$ if we previously found out that $FP(B)$ includes $FP(A)$, i.e. $FP(A) \subset FP(B)$. An additional constraint is generated if the explicit address of a function f is assigned to a function pointer object A. This results in the explicit constraint that $f \in FP(A)$. The set of all functions that are explicitly included in $FP(A)$ by such constraints shall be denoted by $FP_{min}(A)$. We include a node for each function pointer object in the graph even if there are no edges to or from the object yet.

We want this inclusion graph to be a conservative model of all the sets $FP(X)$ in the graph. To make sure that this is actually the case we first add an artificial node $EXTERN$ to the graph. Initially $FP_{min}(EXTERN)$ is empty and there are no edges to or from $EXTERN$. Then we go through all statements where an address of a function is taken or where a function pointer object is used. For each use we distinguish the following cases:

- The effects of the statement are already modeled in the graph, i.e. the use is in one of the assignments described above.

- The function pointer object is the operand of an expression that can never yield a function pointer. E.g. if a function pointer object is compared to the NULL pointer, the result is boolean. Such a use of a function pointer object can safely be ignored.

- A function pointer object is used to specify the function being called in a call expression. These uses of function pointer objects do not contribute to the inclusion graph.

- All other read accesses to a function pointer object X generate an edge from $FP(EXTERN)$ to $FP(X)$, i.e. the set $FP(EXTERN)$ must include all functions in $FP(X)$.

- All other write accesses to a function pointer object X generate an edge from $FP(X)$ to $FP(EXTERN)$.

- If there is any other place where the address of a function is calculated via an address-of operator we include the affected function in $FP_{min}(EXTERN)$.

An informal description of these rules is: If a function pointer object is used in a way that is not modeled by other nodes in the inclusion graph (i.e. the function pointer "escapes" from the graph), $FP(EXTERN)$ includes those function pointers. Conversely, if a value is assigned to a function pointer object and this assignment is not modeled by other nodes in the graph (i.e. the pointer on the right hand side of the assignment "appears" from outside the graph), the FP-set of the function pointer object on the left hand side includes all functions in $FP(EXTERN)$.

3.6.3 Generic Algorithm

Once we have defined the inclusion graph as described in Section 3.6.2 we can in principle resolve the inclusion equations implied by the inclusion graph by any of the various algorithms for such a task. Actually the problem is very similar to the the one solved in conjunction with live ranges (see Section 3.5). However, the algorithm used there has drawbacks if we want to implement it efficiently in the scenery at hand. Instead we use standard graph algorithms to solve the set inclusion equations. First we decompose the graph into its strongly connected components. All sets in a strongly connected component are obviously equal, i.e. they contain the same elements. Second, We can replace the original graph with the directed acyclic graph of its strongly connected components. The nodes of this reduced graph can be processed in topologically sorted order to calculate the actual elements of each set.

Computational Complexity

Calculation of strongly connected components is done by two consecutive depth first searches of the graph (see e.g. [12]). Thus, the complexity of this step is linear in the number of edges in the graph. The actual bottom up calculation of the final sets at each node requires that we calculate the union of two sets for each edge in the graph.

The time required for the union of two sets depends on the number of elements in each of the sets and their representation. However, we can realistically assume that the maximum set size is bounded by a constant that is small compared to the number of edges in the whole graph, i.e. the time required to calculate the union of two sets is assumed to be constant. With this assumption, the time required for both steps is linear in the number of edges in the graph.

3.6.4 Optimization

The potential nodes of the inclusion graph have been carefully chosen in such a way that all edges induced by an arbitrary translation unit can be derived from that translation unit alone. I.e. we can construct the inclusion graph in a single pass through all translation units. However, the current algorithm still needs to keep track of all nodes that are generated along the way even though a large fraction of the nodes is only used in a single translation unit. Those nodes where this fact is obvious, namely nodes for local variables and static arrays, can be eliminated from the global tree after we have analyzed the file. This reduces the number of nodes in the graph and the information that must be stored globally.

Let $FP(N)$ be a finalized node in the inclusion graph, i.e. we know all functions that are directly included in $FP_{min}(N)$, all end points $FP(o_1), FP(o_2), \ldots, FP(o_p)$ of outgoing edges, and all start points $FP(i_1), FP(i_2), \ldots, FP(i_q)$ of incoming edges. We can eliminate all incoming edges of $FP(N)$ if we

- include $FP_{min}(N)$ in $FP_{min}(i_k), k = 1, 2, \ldots q$ and
- add an edge from $FP(i_k)$ to $FP(o_l)$ for $k = 1 \ldots q, l = 1 \ldots p$.

Once all local nodes of a translation unit have their incoming edges eliminated in this way, they can only include explicitly named functions and functions included in non local nodes. We keep this information locally in the translation unit where the nodes are used. As a consequence all outgoing edges of nodes that are not local to a single translation unit, point to nodes that are not local to a single translation unit, either. This means that we can now run the algorithm on non local nodes only.

Theoretically, each step that eliminates incoming edges of a local node can potentially add $O(|V|^2)$ edges to the graph where $|V|$ is the number of vertices in the inclusion graph. However, the number of edges of a local node is very small in virtually all practically relevant scenarios (one incoming and one outgoing edge is very common for local variables).

3.6.5 Extension

One important case has been ignored in the discussion above: A function pointer object is passed to a function as a parameter and the called function is itself given by a function pointer object. This problem can be solved efficiently for most practical cases but the worst case computational complexity is hard to assess. Whenever such a function call occurs we save the following items in a global list and do not generate edges to or from $FP(EXTERN)$:

- the name C of the function pointer object being called,
- the name P of the function pointer object that is passed as a parameter, and
- the index i of that function pointer object in the parameter list of the call to C.

Additionally, we make sure that function pointer objects referenced in this global list are not eliminated even if they are local to a single translation unit.

Each entry in this list describes a class of edges that must be added to the inclusion graph: For each function f in $FP(C)$ there must be an edge from the set for the i-th parameter of f to $FP(P)$. Once the inclusion graph not containing these edges is calculated we go through the list and use it repeatedly to add edges to the inclusion graph as follows:

- Resolve all constraints according to the generic algorithm given in Section 3.6.3. If A is an arbitrary function pointer object this gives a first estimate for $FP(A)$.
- For each entry in the list described above add explicit edges that reflect the given constraint assuming that the sets generated by the previous step are already complete.
- If the previous step made any changes to the inclusion graph: repeat.

The computational complexity obviously depends on the number of iterations and it is certainly possible to construct cases where this number can become large. However, tests with the real life programs used in our case studies (see Chapter 7) did not require more than two iterations.

For a function pointer object X that specifies the function in a calculated call, $FP(X)$ gives a conservative set of functions that might by called. For other calculated calls this set must be assumed to be $FP(EXTER)$.

3.7 Static Structure Fields

3.7.1 Definition

Given a pointer to a structure we must assume that the structure resides somewhere on the heap and that other threads can also hold references to the same structure. Consequently, it is possible that the contents of the structure change at any point in time. Sometimes it is still possible to verify that an unaddressed structure field does not change unexpectedly. One obvious case is if the structure field is never used on the left hand side of an assignment, i.e. the field is never changed. While this may sound rather trivial it is often useful because the field can still be initialized as a part of a variable declaration. Additionally, if we know that a pointer points to unaliased memory, an assignment to a structure field through that pointer cannot implicitly change the field in another control flow. If all modifications of an unaddressed field are in such a context, we can conclude that the field is not modified without an explicit assignment in the current control flow. We call a field with this property a *static* field.

Static fields are actually rather common. In the case studies we conducted (see Chapter 7) about one third of all fields turned out to be static.

3.7.2 Unaliased Memory

The identification of pointers that reference private unaliased memory plays a key role in the determination of static structure fields. We know that the return value of a memory allocation function is a pointer to unaliased memory, i.e. the pointer returned by an allocation function is currently the only way to access that memory object. Thus, given a memory allocation we want to identify stores into that memory that happen provably before the memory can be accessed outside the current thread. As long as the pointer returned by the memory allocation is only stored in local unaddressed variables, it is still guaranteed that only explicit accesses via one of those variables can change the contents of the memory object. But then, if one of those variables is used in a way that can make the pointer available outside the current function or thread, we must assume that the memory is aliased from then on. Basically, the only safe operations that do not publish the pointer are:

1. Statements that do not use any of the variables that might contain the pointer.
2. Use of the pointer as an operand of a comparison operation that yields a boolean result. The most common case is a comparison with a NULL pointer.
3. Assigning the pointer to a local unaddressed variable. This variable can now obviously contain the pointer as well.

4. Assigning the result of an arithmetic operation that uses the pointer to a local unaddressed variable. We must conservatively assume that the pointer value can be reconstructed from the result. I.e. the variable on the left hand side of the assignment can potentially be used to make the pointer public.[2]

Unfortunately, this definition of safe statements depends on the set V of variables that might contain a value derived from the pointer. Let us assume for the moment that we already know this set of variables. In this case we can use a depth first search to find all statements that are reachable from the allocation statement via at least one code path that consists entirely of safe statements. If we discover additional variables that might contain the pointer in cases 3 or 4 from the list above we add these variables to V and repeat the depth first search for safe statements with the modified set V. Once the set V does not change anymore, we get a set S of candidate statements and a fixed set V of variables. It is important to note that along any code path that starts at the allocation statement and consists only of statements in S only variables in V can contain a pointer to the newly allocated memory area.

In the next step, we remove all statements from S that are not dominated by the allocation statement or that are reachable from the allocation statement via a code path that contains a statement that is not (or no longer) in S. All statements remaining in S are dominated by the allocation statement and all possible code paths from the allocation statement to a statement in S contain only statements that are also in S. At this point we know for each statement s which remains in S that

- at s the pointer has not been published and
- at s only variables in V can point to the newly allocated memory.

Note that given a statement $s \in S$ and a variable $v \in V$ it is not yet clear if v really points to the newly allocated memory area. Thus we consider each pair $(s, v) \in S \times V$ and try to prove that after the statement s the variable v always contains a pointer to the newly allocated memory area. We will abbreviate this property by $PTR(s, v)$. The following rules allow us to prove $PTR(s, v)$ for some candidate statements. $PTR(s, v)$ holds if one of the following is true:

- s is the allocation statement and v is the variable that initially contains the pointer.
- s is an assignment to v, the right hand side is a variable $v_2 \in V$, and $PTR(s, v_2)$ holds.
- s is not an assignment to v and $PTR(p, v)$ holds for all immediate predecessor statements p of s in the CFG, i.e. for all p with $s \in CFG\_next(p)$.

---

[2] This case is actually forbidden in a normalized AST, it is still useful in general though.

For each pair $(s, v) \in S \times V$ the function PTR can be evaluated recursively using the rules above. To speed this up a result that is already calculated can be cached and reused in subsequent evaluations of PTR.

If $PTR(s, v)$ holds for a statement s that stores data into the newly allocated memory area, this store can be considered an initialization.

3.8 Summary

In this chapter we have described the following basic techniques that will be helpful in the forthcoming chapters where the behavior of the program along different control flows is analyzed:

Names are assigned to all translation units, nodes in ASTs, and exported objects. We have also discussed their scope and uniqueness.

Addressed Variables and Structure Fields were identified. The detection of additional return values allowed us to treat some fields or variables as unaddressed if all address-of operations are well understood.

Basic Blocks and the Control Flow Graph were defined and algorithms to calculate these data structures were given.

Live Ranges of function local variables were determined by a simple iterative algorithm.

Function Pointers were analyzed and we gave an algorithm that calculates a conservative set of actual functions that each function pointer can point to.

Static Structure Fields were defined and a method to identify many static structure fields was given.

Chapter 4

Function Body Analysis

In this chapter, we will discuss analysis methods for a block of code – usually the body of a function – while function calls and inter procedural analysis is postponed to the next chapter. The main tool used in the analysis is a so-called *predicate* that describes the important aspects of the system's overall state.

4.1 Predicates

The *data space* of a program is the set of all memory objects that are used by the program. For each object there exists a well defined set of values that are allowed for the object. A function that assigns a specific value to each object of the data space is called a *state* of the data space. Each statement that is executed performs a well defined transformation of the data space and advances the program pointer to the next statement. The next statement may depend on the new state of the data space, e.g. in case of a conditional expression. Let us denote the set of all statements in a program by S and the set of possible states of the data space by D. If we consider a single threaded program, its execution can be interpreted as an automaton whose states are all pairs $(d, s) \in D \times S$, i.e. a state of the data space together with a current statement. The transitions of the automaton are determined by the effects of a statement. More precisely if the automaton is in state (d_0, s_0) this means that the data space is in state d_0 and the next statement to be executed is s_0. If we ignore the possibility of external input, s_0 modifies the contents of the data space in a deterministic way, i.e. from d_0 it is clear how the data space looks after the execution of s_0. Let us call this state of the data space d_1. Additionally, the next statement s_1 that is executed can be derived from d_0 and s_0 deterministically. Thus the automaton has an edge from (d_0, s_0) to (d_1, s_1). If s_0 is a statement that reads input from an external source, there are multiple possibilities for the contents of the data space, one for each value that is possible for the memory object that the input value is stored in.

As we do not know anything about the input this makes the automaton effectively non-deterministic.

Obviously, even if the program is single threaded it is practically impossible to track all possible states of the data space. In order to deal with this problem we will define sets consisting of many states of the data space (called *predicates* from now on) and let the automaton operate on these predicates. Different predicates can have a non-empty intersection, i.e. it is possible that a state d of the data space is included in two distinct predicates p_1 and p_2. Additionally it is useful to include D, i.e. the set of all possible states of the data space as a predicate. Let us assume that we have a given family P of predicates and at least $D \in P$. We can define a not necessarily deterministic automaton that conservatively models the execution of the program according to the following restrictions:

- The states are all pairs $(p, s) \in P \times S$.
- Let (p_1, s_1) be a state of the new automaton and $d_1 \in p_1$. If the original automaton can transit from (d_1, s_1) to (d_2, s_2) our automaton must have a transition from (p_1, s_1) to (p_2, s_2) for some $p_2 \in P$ that satisfies $d_2 \in p_2$.

Note that $p_2 := D$ satisfies all constraints independent of p_1. Thus it is always possible to construct such an automaton if $D \in P$. This also shows that the automaton defined above is not uniquely determined by the family P.

4.1.1 Specification of Predicates

In this section we describe how predicates are specified for practical purposes. This is generally done by *excluding* certain states in D from the predicate, i.e. the predicate consists of all states of the data space in D that are not explicitly excluded from the predicate. The following properties of a data space state can be used to restrict which data space states belong to a predicate:

Value Restrictions The predicate can require that the value of a variable v is in a given not necessarily contiguous range. This will exclude all data space states where the value v is not within that range.

Individual Bits A predicate can require that some bits of a variable v_1 must be set or clear. Obviously this information is partly redundant with value restrictions. However, depending on the context, one way or the other will turn out to be more convenient to express the restricted set of values that the variable can have.

Equality The predicate can require that two variables v_1 and v_2 have the same value. This obviously implies that both variables have the same restrictions for their values and for individual bits.

Inequality Analogously, a predicate can require that two variables v_1 and v_2 have different values. As opposed to an equality, an inequality does not impose additional restrictions on the range of values or on individual bits.

Structure Fields Given a variable v_1, a pointer variable v_2 that points to a structure, and a field declaration f for a field inside that structure, a predicate can require that the field f in the structure pointed to by v_2 has the same value as the variable v_1.

It is obvious that not all possible predicates can be specified in this way and theoretical considerations will still use the general definition of a predicate as a family of subsets of the set D consisting of all states of the data space. However, actual implementations of predicates will use these restricted predicates or a slightly generalized version thereof.

Having said that, it is also clear that these restrictions allow several ways to specify the same set of states of the data space. For example, it is possible that two predicates are equal in a non obvious way or that a predicate is actually empty even though we cannot prove this from its specification in the form given above. The most obvious case where this can happen is if ranges of values and individual bits are specified at the same time, e.g. if a variable v has bit number zero set (i.e. v is odd) and at the same time only allows even values. There is obviously no state of the data space that can fulfill both conditions. Other less obvious cases are also possible though: E.g. if the variables x, y, and z are mutually different and all of them must be either zero or one this is also impossible.

All k-coloring problems of graphs can be formulated as a predicate where each node in the graph corresponds to a variable in the predicate that is restricted to the values $1 \ldots k$. Additionally, variables that correspond to adjacent nodes in the graph must have different values. This predicate is not empty if and only if there exists a solution to the corresponding coloring problem. As graph coloring problems are known to be NP-hard under various restrictions[21], we can conclude that deciding whether a predicate is empty is also NP-hard in general.

Instead of taking all of these problems into account, we restrict ourselves to predicates that are specified as described above and keep in mind that seemingly different predicates might actually be equal. Where necessary, the results of operations that modify predicates are approximated in a conservative way.

4.1.2 Multi-Threaded Programs

So far, neither the automaton defined above nor the predicates take multi-threaded programs into account. We will now add a restriction to the predicates that will make the automaton defined above suitable for the analysis of a single control flow in a multi-threaded program: Let p be a fixed predicate and $d_1, d_2 \in D$ any two states such that d_1 can be transformed into d_2 by some statement that can be executed outside of the current control flow. The predicate p is called *multi-thread* ready if $d_1 \in p$ implies $d_2 \in p$ for all such $d_1, d_2 \in D$.

If we restrict the set P of predicates used to define the automaton to predicates that are multi-thread ready, the automaton conservatively models the execution of a single control flow even in the presence of other control flows.

In fact this is a natural restriction if we want to analyze a single thread in a multi-threaded environment because data that can be modified by a different thread at any time is unlikely to contain meaningful information about the state of the current thread. I.e. the restriction to multi-thread ready predicates in general does not reduce the precision of code analysis but rather uncovers hidden assumptions that may not be always true.

Sufficient Conditions

The general definition of a multi-thread ready predicate given above can sometimes be hard to check in practice. Thus, we give two conditions that can be used to construct multi-thread ready predicates:

Local variables Local variables (as long as they are unaddressed) can only be accessed by the current thread. Thus it is safe to add value or bit restrictions, equalities, and inequalities involving only local variables.

Static structure fields Structure field information can be stored in a predicate by giving a pointer to the structure, a structure field declaration, and a variable that equals the structure field. Static structure fields as defined in Section 3.7 cannot change their value. Thus, it is safe to add structure field information for such a field if both the pointer to the structure and the variable that equals the contents of the structure field are local unaddressed variables.

These conditions guarantee that any predicate constructed in this way is multi-thread ready. However, we have to allow additional predicates later on to deal with certain rather common program constructs. Whether such predicates are multi-thread ready, must be discussed separately.

4.2 Representing and Modifying Predicates

4.2.1 Representation

In the representation of a predicate we separates the variable names from the description of their properties. Consequently, the representation of a predicate consists of a set of variable descriptions and a map that associates each variable name in the predicate with a variable description. This way it is possible to use the same variable description for different variables that are known to be equal. If a variable is not mentioned in the map it is not restricted in any way by the predicate.

A variable description contains the restrictions on a variable as specified in Section 4.1.1. More precisely each description contains the following elements:

- The possible values of the variable. Consecutive values are combined into a range, i.e. the possible values of a variable that is neither zero nor ten is represented as $-\infty \dots -1, 1 \dots 9, 11 \dots \infty$. We will use more compact albeit less normalized range specifications in human readable representations of predicates as long as it is clear how these can be transformed into the normalized form.
- A bit mask *zerobits* that specifies all bits of the variable that must be zero.
- A bit mask *onebits* that specifies all bits of the variable that must be one.
- Each description contains two possibly empty lists to describe the field-of relations of that variable, namely the *fields* list and the *field-of* list. If the field given by the declaration d in a structure pointed to by p has the same value as the variable v,
  - the *fields* list in the description of p contains the pair (d, v) and
  - the *field-of* list in the description of v contains the pair (d, p).
- A list of variable names that are equal to the variable being described. All of these variable names are mapped to the same variable description.
- A list of variable names that are not equal to the variable being described. This list is obviously disjoint with the list of equal variables.

4.2.2 Copy on Write and Hashing

In an actual implementation a new predicate will generally be derived from an existing predicate by applying several elementary operations to a copy of the original predicate. These operations are discussed in more detail below. After all modifications are done, the predicate is *finalized*. A finalized predicate can no longer be altered. During finalization it is useful to perform the following steps as well:

Useless Variables It can sometimes happen that a variable is mentioned in the predicate but it does not carry any information, i.e. all values are allowed and the variable is not part of any field-of relation. These variables are removed from the predicate during finalization.

Hash Calculation A hash value is calculated from the contents of the predicate. The hash function is designed in such a way that predicates that are obviously equal, i.e. those where we can show that they define the same subset of data space states, get the same hash value.

Variable Signature Additionally, each name of a variable that is used in the definition of the predicate is hashed to a small integer value. All integers that are calculated in this way are combined in a bit mask. Each bit that corresponds to the hash value of a variable name is set in the bit mask all other bits are clear. This bit mask is called the variable signature of the predicate.

Both the hash value and the variable signature are stored in the finalized predicate. They are used to speed up otherwise expensive operations on predicates. E.g. finding two predicates that are equal in an obvious way in a larger family of predicates can be done much faster with the help of the hash value because only those predicates that have the same hash value must actually be compared.

4.2.3 Modifying Predicates

The following operations can be used to modify a predicate that is not yet finalized. If any of the operations detects that the resulting predicate is empty, i.e. it does not include any state of the data space, an error is returned.

Restricting Values of a Variable

Restricting the values of a variable to a given range is one of the simplest operations on a predicate. First, if the variable is not mentioned in the predicate yet, it must be added with a range of $-\infty \dots \infty$. Afterwards we calculate the intersection of the old range of the variable, which may be different from $-\infty \dots \infty$ if the variable was already mentioned in the predicate, and the given range that the variable should be restricted to. The result is the new range that is allowed for the variable. If the resulting range is empty, an error is returned. If the resulting range contains only a single value, the bit masks for that variable are updated accordingly.

Restricting Bits of a Variable

Restrictions on the bits of a variable can be added by supplying a set of bits that must be zero and another set of bits that must be one. These bit sets are added to *zerobits* or *onebits*, respectively. If this results in a specific bit being set in both *zerobits* and *onebits*, an error is returned.

Adding a Field-Of Relation

Three items must be specified in order to add a new field-of relation:

- The pointer variable p which must already be mentioned in the predicate,
- a field declaration d and
- the variable v that holds the same value as the structure field. This variable must also be mentioned in the predicate.

If there is already a field-of relation that involves the same pointer and the same field declaration with a variable v_2, it is obvious that v and v_2 must be equal. Thus we add an equality between v and v_2 to the predicate (see Section 4.2.3). Any error that might occur while adding the equality is propagated. If the field-of relation is new, we add the pair (d, v) to the *fields* list of the pointer p and the pair (d, p) to the *field-of* list of v (see Section 4.2.1).

Adding an Equality

Adding an equality between two variables v_1 and v_2 leads to the creation of a new variable description. Both variable names and the names of variables that were already equal to either v_1 or v_2 are mapped to the new description. The new description only allows values for the variables that were previously allowed for both v_1 and v_2 because otherwise the variables cannot be equal. Thus,

- the range of possible values is the intersection of the previous ranges of v_1 and v_2,
- zerobits and onebits contain all bits that were also set in the respective bit mask of v_1 or v_2 and
- the list of equal and unequal variables is the union of the respective lists of v_1 or v_2.

Maintaining the *fields* lists requires a little more care. As a first step the *fields* lists of v_1 and v_2 are concatenated. Unfortunately this can result in a list with two entries (d, x_1) and (d, x_2) for the same field declaration d. If x_1 and x_2 are equal one of these entries is superfluous and can be removed. However, if x_1 and x_2 are not yet equal an equality must be added first. The *field-of* lists can simply be joined, it is useful to remove duplicate entries that might occur though. Once the new description is complete we also have to update the list of unequal variables in other descriptions.

If the resulting range is empty, a single bit is set in zerobits and onebits, or if a variable is required to be unequal to itself, an error is returned. Any errors that result from additional equalities that are added as a consequence of the merge of the field lists are propagated to the caller.

Adding an Inequality

Adding an inequality between two variables v_1 and v_2 is rather simple: We only have to add v_1 and all variables that are equal to v_1 to the list of unequal variables in the description of v_2 and vice versa. If v_1 and v_2 are already required to be equal an error occurs.

Deleting a Variable

Deleting a variable v basically means that the entry for v in the map is deleted. The description associated with the variable will survive if there are other entries in the map that use it. After the map entry has been deleted, we search for all occurrences of the name in *fields* and *field-of* lists. If there is another variable w that is (or used to be) equal to v each occurrence of v in these lists is replaced with w, otherwise the list entries are deleted. Additionally, all occurrences of v in the lists of equal or unequal variables of other descriptions are removed.

Deleting a variable is the only operation that modifies a predicate in such a way that the number of states included in the predicate increases. All other operations reduce the number of states included in the predicate.

4.3 Inspecting a Predicate

Once a predicate is finalized, the operations described in Section 4.2.3 are no longer allowed. A finalized predicate is only inspected or compared to other predicates. This section describes the possible operations. These operations will not be able to return an accurate answer for all possible predicates. In these cases the return value will indicate that the question could not be answered. In all cases, a definitive positive or negative result will only be returned if there is no doubt that the result is correct. One will still have to keep in mind that these functions can in fact return saying “unknown”.

4.3.1 Range of a Variable

We will frequently need to know if the value of a variable is in a given set s_{test} of values. Let s_v be the set of values that are possible for the variable in question according to its description in the predicate. If the variable is not mentioned in the predicate, this means that $s_v = (-\infty \ldots \infty)$. Three cases are possible:

- s_v is a subset of s_{test}. In this case the answer is obviously positive, i.e. the value of the variable is in the set s_{test}.
- s_v and s_{test} are disjoint sets, this results in a negative answer.
- In all other cases, the answer is unknown and the return value reflects this.

4.3.2 Bits of a Variable

The same thing can be done for bits: Given a set one_{test} of bits that must be set and a set $zero_{test}$ of bits that must be clear, we can test if a variable v meets these requirements. If some bit is in both one_{test} and $zero_{test}$ this is impossible. Otherwise, we compare these sets to the sets one_v and $zero_v$ of bits that are set or clear respectively according to the variable description in the predicate. The return value is true, if one_{test} is a subset of one_v and $zero_{test}$ is a subset of $zero_v$. If the intersection of $zero_v$ and one_{test} or of $zero_{test}$ and one_v is not empty, the return value is false. In all other cases, the answer is unknown.

4.3.3 Equality of Two Variables

The only case where we know for sure that two variables are identical is if the predicate explicitly states this, i.e. both variable names refer to the same description. If the predicate explicitly states that the variables are different, we clearly know the answer as well. However, there are additional cases where we can conclude that two variables are not equal:

- Their value ranges are disjoint.
- Some bit must be set for one variable and clear for the other.

4.4 Binary Operations on Predicates

In the handling of finalized predicates there are a few binary operations that will frequently be used. These take two predicates and potentially other parameters.

4.4.1 Special Case

Given two predicates p_1 and p_2 we often want to know if p_1 is a special case of p_2. More precisely, p_1 is called a special case of p_2 if all states of the data space that are in p_1 are also in p_2. For most cases the question can be answered by looking at the description of the predicate:

- Variables that are mentioned in p_2 must also be mentioned in p_1.
- Values that are allowed for a variable in p_1 must also be allowed in p_2.
- Individual bits of a variable that are restricted in p_2 to either zero or one must be restricted in the same way in p_1.
- Two variables that must be equal or unequal according to p_2 must also be equal or unequal respectively according to p_1.
- A field-of relation in p_2 must have a corresponding field-of relation in p_1. The variable names used in the description of the field-of relation do not necessarily have to be equal. It is sufficient if there is a field-of relation in p_1 that uses the same field declaration and the same pointer where the respective variables used for the value of the field are equal according to p_1. More precisely: Assume that the *field-of* list in the description of a variable *ptr* in p_2 contains an element (d, val_2) where d is a field declaration and val_2 is the name of the variable that holds the value of the field in the structure pointed to by *ptr*. If p_1 is a special case of p_2, the *field-of* list of *ptr* in p_1 must contain an element (d, val_1) where val_1 is equal to val_2 according to the predicate p_1. It is not necessary that the variable names val_1 and val_2 are identical.

As an optimization we can first check if the variable signature (see Section 4.2.2) of p_2 includes bits that are not in the variable signature of p_1. In this case p_1 cannot be a special case of p_2.

The conditions defined above are sufficient to conclude that p_1 is in fact a special case of p_2. The special-of test returns *true* if these conditions are met and *false* otherwise. This is slightly different from the tests in the previous section because the return value cannot be *unknown*. Instead *false* means that we could not prove that p_1 is a special case of p_2.

4.4.2 Equality of Two Predicates

Based on the special case test described in the previous section we can easily test if two predicates p_1 and p_2 are equal. This is the case if p_1 is a special case of p_2 and vice versa. If the hash values of p_1 and p_2 that were calculated during finalization differ we can conclude that the predicates are different without doing potentially expensive special case checks. Again, this test does not return *unknown*.

4.4.3 Merging

Two predicates p_1 and p_2 can be merged into a single predicate that includes all states that are in at least one of the predicates p_1 and p_2. The result is a new predicate. Depending on the way the merged predicate is described, it will in general include additional states. If all else fails, the trivial predicate (written as []) which includes all states and does not restrict anything can always be used as the merged predicate. This implies that replacing two predicates with a single merged predicate potentially discards information. Thus merging predicates is avoided except in a few special cases where the loss of information is negligible even if the merged predicate is described in the standard way.

If the predicates are equal or one is a special case of the other, the more general predicate obviously describes the merged predicate exactly. Additionally, if the two predicates differ only by the values allowed and the bits that are set or clear for a single variable and are otherwise identical, a merged predicate can be constructed from one of the predicates by allowing all values and bits for the variable in the merged predicate that are allowed in at least one of the original predicates.

In fact, the latter construction of a merged predicate is not always exact: Consider for example a predicate p_1 that requires that the value of a variable v satisfies $0 \leq v \leq 10$ and that the least significant bit is set and a predicate p_2 that requires that $5 \leq v \leq 20$ and the least significant bit is clear. A merged predicate constructed as described above would require that $0 \leq v \leq 20$ and not make any restriction for the bits of v. This clearly allows additional states in the merged predicate that are neither in p_1 nor in p_2. However, most of the time a variable will either have restrictions for its values or restrictions for its bits but not both. In this case, the given construction of the merged predicate exactly describes the union of all states in p_1 and p_2.

Even if the description used for the merged predicate does not describe the union of the states in p_1 and p_2 exactly, this construction always guarantees that the merged predicate includes all states that are in at least one of p_1 and p_2.

4.4.4 Intersection

Given two predicates p_1 and p_2 it is sometimes useful to generate a predicate that includes only those states that are in both p_1 and p_2. We assume for the sake of simplicity that p_1 is not yet finalized, i.e. we are allowed to modify p_1 directly. If this is not the case we must start out with an identical copy of p_1. In all cases p_2 is always accessed read only. With this assumption the intersection of p_1 and p_2 can be calculated by applying a series of basic modification operations to p_1. All of these operations have previously been described in Section 4.2.3:

- As a first step, all variables that are mentioned in p_2 but not in p_1 are added to p_1 without any restriction.
- If two variables must be equal according to p_2 we add an equality between those two variables to p_1. The same applies to inequalities.
- Any field-of relations in p_2 are added one by one to p_1. This might add more equalities.
- If a variable is mentioned in p_2, its range and bits in p_1 are further restricted according to p_2.

Note that any restrictions originally imposed by p_1 remain intact. The result is a predicate that includes exactly those states that are in p_1 and p_2. It is important to note that the intersection as opposed to the union of two predicates can always be described accurately.

Mapped Intersection

Actually, a generalized version of this operation will turn out to be useful. The key point is that all variable names mentioned in p_2 are translated according to a map that is given as an additional parameter. This translation can be done implicitly by replacing all variable names with their mapped version in all operations applied to p_1. This allows us to still use a finalized read only predicate for p_2.

This version of the intersection will be useful in conjunction with function calls where formal parameters are mapped to actual parameters.

4.5 Predicates and Statements

We want to use predicates and the operations defined in the previous sections to reason about the effects of a block of code. Conceptually, we use a non-deterministic automaton as outlined in Section 4.1. Recall that this automaton operates on an externally specified set P of predicates. Thus, we have to specify how the set P is chosen. As we have to maintain an actual list of all predicates in P the set must be relatively small. Then again, it should be possible to describe a large part of the states that the program can be in without losing too much information.

Before we continue with a formal description of the automaton we give an intuitive interpretation and a rather informal description of the logic behind it: At each point in a block of code there is a set of predicates that includes all states that are possible at that point. We only use predicates that are multi-thread ready which basically means that the facts stated in a predicate cannot be changed by a control flow different from the one being analyzed. Consequently, other control flows have no influence on the set of predicates describing all states that are possible at a certain point in the program. Now, if a set of predicates describes the states that are possible before a statement s, we can apply the effect of s to each predicate. The result is a set of predicates that describes the states that are possible after s. Predicates that are newly created in this process, are added to the set P. This step must be repeated if the predicates that are possible before s change. This is very likely to happen several times for each statement if the block of code contains jumps and loops. Once nothing changes, the predicates that are possible at the end of a block can be used as an aggregate description of the whole block.

In order to guarantee termination, it is possible to limit the size of the set P. A predicate that would have to be added to P after that limit is reached, is replaced by the trivial predicate [] that is always included in P. As Enforcing this limit introduces inaccuracies, it is often undesirable. Fortunately, the case studies in Chapter 7 show that this is not a problem in practice.

4.5.1 Pre- and Post-Predicates

Let S be the set of statements in a block (usually the body of a function). For each statement $s \in S$ we will maintain two lists of predicates: $\mathit{Pre}(s)$ and $\mathit{Post}(s)$. Initially all lists are empty except for the first statement s_0 in the block where we initialize $\mathit{Pre}(s_0)$ with the trivial predicate which includes all states of the data space. $\mathit{Pre}(s)$ will contain a set of predicates that describe all states that are possible immediately before the execution of s. Similarly, $\mathit{Post}(s)$ will contain a set of predicates that describe all states that are possible immediately after the execution of s. Introducing the $\mathit{Post}(s)$-sets allows us to separate the effects of a statement on predicates in $\mathit{Pre}(s)$ from the propagation of the result along the edges of the control flow graph. Each predicate in $\mathit{Post}(s)$ is propagated to the *Pre*-set of at least one statement that immediately follows s in the control flow graph.

4.5.2 Per Function Predicate List

During analysis of a block of code or a function, a list P of all predicates that are part of one of the *Pre*- or *Post*-sets is maintained independently of these Sets. The *Pre*- and *Post*-sets include predicates by reference.

After a newly constructed predicate has been finalized, we first check if it is equal to one of the predicates that are already in P. In this case the existing predicate is used instead. Additionally, whenever a predicate p is about to be added to the set

Pre(s) we check if p is a special case of some predicate $q \in$ *Pre*(s). In this case there is no need to include p in *Pre*(s) because q already includes all states that are in p. Finally, if the new predicate p and an existing predicate $q \in$ *Pre*(s) differ only in the description of a single variable, the predicates p and q are merged as described in Section 4.4.3. The merged predicate then replaces q in *Pre*(s). This is important because it reduces the number of elements of *Pre*(s). Additionally, a new predicate that might be added to *Pre*(s) in the future is more likely to be a special case of the merged predicate. This potentially reduces the number of elements in *Pre*(s) further.

4.6 Predicate Transitions at Statements

This section describes how one or more new predicates are constructed from the set *Pre*(s) and the rules that are used to distribute the predicates in *Post*(s) among the successors of s in the control flow graph. The new predicates constructed from a predicate $p \in$ *Pre*(s) will describe all states of the data space that are possible immediately after the execution of s if the data space is in one of the states described by p, immediately before the execution of s. All predicates constructed in this way from the predicates in *Pre*(s) comprise the set *Post*(s). The elements of *Post*(s) are then inserted into the appropriate *Pre*-sets of the successors of s in the control flow graph. The remainder of this section will give detailed descriptions of this process depending on the type of the statement.

Provided that the predicate $p \in$ *Pre*(s) that is used as a basis for the construction of new predicates is multi-thread ready, the following constructions of new predicates are designed to ensure that the newly constructed predicates are multi-thread ready as well. This means that all predicates in the *Pre*- and *Post*-sets will be multi-thread ready if all predicates that are initially in one of these sets are multi-thread ready. The only predicate that is initially in one of the *Pre*- and *Post*-sets is the trivial predicate which is always multi-thread ready. Thus all predicates constructed according to the rules in this section will be multi-thread ready.

4.6.1 Assignments

Throughout this subsection, $p \in$ *Pre*(s) is a predicate that describes a subset of the states that are possible immediately before the execution of s. Now let s be an assignment statement without side effects. The case of an assignment statement that has side effects, i.e. the right hand side is a function call, is considered in Chapter 5.

If the left hand side of the assignment is a variable or a formal function parameter v, we can start with a predicate q that is constructed from p with all restrictions on v removed. If the left hand side is something else, we still remove all the references to the object on the left hand side. Particularly, if the left hand side is a component reference, we remove all field-of relations that use the same field declarations independent of the pointer. The predicate p is multi-thread ready as an element of $Pre(s)$ and removing restrictions as described above cannot change this. Consequently, q is multi-thread ready as well.

In theory, the resulting predicate q is suitable for inclusion in $Post(s)$ because it includes all states that are possible after s if the data space is in a state that is in p before the execution of s. However, in some cases it is possible to add additional restrictions while at the same time the predicate is still multi-thread ready and includes all necessary states. Thus, if none of the following special cases is applicable, q becomes part of $Post(s)$. Even if one of the conditions is applicable it is not clear that the resulting predicate is itself multi-thread ready. In general we can only prove this if all relevant variables are local and unaddressed (see Section 4.1.2). Because of this, all of the special cases that follow are only applied if we can show (normally by the criteria given in Section 4.1.2, i.e. all variables involved must be local and unaddressed) that the resulting predicates are multi-thread ready.

Variable-Variable Assignment

If both sides of the assignment are variables or formal function parameters their contents must be equal after the assignment. Thus, we add an equality for those two variables to q.

Constant Right Hand Side

If the right hand side is constant and the left hand side is an unaddressed local variable its bits and its value are restricted according to the value given by the constant.

Comparison on the Right Hand Side

If the right hand side tests the equality of two variables or formal function parameters the predicate q is split into two predicates that are initially identical to q. In the first one, the two variables involved in the comparison are equal and the variable on the left hand side is true while in the second predicate both properties are reversed, i.e. the variable on the left hand side is false and the variables involved in the comparison on the right are different. If both predicates are inserted into $Post(s)$ we are again guaranteed that all necessary states actually are in $Post(s)$. The case where the operator on the right hand side is *not equal* ($\neq$) is treated analogously. The only difference is that the value on the left is negated compared to the case of an equality.

Note, that we cannot represent the fact that a variable is less than or greater than another variable in a predicate. However, we do know that a true result for the comparison operators $>$ or $<$ implies that the variables being compared are unequal.

Conversely, we also know that the result for the comparison operators $\leq$ and $\geq$ is true if the variables being compared are equal. In these cases we still split the predicate into two predicates: One where the variables involved in the comparison are equal and one where they are different. However, we only know the value of the left hand side for one of the two predicates, in the other predicate the variable on the left hand side is unrestricted.

If the right hand side is any comparison that involves one variable and one constant, a similar approach can be used. Again q is split into two predicates: One where the variable on the left hand side is true and one where it is false. The single variable used on the right hand side is restricted such that the comparison with the constant is true or false, respectively. As opposed to a comparison of two variable this works for all comparison operators alike.

In all cases discussed in this paragraph, we generate two predicates for each predicate in $Pre(s)$. However, it is possible that either or even both of these predicates turn out to be empty, i.e. the predicate does not contain any state at all. This is the case if the result of the comparison on the right hand side can be deduced from the restrictions in the original predicate. There is no need to add these empty predicates to $Post(s)$.

Logical Expression on the Right Hand Side

If the right hand side is a logical expression, the assignment can be replaced by nested if statements that directly assign the proper result inside their branches. Conceptually, assignments with a logical expression on the right are treated as if they were rewritten in this way. As with comparison operations it is possible that the result of such a logical expression can be derived from the restriction on the operands. In this case no rewriting is necessary and the variable on the left hand side is simply restricted in accordance with the result.

Component Reference on the Right Hand Side

If the right hand side is a component reference with pointer ptr and field declaration d where d has been found to be static by the process described in Section 3.7, we add an appropriate field-of relation to the predicate. Special care must be taken if the predicate p already contains this field-of relation. In this case we just add p to $Post(s)$ because the statement in fact does not change anything. Figure 4.1 shows a code fragment where this optimization makes a difference. If field is indeed a static structure field we know that the second assignment to x does not change anything because x already contains the same value as `ptr->field`. We also know that at this point the value of x is zero. However, the normal procedure would be to remove all

```
x = ptr->field;
if (x == 0) {
        x = ptr->field;
}
```

Figure 4.1: Component Reference Optimization

restrictions on x which means we know nothing. Subsequently we add the field-of relation again but there is no way to recover the information that the value of the field is in fact zero. Conversely, if we first check if the assignment does not change anything, we can preserve the information that the value of the field is zero.

One might ask why a redundant statement like this was written in the first place and if the compiler cannot remove it during optimization. These kinds of redundant assignments arise because in general the compiler cannot know if `ptr->field` changes between two accesses, i.e. the value must be reloaded from memory. In contrast, we know from previous analysis that this particular field is static, i.e. it cannot change between these two accesses.

Integer Type Conversion

If the right hand side of the assignment converts an integer variable v from one integer type to another, we check if all the values that are possible for that integer according to q can be represented in all integer types involved in the conversion. The same check is performed for those integer values that correspond to bits which are restricted in the description of the variable on the right side of the assignment. If all of these checks succeed the conversion does not change the value and we treat the assignment as if the conversions had been omitted.

Bit Operations

This paragraph considers assignments where the right hand side is a bit operation. The simplest case is a bitwise negation of a variable. In this case all bits that are known to be zero for the variable on the right hand side must be one for the variable on the left and vice versa. Note, that only the bits of the variable on the left hand side are restricted while the range of values that are possible for the left hand side according to the predicate remains $(-\infty, +\infty)$.

Similarly for bitwise "and", bitwise "or", and bitwise "xor" expressions on the right side, we inspect the operands and find out what is known about their bits. If this allows conclusions about some bits in the result, these bits of the variable on the left hand side are restricted accordingly. Note that this also works if any of the

```
i=0;
while (i<100) {
        /* do something */
        i = i+1;
}
```

Figure 4.2: Handling of arithmetic expressions

operands is a constant. Again there is no other restriction on the value of the variable on the left hand side.

Finally, if the right hand side is a left or right shift by a constant amount of bits we find out what we know according to q about the bits of the variable that is shifted. This information is shifted by the specified amount of bits and the variable on the left side is restricted accordingly.

Contrary to intuition, the bits shifted in from either side are considered unknown not zero. In particular this means that we do not know anything about the bits of the left side if we did not know anything about the bits of the variable on the right side. This decreases the number of states that must be considered significantly because we do not generate additional and often unnecessary information about bits at each and every shift operation. Conversely, existing information about bits is not lost due to a shift operation.

Address of the First Field

Assume that the Assignment is of the form y=&ptr−>fld. In general we cannot say anything about the left side but if *fld* is the first field in the structure pointed to by *ptr*, we know that y and *ptr* must be equal.

Other Arithmetic Operations

The result of other arithmetic operations like addition or multiplication is always treated as unknown. Consider the code fragment given in Figure 4.2. The essence here is that the *do something* part is entered with $i < 100$ and the while loop is left with a value of $i >= 100$. If we treat the result of $i + 1$ as unknown (even if the predicate in fact allows only a limited range of values) we only need two iterations to derive the aforementioned conditions on i. This example illustrates that ignoring the result of arithmetic expressions speeds up the analysis process while the results often remain roughly the same.

Propagation of Predicates

An assignment statement s cannot alter the control flow of a program. Thus, it can only have a single successor s_{next} in the control flow graph. All predicates that are added to $Post(s)$ must be propagated to $Pre(s_{next})$.

4.6.2 Goto

A **goto** statement has no effect on the data space itself, i.e. $Post(s)$ equals $Pre(s)$ for a **goto** Statement. All predicates in $Post(s)$ are propagated to $Pre(s_{jmp})$ where s_{jmp} is the statement that the control is transferred to by the **goto** statement.

4.6.3 Conditional Expression

The condition in a conditional expression s consists of a single variable that can be either true or false (see Section 2.2.6). Additionally there are two next statements in the control flow graph, namely the first statement of the true branch (s_{true}) and the first statement of the false branch (s_{false}). All predicates in $Post(s)$ will be examined and if it is clear whether the branching condition is true or false for all states in the predicate the predicate is only propagated to $Pre(s_{true})$ or $Pre(s_{false})$, respectively. Otherwise the predicate is added to both $Pre(s_{true})$ and $Pre(s_{false})$.

$Post(s)$ itself is constructed from $Pre(s)$ and the variable v that is used as the condition. If adding restrictions on v to $p \in Pre(s)$ would result in a predicate that is not multi-thread ready we insert p directly into $Post(s)$. This is always possible because the evaluation of a condition does not change the data space. The drawback is that such a predicate must be propagated down both the true and the false branch. Thus, if we can add restrictions on v we split the predicate p into two predicates q_{true} and q_{false}. The condition variable v is restricted to true in q_{true} and to false in q_{false}. Both predicates are then added to $Post(s)$. If we also make sure that the predicates are not merged right away q_{true} will only be propagated down the true branch and q_{false} only down the false branch. This way the predicates accurately reflect the outcome of the evaluation of the condition.

4.6.4 Switch Expression

A switch expression s is treated similarly to a conditional expression. The only difference is that more than two branches are possible. We know the first statement of all possible branches from the control flow analysis and we can obtain the values that belong to each branch. As a switch expression must have a default branch (either explicitly or implicitly, see Section 2.2.6), these values partition the set of all possible values of the condition variable into disjoint sets, one set for each branch. If we can add restrictions on the condition variable and still get a multi-thread ready predicate we split each predicate $p \in \mathit{Pre}(s)$ into several predicates, one for each branch, and the condition variable is restricted such that it only allows those values that belong to this branch. All of these predicates are added to $\mathit{Post}(s)$. Should any of those predicates turn out to be not multi-thread ready we add p itself to $\mathit{Post}(s)$ instead. Again we make sure that the predicates in $\mathit{Post}(s)$ are not merged.

Finally, each predicate in $\mathit{Post}(s)$ is propagated to the *Pre*-sets of those branches that are possible for the respective predicate.

4.6.5 Addresses of Global Variables

In the way our predicates are described, it is not possible to express restrictions on the contents of a global aggregate variable. In general this would not be of much use because restrictions on global variables make it difficult to prove that the predicate is multi-thread ready. But if a global aggregate variable has static fields, it should be possible to restrict the values of those fields even in multi-thread ready predicates. If we try to add such a restriction to a predicate, we need a variable that contains the same value as the field and a pointer to the structure. So far this is relatively easy to achieve in a normalized AST (see Section 2.2) because the address of the global variable is calculated and stored in a temporary variable as shown in Figure 4.3. This address is then used to access the field. The code fragment shown in Figure 4.3 also illustrates the problem with this simple approach: Although it is obvious that T.1 and T.2 contain the same value, this is not detected by the analysis. Consequently, we cannot detect that v1 and v2 are in fact equal as well.

To solve this problem, we introduce an artificial global variable that holds the address of global (e.g. _addr_of_global). Whenever we encounter an expression that takes the address of global (usually on the right hand side of an assignment), we use this variable instead of the expression. This way we can conclude that T.1 and T.2 and consequently also v1 and v2 are equal in Figure 4.3. Exactly one artificial variable for each global object is introduced. These variables have a few additional properties that can be exploited in the analysis process:

- The value of such a variable is constant because the address of a global variable cannot change. This is important because otherwise predicates that use these variables would not necessarily be multi-thread ready.
- As the address of an existing object, the value of such a variable cannot be zero.
- The values of any two of these artificial variables are different because different variables hold the addresses of different objects.

```
struct X {
        int a,b;
};
struct X global;
void f() {
        struct X * T.1, * T.2;
        int v1, v2;
        T.1 = &global;
        v1 = T.1->b;
        /* ... */
        T.2 = &global;
        v2 = T.2->b;
        if (v1 == v2) {
                /* ... */
        }
}
```

Figure 4.3: Global Variable Addresses

4.7 Analysis Algorithm for a Block

The result of the previous section can be used to analyze a block of code that does not contain function calls. Figure 4.4 shows the algorithm. The input is a block of code that must not contain function calls. We make use of the control flow graph described in Section 3.4. The result is the set of predicates $Pre(EXIT)$ that include all states that are possible immediately before the block is left. Depending on the circumstances it might also be useful to treat the resulting set P as well as $Pre(s)$ and $Post(s)$ for each statement as part of the result. After the while loop terminates it has iterated through all statements once without making changes to the *Pre*-sets and *Post*-sets. The pairs $(s, p) \in S \times P$ of a statement s and a predicate p with $p \in Pre(s)$ can now be used to construct an automaton as described in Section 4.1 that conservatively models the execution of the code block in question. There is an edge from (s_1, p_1) to (s_2, p_2) if there is an edge from s_1 to s_2 in the CFG and p_2 (or a special case of p_2) was derived from p_1 during analysis of s_1. The fact that the last iteration through the while loop did not make any changes guarantees that there is always a suitable predicate p_2 in $Pre(s_2)$. This in turn guarantees that all states that are possible before the execution of a statement s are included in at least one predicate in $Pre(s)$.

Conceptually, we can now return to the general definition of a predicate as a subset

```
$S :=$ Set of all statements in block
$P := \{[]\}$ /* Set of all Predicates */
foreach $s \in S$ {
        $Pre(s) := \emptyset$
        $Post(s) := \emptyset$
}
foreach $s_n \in CFG\_next(ENTRY)$ {
        $Pre(s_n) := \{[]\}$
}
while anything changes {
        foreach $s \in S$ {
                Try to compact $Pre(s)$
                foreach $p \in Pre(s)$ {
                        Q := Set of predicates that must be inserted
                             into $Post(s)$ for $p$ according to Section 4.5
                        foreach $q \in Q$ {
                                Try to simplify $q$
                                if $q$ equals some $q_0 \in P$ {
                                        replace $q$ with $q_0$ in $Q$
                                }
                        }
                        foreach $q \in Q$ {
                                if $q$ is a special case of $q_0 \in Post(s)$ {
                                        remove $q$ from $Q$
                                }
                        }
                        $Post(s) := Post(s) \cup Q$
                        $P := P \cup Q$
                }
                foreach $q \in Post(s)$ {
                        Propagate $q$ to $Pre(s_{next})$ where
                        $s_{next} \in CFG\_next(s)$ as described
                        in Section 4.5
                }
        }
}
Result: $Pre(EXIT)$
```

Figure 4.4: Analysis of a block of statements

```
          /* 1. {[]} */
x = 3;
          /* 2. {[x = 3]} */
z = y;
          /* 3. {[x = 3, y = z]} */
if (x == z) {
                    /* 4a. {[x = y = z = 3]} */
          x = x + 1;
                    /* 5. {[y = z = 3]} */
}
          /* 4b. {[x ≠ y, x ≠ z, x = 3, y = z]} */
          /* 6. {[x ≠ y, x ≠ z, x = 3, y = z], [y = z = 3]} */
```

Figure 4.5: Analysis of an if statement

of the set of all possible states of the data space. In this case we can define exactly one predicate $P_0(s)$ for each statement as the union of all predicates in $\mathit{Pre}(s)$:

$$P_0(s) := \bigcup_{p \in \mathit{Pre}(s)} p$$

Using these predicates we get an automaton that consists only of the states $(s, P_0(s))$ and there is an edge from $(s_1, P_0(s_1))$ to $(s_2, P_0(s_2))$ if and only if there is an edge from s_1 to s_2 in the control flow graph ($s_2 \in \mathit{CFG\_next}(s_1)$). This automaton still models the execution of the code conservatively.

Now we consider a few examples that illustrate how the algorithm works on realistic code. Section 4.8 discusses performance issues and the details of those steps that are only described informally in Figure 4.4.

4.7.1 Notational Conventions

A single predicate is always written in brackets. Restrictions made by the predicate are given inside the brackets in a compact, human readable form. This means that [] describes the trivial predicate that includes all states. The *Pre*-sets for each statement are given in the form of a comment before each statement. Most of the *Post*-sets are omitted but they can generally be deduced from the *Pre*-sets of the next statements. There is one line of comment for each time the set is changed in the progress of the algorithm. The order in which these changes happen is given by consecutive numbers. A higher numbered set of predicates always replaces a previous one. If processing of a single statement changes several sets of predicates all of these changes are labeled with the same number and an additional letter. Sometimes a modified set of predicates can be expressed in a more compact form, this is indicated by an arrow (⇒). Unless otherwise noted, all variables are local and unaddressed.

4.7.2 Example: If Statement

Figure 4.5 shows the analysis of a simple block containing an if statement. After Step 6, there are two predicates that cannot be merged.

4.7.3 Example: While Loop

Figure 4.6 shows how the analysis of the simple while loop from Figure 4.2 works. The first time the while loop is entered we still have $i = 0$ which means that the while loop is entered unconditionally. The jump to the start of the while loop after the first iteration adds the possibility that the while loop is entered with an arbitrary value of i. This means that this time we have to decide for which values of i the while loop is entered. After Steps 6a and 6b no more changes happen. Note that after the last statement in the body of a while loop, control is always transfered to the beginning of the loop where the condition is checked (Step 5).

```
        /* 1. {[]} */
i=0;
        /* 2. {[i = 0]} */
        /* 5. {[i = 0], []} ⇒ {[]} */
while (i<100) {
                /* 3. {[i = 0]} */
                /* 6a. {[i = 0], [i < 100]} ⇒ {[i < 100]} */
        i = i+1;
                /* 4. {[]} */
}
        /* 6b. {[i ≥ 100]} */
```

Figure 4.6: Analysis of a while loop

4.8 Optimizations and Performance

So far we have detailed the principles of how analysis of a block of code works. The main focus has been on the correctness of the results. In contrast, considerations concerning efficiency and feasibility of such an analysis mostly have been omitted. There are two general classes of optimizations that contribute to the overall speed of the analysis:

- Reducing the total number of predicates generally speeds up the analysis. One consequence is that we should try to remove unnecessary information from predicates. As an additional benefit, this will in turn allow us to merge predicates which further helps to reduces the total number of predicates.

- The other approach is to reduce the number of operations on predicates as much as possible and make the remaining operations more efficient.

4.8.1 Compacting of Pre-Sets

In the algorithm outlined in Figure 4.4, processing of each statement starts with an operation somewhat vaguely described as “Try to compact $Pre(s)$”. This step tries to reduce the number of predicates in $Pre(s)$ without losing information. For this purpose we first try to merge predicates that only differ in the values and bits allowed for a single variable. This operation is described in Section 4.4.3 in more detail. We only merge predicates that use exactly the same variable names in their description. Thus, we can decompose the set into disjoint groups of predicates that use exactly the same variables. As an optimization we only try to combine predicates that are in the same group. In a second step we check if there are any two predicates p and q in $Pre(s)$ such that p is a special case of q. In this case, p can be removed from $Pre(s)$ without losing information. Again the variable signature can be used to reduce the actual number of tests that need to be done.

Additionally, both merging and the special case check are only necessary if at least one of the predicates involved has been added since the last time we tried to compact $Pre(s)$. Furthermore, there is not much chance that merging and special case checks are of much help inside a single basic block. Thus, this potentially expensive step is only done at the start of a basic block.

4.8.2 Simplifying Predicates

The total number of predicates can be reduced significantly if we only keep a restriction if it will actually be used later on. Of course it is not always clear if this is the case. However, the live ranges of local variables described in Section 3.5 give an important hint. If a variable dies at a statement s, i.e. it is live before but not after s there is in most cases no point in keeping information about the value stored in that variable. Thus, we can simplify the predicate by removing all information about dying variables from the predicate. This simplifies the predicate and helps to reduce the total number of predicates. Consider the example in Figure 4.7: In Step 5 the variable t dies and we can remove it from the predicate. The result is shown to the right of the arrow. In Step 6, this simplification allows us to merge two predicates into a single one.

```
        /* 1. {[]} */
if (a != b) {
                /* 2a. {[a ≠ b]} */
        t = a;
                /* 3. {[t = a ≠ b]} */
        a = b;
                /* 4. {[t ≠ b = a]} */
        b = t;
                /* 5. {[b = t ≠ a]} ⇒ {[b ≠ a]} */
}
        /* 2b. {[a = b]} */
        /* 6. {[a = b], [a ≠ b]} ⇒ {[]} */
t = 0;
        /* 7. {[t = 0]} */
```

Figure 4.7: Live Ranges and Dying Variables

Unfortunately, it is not always helpful to apply this optimization because sometimes a dying variable can contain information that is still relevant to the analysis. Such a problem arises if the variable is in fact a placeholder for information that is actually stored elsewhere. This can happen if the variable is used in a field-of relation. Figure 4.1, that has been discussed earlier, shows a code fragment where blindly removing dying variables from a predicate can have undesired effects. Thus, we only remove dying variables from a predicate if they are not used in a field-of relation.

4.8.3 Multi-Thread Ready Predicates

A very helpful property of multi-threaded programs is that from the view of a single thread, global variables and data in the heap can change spontaneously at any point in time. Thus it is not worthwhile to add information about such data to a predicate in the first place. This is guaranteed by the exclusive use of multi-thread ready predicates. This reduces the total number of predicates significantly. Having said that, the reduction in accuracy is mostly negligible. This is also supported by the case studies that we have performed (see Chapter 7).

4.8.4 Caching

Finally, the results of special case tests and attempts to combine predicates can be cached. This is possible because both of these operations are only used on finalized predicates that can no longer change. The global list of predicates helps to increase the hit rate of the cache because *Pre*- and *Post*-sets only contain references to predicates in the global list. This gives an additional performance boost.

4.9 Summary

In this chapter we have described how a conservative analysis of a single block of code can be performed. Function calls have been postponed to the next chapter. Predicates were introduced as the main tool that is used in the process of this analysis. In order to perform the analysis we have defined a restricted class of predicates that can be stored and modified in an efficient way. Special care was taken to make sure that the results of the analysis remain valid in the presence of other control flows. The notion of multi-thread ready predicates has been defined and used to achieve this. In the next chapter we will extend the analysis algorithm in a way that supports function calls and inter procedural analysis.

Chapter 5

Global Analysis

So far we have only analyzed blocks of code but we have not taken the possibility into account that other functions might be called from this block. This chapter discusses methods to extend the analysis of a block of code to a complete program. The basic idea is that the results of the analysis of a single function are aggregated such that the actions of the function as a whole are described. The results can then be used at those places where the function is called.

5.1 Function Call Semantics

First of all, we need to have a closer look at the semantics of a function call. In a normalized AST all function parameters are passed by value, call by reference semantics can only be achieved by means of pointers. Let **void** f() be a function that does not return a value and that does not have any parameters either. Semantically, a call to such a function is almost equivalent to the execution of an inlined compound statement that contains the same code as the function body. The only actual change that needs to be made applies to return statements in the middle of the function body. These must be replaced with jumps to the end of the inlined compound statement. Obviously, the inlined version must not access local variables of the calling function that would in theory be accessible from a normal compound statement. However, this cannot happen in a syntactically correct program.

This idea can be extended to describe the semantics of a function call with parameters and return values. Let us first consider a function **void** f(p1,p2 ,..., pn), where p1,p2 ,..., pn are formal parameters. The function still does not return a value. In this case the body of the inlined compound statement declares one local variable for each formal parameter. These variables are initialized with the corresponding values of the actual parameters at the start of the block. For the purpose of our analysis such an initialization is equivalent to an ordinary assignment.

Handling of the return value can be treated in a similar way. If f returns a value, an additional artificial variable is created at the *beginning* of the compound statement. At each **return** Statement the value that is returned is assigned to this artificial variable before control is transfered to the end of the compound statement. After the execution of the body of the called function the value stored in the artificial variable is assigned to the variable that actually receives the return value. If the return value is ignored, i.e. the function is called in statement context, this final assignment step is omitted.

Finally, consider a function that has been found to have multiple return values by the procedure described in Section 3.3. In this case multiple artificial variables are created, one for each primary or secondary return value. Handling of the primary return value does not change. Each secondary return value corresponds to a pointer in the list of formal function parameters. An assignment to the memory location that is indirectly referenced by such a pointer is used to specify the value of the respective secondary return value. For the purpose of our analysis each of these assignments is replaced with an assignment to the artificial variable corresponding to the respective secondary return value. As with primary return values the contents of these artificial variables are assigned to the variables that actually receive the secondary return values immediately after the execution of the function body. Note, that the right hand side of these assignments must be treated as unknown if the aliasing situation for the variable that receives the secondary return value is unclear. For example consider a situation where a function has two secondary return values and the calling function specifies the same address for both. This means that the values given for these secondary return values in a description of the called function are not necessarily correct (see Section 3.3 for more details regarding these aliasing issues).

5.2 Function Description

In the previous section, we have described the semantics of a function call by means of assignments and control transfers. As we already know how to handle these elements of an AST in our analysis, we can directly use this description to analyze a block of code that contains a function call together with the body of the function being called. This way of handling a function call more or less corresponds to the situation where the function body of the called function was inlined into the calling function. However, we still need a way to handle function calls that cannot be inlined because of recursion or because the block of code would become too complex. The semantic description in the previous section can help with this problem as well. The core difference is that the body of the called function is described in an aggregate way.

The basic idea is that a function body is analyzed without making any assumptions

Plain Function

```
int f (int x, int y)
{
        assert (x != y);
        if (x < y)
                return x;
        return y;
}
/* ... */
_result = f(_arg1, _arg2);
```

Generic Inlined Call

```
x = _arg1;
y = _arg2;
assert (x != y);
if (x < y) {
        _result = x;
        goto end_of_block;
}
_result = y;
end_of_block:
```

Figure 5.1: Analysis of a Function

about the state of the data space at the time of the call and all states in *Pre*(*EXIT*) are used to describe the function. It is obvious that the data space must be in a state that is part of *Pre*(*EXIT*) after the call to the function. Thus we can in theory use *Pre*(*EXIT*) as the set of predicates that are possible after the call to the function in question. However, this approach does not allow for any interaction with the analysis of the calling function because the predicates that are possible after the call do not even depend on the predicates that are possible before the call. Furthermore, most if not all variables that can be used in predicates in *Pre*(*EXIT*) are local to the called function and thus die before the *EXIT* block is reached. This means that it is very likely that *Pre*(*EXIT*) only consists of the trivial predicate [].

This problem is solved by making those variables that are used to interact with the called function available during analysis of that function in a generic way. In order to achieve this we do not only analyze the body of a function but a basic block that consists only of a single call to the function. The call uses generic variable names for the actual parameters and the return value. Analysis of the function call in the block is performed by inlining the function body as described in the previous section. Consider the example in Figure 5.1. On the left we can see the function definition of a simple function that returns the minimum of its two parameters x and y (provided that a unique minimum exists) and a generic call to that function. The code block on the right is the result if the generic call to the function on the left is put into a block of its own and the function body is inlined. Analysis of this block now progresses as shown in Figure 5.2. The final result given in Step 7 is a combination of the predicates obtained in Steps 6a and 6b. From this analysis we can conclude that if the function returns, the actual arguments must have been unequal. We can also see that the return value equals one of the arguments. In general we can say that the data space must be in one of the states described by the predicates obtained

```
        /* 1. {[]} */
x = _arg1;
        /* 2. {[x = _arg1]} */
y = _arg2;
        /* 3. {[x = _arg1, y = _arg2]} */
assert (x != y);
        /* 4. {[x = _arg1 ≠ y = _arg2]} */
if (x < y) {
                /* 5a. {[x = _arg1 ≠ y = _arg2]} */
        _result = x;
                /* 6a. {[_result = x = _arg1 ≠ y = _arg2]} */
                /*     ⇒ {[_result = _arg1 ≠ _arg2]} */
        goto end_of_block;
}
        /* 5b. {[x = _arg1 ≠ y = _arg2]} */
_result = y;
        /* 6b. {[x = _arg1 ≠ _result = y = _arg2]} */
        /*     ⇒ {[_arg1 ≠ _result = _arg2]} */
end_of_block:
        /* 7. {[_arg1 ≠ _result = _arg2], [_result = _arg1 ≠ _arg2]} */
```

Figure 5.2: Analysis of the block in Figure 5.1

in Step 7 after the function call. In general, we use the predicates obtained at the end of the inlined function block as an aggregate description of the function. The generic variables for the arguments and the return values are placeholders for the actual variables. Note, that a generic argument can be interpreted as the initial value assigned to the corresponding formal argument at the time the function is called.

5.2.1 Initial States

The example in Figure 5.2 illustrates that the description obtained for a function usually includes those effects of a function that are likely to be important for the analysis of the calling function. In Figure 5.2 this is the fact that the function does not return if both arguments are equal and that the return value equals one of the arguments. The analysis of a code block tries to distinguish states where this is likely to be useful later on and in cases where this is not likely the relevant predicates are joined when possible. Thus, if the function has no pre-conditions it is often possible to derive meaningful post-conditions of a function automatically without human help.

However, there are cases where the function description obtained from the analysis described above is not as precise as it could be. This is due to asymmetries in the handling of statements depending on the information that is known about a certain variable. The most common case is a type cast: If it is known that a variable that is cast to another type contains only values that can be represented in both types (i.e. the cast has no effect) the original variable and the result of the cast are assumed to be equal. However, if nothing is known about the value that is cast, the result of the cast is also unknown.

E.g. consider a function to_unsigned(**short** x) that returns the value of its parameter x cast to **unsigned int**. The aggregate description of such a function would be the empty predicate, i.e. nothing is known about the return value or the parameters. However, if this function is called with a constant value of e.g. 1 or with a variable that is known to be between 0 and 32767 we would like to conclude that the return value equals the parameter and the aggregate function description should reflect that.

This can be achieved if we start the analysis of the function with several predicates that together include all states of the data space. E.g. for the to_unsigned function we could start with $\{[0 \leq \_\mathit{arg1} \leq 32767], [\_\mathit{arg1} \in \{-\infty \ldots -1, 32768 \ldots \infty\}]\}$ where _arg1 is the generic argument that corresponds to the formal parameter x. As both predicates together describe all states of the data space this is equivalent to an initial predicate of [].

In fact, the analysis would merge the two predicates as soon as possible. This is prevented by an additional dummy variable that does not exist in the program. It is present in all initial predicates and has different values in each of them. Throughout the analysis of the function, the value of the dummy variable indicates the initial predicate which a specific predicate is derived from. Thus, analysis of a function with several initial predicates progresses as if each case was considered separately. Of course, the dummy variables are removed from the final function description. In the example of the to_unsigned function this would result in two final states $\{[\_\mathit{result} = \_\mathit{arg1} \in \{0 \ldots 32767\}], [\_\mathit{arg1} \in \{-\infty \ldots -1, 32768 \ldots \infty\}]\}$. This reflects that the return value equals the only argument if the argument is known to be between 0 and 32767.

The initial hint as to which cases need to be distinguished must be specified manually. However, this is only necessary for a very small amount of functions. For most functions the standard analysis is able to derive suitable and sufficiently precise function descriptions.

5.2.2 Pre-Conditions

Some functions just cannot be called with arbitrary values for their arguments. Ideally these restrictions are checked by means of assertions in the function itself. If this is the case, the pre-conditions for the function arguments can usually be derived automatically from the function itself. The function f shown in Figure 5.1 is an example of such a function: The pre-condition that the arguments must not be equal is checked by an assertion and the aggregate description reflects this.

If a pre-condition is not checked by an assertion, the function will usually yield incorrect or unexpected results. In the analysis we always have the option of ignoring the pre-condition. If the function is actually called and the pre-condition is not true our analysis will accurately reflect the effects of such a call. Naturally this can lead to problems in the analysis of the function itself or later on in a calling function that violates the pre-condition.

The other option is to specify pre-conditions explicitly. This is done similarly to the way we gave hints for the analysis in the previous section: We simply do not start analysis with the trivial predicate [] but with a more restrictive one that reflects the pre-condition. The aggregate result will then reflect the effects of the function *provided that the pre-condition is true* at the time of the call. Obviously such a pre-condition must be checked each time the function is actually called. If the pre-condition cannot be verified, this leads to an error.

Note that pre-conditions are limited to a single predicate. This restriction is not strictly necessary but it is usually sufficient and greatly simplifies the checking of pre-conditions.

5.3 Function Calls

In the previous section we have discussed how to obtain an aggregate description of a function. In this section we see how such a description can be used during analysis of the calling function. First of all, we take the function description and replace all generic variable names that were used for the arguments with the corresponding names of the variables that are actually used in the call. To ensure that all predicates are still multi-thread ready, we only replace a generic variable with a real variable that is unaddressed and local to the calling function. All generic variables for arguments that could not be replaced as a consequence of this restriction are removed entirely from the predicate. For the time being, the variables that hold the primary and potentially any secondary return values are left unchanged.

Now let us look at the point in the caller immediately after a call to a function but before the return value, if any, is evaluated by the caller. At this point we know that the data space must be in one of the states that are described by the predicates in the modified function description. The remaining artificial variables are placeholders for the primary and secondary return values. Sometimes, the fact that the call to a function returns, already gives information about the actual parameters. Additionally, the values of the parameters naturally influence the primary and secondary return values. This information is properly reflected by the predicates in the modified function description. However, anything that was known before the call to the function is lost although many properties that were known before cannot be changed by the function call. In fact, restrictions that only involve local unaddressed variables and static structure fields (see also Section 4.1.2) cannot be changed by a function call for the simple fact that the called function cannot access these variables. The only exception affects variables that receive one of the secondary return values. Thus if we remove all restrictions that do not fall into this category from the predicates in the *Pre*-set of the call, we know that the state of the data space immediately after the call must be in one of these predicates. Thus we have two different sets of predicates

- the predicates derived from the function description by replacing generic variable names with the names of the actual parameters and
- the predicates derived from the *Pre*-set by removing restrictions on variables that can potentially be modified by the function call.

We know that the state of the data space immediately after the call must be in both of theses sets. This is a reasonable description of the states that are possible immediately after the call but it is not yet suitable for use in the analysis algorithm (see Figure 4.4). For this purpose we want to describe the same total set of states by a single set of predicates. To do this, we build all pairs of predicates where one is in the first and the other is in the second set. The intersection of the predicates in a single pair can be calculated by the intersection operation described in Section 4.4.4. The union of the predicates resulting from the intersection of the predicates in a pair contains all states that are possible immediately after the call. Thus the set of these predicates can be used as the *Post*-set immediately after the call.

Actually this process can be simplified by means of the mapped intersection described in Section 4.4.4. This way there is no need to translate the function description first. Instead we construct a map that maps the generic variables for the arguments to their actual counterparts. If the intersection operation above is replaced with a mapped intersection that uses this map we can use the predicates of the real function description directly without modifying them first.

Of course, each of the intersections can be empty. If this is detected, the predicate is simply ignored. This way predicates in the function description are effectively excluded if they describe cases that are not applicable to the current call.

Handling of return values can be done by assigning the generic variables that hold the return values to their respective counterparts that actually receive the return values. With regard to the predicates this can be done in exactly the same way as it is done for normal assignments. Finally, the remaining generic variables can be removed from all predicates.

5.3.1 Checking of Function Call Pre-Conditions

In the cases where a pre-condition has been specified explicitly this condition must be checked before the function call. Otherwise the function description does not describe the function accurately. As pre-conditions are limited to restrictions that can be expressed in a single predicate, it is relatively easy to check them. As always we start by replacing generic variables for function arguments with the real arguments in the pre-condition. The resulting predicate contains all states of the data space that are allowed immediately before the function call. Thus, all predicates in the *Pre*-set of the call must be special cases of the pre-condition. This can be checked by the normal special case operation for predicates. If the check fails this must be treated as an error. However, if the analysis fails to verify the pre-condition but manual inspection shows that the pre-condition is in fact always true, such an error can be ignored or turned into a warning.

5.3.2 Calculated Calls

So far, we have assumed that the function being called is given by name, i.e. we know which function is called and as a consequence we also know where to look for its description. In case of a calculated call, the situation is a bit more complicated.

From the function pointer analysis described in Section 3.6, we already know which functions can possibly be called at this point. With this knowledge, the considerations for normal calls above apply analogously to the case of a calculated call. The only difference is that we must consider all function that can potentially be called. For each possible function we get a set of predicates that is possible after the call and the union of these sets from all functions gives the set of predicates that are possible after the call.

5.4 Example

Let us consider the function f shown in Figure 5.3. Analysis of this function yields the description also given in Figure 5.3. Please note that the assertion and the comparison of x and y affect the variables for the generic arguments whereas the assignment to x does not. This is desired because the first two operations actually tell us something

```
int f (int x, int y) {
                    1. ${[x = \_arg1, y = \_arg2]}$
          assert (x > 0);
                    2. ${[x = \_arg1 \geq 1, y = \_arg2]}$
          if (x == y) {
                              3a. ${[x = y = \_arg1 = \_arg2 \geq 1]}$
                    return 1;
                              4. ${[\_arg1 = \_arg2 \geq 1, \_result = 1]}$
          }
                    3b. ${[y = \_arg2 \neq x = \_arg1 \geq 1]}$
          x = y;
                    5. ${[x = y = \_arg2 \neq \_arg1 \geq 1]}$
          return 2;
                    6. ${[\_arg2 \neq \_arg1 \geq 1, \_result = 2]}$
}
```

Function description (results of Steps 4 and 6):

$$\{[\_arg1 = \_arg2 \geq 1, \_result = 1], [\_arg2 \neq \_arg1 \geq 1, \_result = 2]\}$$

Figure 5.3: Calculation of a function description

about the actual arguments while the third does not. This might seem a bit surprising at first because a function cannot change its actual arguments in the caller. However, the return value and in case of an assertion the fact that the function returns at all can give us information about the values that are passed to the function. This is what the function description reflects.

Now let us look at a call x=f(a,b) to this function. Interesting things only happen if the *Pre*-set of the call contains restrictions on the variables a and b. Potential restrictions on x are irrelevant because x is not used in the function arguments and with the assignment of the return value to x these restrictions are removed anyway. Thus let us assume that the *Pre*-set of the call is

$$\{[5 \leq a = b \leq 7], [a \leq 4]\}.$$

After replacing _arg1 and _arg2 in the function description of f the result looks like this:

$$\{[a = b \geq 1, \_result = 1], [b \neq a \geq 1, \_result = 2]\}$$

From these two sets we can build four pairs of predicates. The following table shows these pairs and their intersections:

Pre-set	**Function Description**	**Intersection**
$[5 \leq a = b \leq 7]$	$[a = b \geq 1, \_result = 1]$	$[5 \leq a = b \leq 7, \_result = 1]$
$[5 \leq a = b \leq 7]$	$[b \neq a \geq 1, \_result = 2]$	$\emptyset$
$[a \leq 4]$	$[a = b \geq 1, \_result = 1]$	$[1 \leq a = b \leq 4, \_result = 1]$
$[a \leq 4]$	$[b \neq a \geq 1, \_result = 2]$	$[1 \leq a \leq 4, a \neq b, \_result = 2]$

The combination in the second row results in the empty set. This is an example where the predicate from the *Pre*-set and the predicate from the function description are incompatible. The results in the first and the third row can be combined to

$$\{[1 \leq a = b \leq 7, \_result = 1]\}.$$

Finally _result is assigned to x and _result is removed which results in the following *Post*-set for the function call:

$$\{[1 \leq a = b \leq 7, x = 1], [1 \leq a \leq 4, a \neq b, x = 2]\}$$

This result is not immediately obvious but we have seen that it can be derived from the function and the conditions that are true before the function in an efficient and deterministic way.

5.5 Recursive Calls

The function descriptions and their use described in the previous sections allow us to extend the analysis algorithm from Figure 4.4 to functions that call other functions as long as a description of the called function is known. Thus as long as the call graph of a program is acyclic it is possible to process all functions bottom up in topological order. This way there is always a function description available for each function that is called.

If the call graph has cycles this method does not work. In this case we must break the cycles by providing initial, usually unverified function descriptions for some functions explicitly. Fortunately, the trivial predicate [] is always a valid function description. Such a function description simply means that nothing is known about the effects of the function. With this knowledge we can again try to process the call graph bottom up in topological order and calculate the function description for functions until a cycle is detected. The cycle is broken by giving one or more functions in the cycle the default description []. All incoming edges from functions that this function is called from are removed. All of the function descriptions obtained in this way are correct under conservative assumptions. However, some descriptions might not be as precise as they could be. Note, that functions that are given a default function description initially will obtain a real function description once all other functions that they call are processed. The latter description can be more restrictive and thus more precise than the initial default description. These more precise descriptions in turn might allow more precise descriptions of other functions. To exploit this fact it is useful to iteratively run this process several times. Ideally, this iteration is repeated until no function description changes in a pass. Unfortunately, there are two problems:

- It is not necessarily possible to decide efficiently, if two function descriptions are equal. This is especially true if the function description consists of multiple predicates but we have also seen cases where two predicates can be equal in a non obvious way.
- The process is not guaranteed to converge. In fact there are cases where the function description of a function alternates between several representations that are equal in a non obvious way.

Thus it is useful to abort iteration after a relatively small number of passes.

5.5.1 Preprocessing of the Call Graph

Consider a directed graph where each node represents a function. If a function f contains a call to another function g there is a directed edge from the node representing f to the node representing g. This graph is said to be the *call graph* of a program[45][7]. It might be tempting to ask if it is possible to analyze the call graph and choose the set of functions that start with a trivial function description optimally, prior to the calculation of other function descriptions. In this section we briefly describe the problems that arise in this context. We will see that the problems turn out to be NP-complete, i.e. we cannot expect to find efficient algorithms that solve them optimally.

Maximum Acyclic Subgraph

We want to remove as few nodes as possible from this graph such that the resulting graph has a certain property P. Thus the result is the largest subgraph that has property P. In our case the property is that the graph can be sorted topologically, i.e. it does not contain cycles. This problem is known as the *minimum node-deletion problem* and it is known to be NP-complete for a wide range of properties P. In [31] it is shown that the minimum node deletion problem is NP-complete for all properties P that are

nontrivial, i.e. there are infinitely many graphs for which P is true and infinitely many graphs for which P is false, and

hereditary, i.e. if P is true for a graph it must also be true for all vertex induced subgraphs.

This result holds for directed as well as undirected graphs.

In our case the property P is *acyclic* which clearly fulfills both conditions given above. Thus our instance of the minimum node-deletion problem is NP-complete. Similar problems where edges of the graph are deleted instead of nodes are also known to be NP-complete[5].

Optimal Vertex Ordering

Another problem significantly affects performance: As functions are processed in topological order, it is likely that after a function has been processed, the next function definition is not in the same translation unit. This is suboptimal because changing to a different translation unit requires loading of its entire AST into memory. This can be expensive and should be avoided. Thus we are faced with the following special case of a vertex ordering problem: We are given the call graph with some nodes removed such that the remaining graph is directed and acyclic. Additionally a partitioning of the nodes is given by the translation units, i.e. two nodes belong to the same partition if and only if the functions they represent are defined in the same translation unit. This partitioning can also be interpreted as a coloring of the graph nodes[1] and we will refer to the partition that a node belongs to as the node's color where convenient.

The edges of the graph define a partial ordering of the nodes and we want to find a total ordering that is consistent with this partial ordering such that the number of pairs of adjacent nodes in the total ordering belonging to the same partition is maximized. Note that the reverse total ordering corresponds to the order in which the functions are processed. The requirement that this ordering is consistent with the partial ordering ensures that a function is only processed after all the functions that it calls are processed.

We show that this problem is NP-complete. First of all this problem is obviously in NP. To show that the problem is NP-complete we will show that an efficient solution to our problem would lead to an efficient solution to the well-known NP-complete Shortest Common Supersequence (SCS) problem over a binary alphabet. In an SCS Problem we are given k sequences $S_1, S_2, \ldots, S_k$. Let $|S_i|$ denote the length of sequence S_i and $N := |S_1| + |S_2| + \ldots + |S_k|$. The Shortest Common Supersequence problem is to find a shortest sequence S that contains all the sequences $S_i, i = 1 \ldots k$ as subsequences, i.e. S_i can be obtained from S by removing some characters from S. This problem is NP-complete even under various restrictions on the alphabet or

---

[1] Note, that in this context a coloring of the nodes is simply a partitioning, i.e. there is no restriction on the colors of adjacent nodes.

the structure of the sequences (see [43], [32], [38]). E.g. in [43] it is shown that the problem is NP-complete for a binary alphabet, i.e. the sequences only contain letters from $\{0, 1\}$. This is the result we will rely on.

First we consider a SCS problem over an arbitrary alphabet. If we add a single letter to this alphabet and at the same time add this letter at an arbitrary point in exactly one of the sequences, we see that the length of the SCS increases by exactly one. Thus, if we can solve the transformed problem we can also solve the original problem by removing the additional character from the supersequence. Now, we consider an arbitrary SCS problem over a binary alphabet, i.e. there are k sequences $S_1, S_2, \ldots, S_k$ consisting of letters from the alphabet $\{0, 1\}$ and show that our vertex ordering problem can be used to solve this SCS problem. For this purpose, we apply the transformation described above several times until none of the sequences contains adjacent letters that are equal (this increases the size of the alphabet). This is done by repeatedly inserting a separator between two such letters. The separator is a unique letter, i.e. it differs from all other letters in all sequences. At most $N - 1$ of these transformations are needed to obtain sequences that do not contain adjacent letters that are equal. With each transformation the size of the alphabet and the total number of letters in all sequences increases by exactly one. Thus, after the transformation, the sequences contain at most $2N-1$ letters and the alphabet contains at most $N + 1$ letters. Let $\hat{S}_1, \hat{S}_2, \ldots, \hat{S}_k$ be the sequences after all separators are inserted and $\hat{N} := |\hat{S}_1| + |\hat{S}_2| + \ldots + |\hat{S}_k|$. If we can solve this problem efficiently, we can also solve the original SCS problem efficiently.

We will now show that our vertex ordering problem can be used to solve all SCS problems where the sequences do not contain adjacent letters that are equal. This shows that the vertex ordering problem can solve the general SCS problem over a binary alphabet as well, i.e. the given vertex ordering problem is NP-complete. The idea is that each individual sequence is represented by a linear subgraph that contains one node for each letter in the sequence. Two nodes belong to the same partition if and only if they represent the same letter. Figure 5.4 illustrates both transformations.

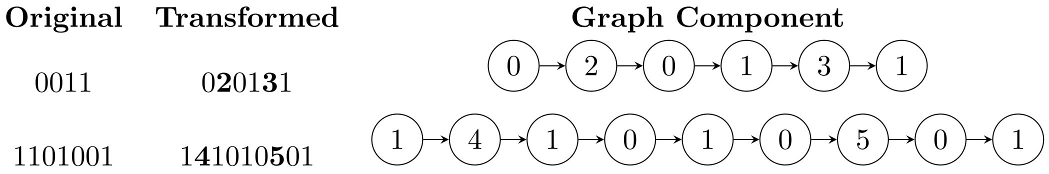

Figure 5.4: SCS problem transformation

Let $V = (v_1, v_2, \ldots, v_{\hat{N}})$ be a solution to our vertex ordering problem. Let us further group the nodes in this list into maximal groups of adjacent nodes that have the same color. For each group of nodes write down the letter that corresponds to the color

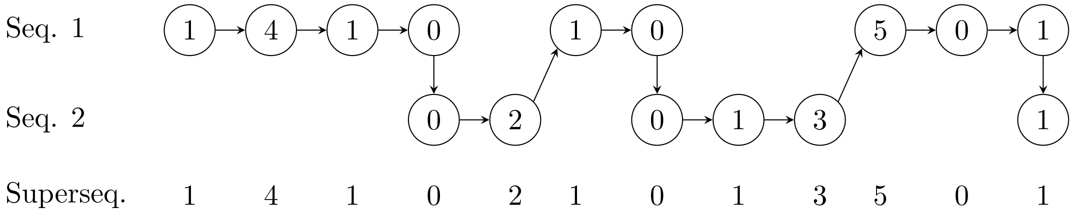

Figure 5.5: Total ordering and supersequence for the graph in Figure 5.4

of the nodes in the group to obtain a sequence $\hat{S}$. Figure 5.5 illustrates this for the sequences from Figure 5.4. The arrows describe the total ordering and the last line shows the resulting sequence $\hat{S}$. Note that the number of adjacent pairs of nodes with different colors in the total ordering is exactly one less than the length of this sequence. We claim that the sequence $\hat{S}$ derived from a total ordering is a SCS of the sequences $\hat{S}_i$ provided that the total ordering is a minimal solution to our vertex ordering problem

We first show that $\hat{S}$ is a supersequence of each $\hat{S}_i$: Each $\hat{S}_i$ corresponds to a sequence $v_{p_1}, v_{p_2}, \ldots, v_{p_r}$ $(r = |\hat{S}_i|)$ of vertices in the graph that form a linear subgraph. As the total ordering is consistent with the partial ordering induced by the edges in the graph, the v_{p_j} form a subsequence of the sequence that defines the total ordering of the vertices. From the fact that two adjacent letters in $\hat{S}_i$ are different we can conclude that v_{p_j} and $v_{p_{j+1}}$ have different colors. Thus, they cannot be in the same group of vertices which implies that each v_{p_j} corresponds to a letter in $\hat{S}$. Hence, $\hat{S}$ is a supersequence of $\hat{S}_i$ for $i = 1, \ldots, k$.

For the sake of contradiction, let us assume that there exists a supersequence $\bar{S}$ that is shorter than $\hat{S}$. We show that this implies that there exists a total ordering of the vertices that is better than the supposedly optimal ordering V. A total ordering $\bar{V}$ is constructed as follows: For each letter in $\bar{S}$ all nodes that have the color given by the letter and that do not have an incoming edge are added to the end of $\bar{V}$ in an arbitrary order and removed from the graph. This results in exactly one group of vertices with the same color for each letter in $\bar{S}$. Additionally, after all letters in $\bar{S}$ are processed, the remaining graph is empty because each linear subgraph corresponds to a subsequence of $\bar{S}$. It is also clear that the resulting sequence $\bar{V}$ is a total ordering that is consistent with the partial order induced by the graph edges. Thus $\bar{V}$ consists of $|\bar{S}| < |\hat{S}|$ groups of adjacent vertices with the same color. This implies that $\bar{V}$ is a better vertex ordering than $\hat{V}$ in contradiction to the assumption that $\hat{V}$ was already optimal. Therefore $\hat{S}$ is in fact a shortest common supersequence.

This completes the proof that our vertex ordering problem can be used to solve the SCS problem over a binary alphabet and is thus NP-complete.

An Improved Heuristic

In the previous sections we have seen that we cannot expect to find an efficient algorithm to determine the optimal order in which functions are processed. This observation relies on the assumption that the graph theoretical problems discussed above accurately model the actual problem. In theory it is possible that special properties that can be expected from a real life call graph give rise to an efficient algorithm for one or both of the problems discussed in the previous sections. However, given the wide range of NP-complete problems that have a similar structure this is very unlikely.

Nonetheless, it is still reasonable to analyze the call graph to determine the order in which functions are processed using a few simple heuristics.

- The call graph is processed bottom up, i.e. functions that do not contain calls at all are analyzed first. A function that does not have any dependencies (outgoing edges in the call graph) is chosen and removed from the graph. If it is possible to choose a function that is in the same translation unit as the previous function such a function is preferred.

- If all function have dependencies the call graph is decomposed into strongly connected components. These components form a graph that is directed and acyclic. Only functions that are in a component that does not have dependencies in the component graph are considered as the next function.

- Among these functions, we prefer functions with high incoming or outgoing degree. This increases the chance that removing the function from the graph splits its strongly connected component into one or more parts.

Reasonable results can also be obtained if the decomposition into strongly connected components is done only once at the beginning. This may be desirable because the calculation of strongly connected components takes up to $O(|V|^2)$ time where $|V|$ is the number of vertices in the graph.

5.6 Summary

In this chapter we have extended the analysis algorithm from Chapter 4 such that it is also suitable for functions that call other functions. This was achieved by the introduction of aggregate function descriptions. We have shown how function descriptions can be derived from the function's AST and how these function descriptions are used during analysis of the calling function. This technique has also been illustrated by an example. Finally, we have shown that the problem of finding an optimal order in which functions are analyzed is an NP-complete problem and we have given a heuristic that allows us to solve this problem approximately.

Chapter 6

Analyzing Synchronization

In the previous chapters, we have proposed methods to analyze large multi-threaded programs. In this process we have given an algorithm that allows us to draw conclusions about possible states of the data space at each point in the program. This implicitly gives us information about possible paths through a program. In this chapter we show how this information can be exploited to analyze problems related to synchronization.

6.1 Synchronization Primitives

In this section we have a closer look at some commonly used synchronization primitives. This list is not intended to be exhaustive or complete but the common cases are included. Section 6.1.2 introduces a generalized model that covers a wide range of practically relevant cases.

6.1.1 Mutex Locks

As synchronization did not play a role in the analysis described so far we first have to identify the primitive operations associated with synchronization. Probably the most simple synchronization primitive is a mutual exclusion lock usually known as a mutex. This type of lock is a boolean value that is true if the resource protected by the lock is currently in use and false if the resource is free. Additionally, two primitive operations are supported:

lock Atomically test if the boolean value is currently false and set it to true if this is the case. The lock operation can fail if the boolean is already set to true. The return value indicates if the lock operation was successful.

unlock The unlock operation unconditionally sets the boolean to false. The unlock function must not be called if the lock is already false. Depending on the implementation, this condition may or may not be checked.

The boolean value itself is passed to these basic operations by reference. It is important to note that these primitives implicitly require that *lock* and *unlock* operations are paired and an *unlock* operation is performed by the same control flow that initiated the corresponding *lock* operation.

6.1.2 A Generic Lock Model

We use a slightly more generalized model than the simple mutex locks described above.

- The data structure that represents the lock consists of a counter. This counter can have values between 0 (unlocked) and some upper limit that depends on the actual synchronization primitive being modeled. In case of a simple mutex lock this upper limit would be 1 which corresponds to *true* while 0 corresponds to *false*.

- A *lock* operation atomically adds one to the counter if this is possible without violating the upper limit. Otherwise the lock operation fails. Failure is indicated by the return value.

- An *unlock* operation atomically decreases the value of the counter by one. It is not allowed if the counter is already zero.

This model clearly includes the mutex case described above. For the purposes of our analysis we assume that lock and unlock operations must be balanced within each control flow, i.e. if a control flow acquires a resource it is this control flow that will release the resource later on.

Blocking on a Lock Operation

It is often useful to have a lock operation that does not fail. Instead the process performing the lock operation is *suspended* until the lock counter can be incremented without failure. Such a function can easily be implemented by means of the simple lock operation described above. To achieve this, the simple lock operation is run in a loop until it returns success. Depending on the precise meaning of *suspend* in the description above the infinite loop can be a busy loop or a loop that allows other processes to run until the lock becomes available. In the latter case, an unlock operation must wake up threads that are waiting for the lock to become available.

6.1.3 Semaphores

Another well known synchronization primitive are Dijkstra's semaphores introduced in [15]. A semaphore is at first sight very similar to our lock model described in the previous section. Conceptually, it consists of a counter that is initialized to a non-negative value N (usually the number of identical resources initially available) and two atomic operations:

- The P operation, that roughly corresponds to *lock*, waits until the value of the semaphore is positive and atomically decreases the value by one.
- The V operation, that corresponds to *unlock*, unconditionally increases the value of the semaphore by one.

Compared to our lock model, we immediately notice a trivial difference: For corresponding operations, the changes to the counter are in the opposite direction, i.e. a P operation that logically corresponds to *lock* decreases the counter while in our model *lock* increases the counter. However, the major difference is not so obvious: Semaphores do not require that P and V operations are performed by the same thread. In fact one classical application of a semaphore is waiting for some event that is signaled by another thread. In this case a semaphore is created for this event and the counter is initialized to 0. The waiter performs a P operation and is suspended because the counter must not be decreased below 0. Another thread can use a V operation to signal that the event occurred. This wakes up the waiter that is blocked in the middle of a P operation and as the V operations increased the counter to 1 the P operation of the waiter succeeds in decrementing the counter. The reason why semaphores are well suited for this kind of application is that their semantics do not depend on the temporal sequence of events. In fact if the V operation is performed before the P operation the latter simply succeeds immediately as desired.

This use of semaphores is not covered by our simple counter-based lock model. However, in many practical applications the actual use of semaphores adheres to the additional restriction required by the simple model that P and V operations are balanced in a single control flow. Semaphores used in this way can be analyzed with the methods that we develop for the simplified lock model.

6.1.4 Read/Write Locks

Read/Write locks are similar to mutex locks but are used for resources that are frequently accessed in a read only manner. This kind of lock supports two different *lock* operations and their corresponding unlock operations:

- *writelock* and *writeunlock* for accesses that actually want to modify the resource and

- *readlock* and *readunlock* for accesses where the resource is accessed without modifying it.

On their own, *writelock* and *writeunlock* behave like a mutex lock, i.e. exactly one thread can successfully call *writelock* and all other calls to *writelock* will fail until the initial thread calls *writeunlock*. Conversely, *readlock* operations alone do not fail at all, i.e. all *readlock* operations, even recursive ones from the same thread, succeed unless some thread is holding a write lock (*readlock* and *readunlock* operations must still be balanced within each thread). The interesting property of Read/Write locks is the interaction between *readlock* and *writelock*: Any active *readlock* causes a *writelock* operation to fail while an active *writelock* prevents all threads from acquiring a *readlock*. In other words, only one type of lock can be held at any point in time, however, *readlock*s are non exclusive whereas *writelock*s are exclusive.

Read/Write locks cannot be modeled directly in the generic lock model. However, if we add an artificial limit to the number of *readlock*s that can be held simultaneously, it is possible to model a Read/Write lock as follows:

- Initially, the lock counter is set to 0 and the maximum value of the lock counter is limited to the number N of *readlock*s that can be held simultaneously.
- A *readlock* operation atomically increases the counter by one.
- A *writelock* operation tries to increase the counter N times. If any of these operations fails, the whole *writelock* operation fails and successful counter increments done by a failed *writelock* operation are reversed.

Note that from a practical point of view this is not a reasonable way to implement Read/Write locks. One of the reasons is that readers can completely starve a writer, i.e. if there are lots of threads that acquire read locks it is likely that all *writelock* operations fail. Thus, real implementations usually have a way to prevent readers from acquiring a new read lock once a *writelock* operation is pending. From an analysis point of view, we only need to distinguish two cases: Either a lock operation succeeds or it fails and the semantics of both cases are accurately modeled by the implementation given above.

6.1.5 Lock-Free Synchronization

Recently, there has been a lot of research on synchronization schemes that do not rely on classical synchronization primitives like locks or semaphores. One of the most notable efforts in this regard is the Read-Copy-Update paradigm (see e.g. [33] and [34]). Lock-Free algorithms usually solve very specific problems, e.g. the Read-Copy-Update paradigm is optimized for those cases where read accesses to a shared resource are much more frequent than write accesses. Additionally, they tend to rely on specific machine instructions like memory barriers or a compare and swap instruction. Often, specific properties of the underlying hardware, especially those of memory caches, need to be considered, too. In the following, we do not discuss this kind of synchronization mechanisms further.

6.2 Identifying Equal Locks

In the remainder of this chapter we only consider lock primitives that follow the generic model described in Section 6.1.2. Thus, a specific instance of such a lock is a data structure that contains the lock counter. Moreover, the structure might contain additional information which is necessary internally for the implementation of lock primitives, e.g. a list of threads that are currently waiting for the lock.

Now if we want to analyze synchronization mechanisms we need a way to determine if two instances of a lock actually refer to the same lock. If pointers are involved, this is in general a difficult problem. However, in practice, the address of a lock is rarely used for other purposes than to obtain a reference to the lock structure that is passed to one of the lock primitives. Conversely, functions that take a pointer to a lock structure as a formal parameter are treated as extended lock primitives whose effect on the lock must be described explicitly.

Now let us look at the ways a lock can be created. As with any other data structure only a limited number of possibilities exists:

- A variable of lock type is explicitly declared either globally or locally. The latter case does not make much sense in the context of a lock because a lock that is local to a function cannot be accessed by other threads. An exception are locks that are declared with the `static` keyword in C but these behave exactly like global variables and can be treated as such.
- Memory for a lock data structure is allocated dynamically.
- Memory for a data structure that includes a lock data structure as a field is declared globally or allocated dynamically.

With this knowledge we can classify all locks into three categories:

Global Locks This group includes all locks that are declared as a global variable. Locks that are declared local to a function with the `static` keyword in C also belong to this group.

Embedded Locks This group includes all locks that are created together with a surrounding structure.

Anonymous Locks This group includes all other locks, i.e. locks that are allocated dynamically or that are declared local to a function.

With this classification we can conclude that two locks can only be equal if they fall into the same group. This follows from the fact that for two locks to be identical their allocation method must be the same.

For global locks, the situation is simple: Two global locks are equal if and only if they share the same declaration. For two anonymous locks we have to resort to their addresses, i.e. two anonymous locks are equal if their respective memory addresses are equal. As anonymous locks are very rare in practice this is not a big problem.

An embedded lock is uniquely defined by the declaration of the structure field that contains the lock and a pointer to the surrounding structure. From the provisions made in Section 2.4.3 we can conclude that two embedded locks can only be equal if they share the same structure field declaration. Two embedded locks that do share the same structure field declaration are in fact equal if and only if the respective pointers to the surrounding structures are also equal. This allows us to further divide all locks into classes as follows:

Classes for global locks One *separate* class for each declaration of a global lock. Each of these classes contains only a single lock.

Classes for embedded locks One class for each declaration of a lock as a structure field. Each of these classes contains those embedded locks that use the respective structure field.

Class for anonymous locks All anonymous locks share a *single* class of their own. It is expected that this class is empty or contains only very few elements.

With this classification, we can say that two locks are identical if the following conditions are met:

- Both locks must belong to the same class.
- For global locks being in the same class is sufficient.
- Embedded locks are equal if the pointers to the surrounding structure are equal, too.
- Anonymous locks are equal if their memory addresses are equal.

Note that in the latter two cases it might not always be possible to actually decide if two locks are equal.

6.3 Locks and Predicates

In this section we show how the generic analysis algorithm can be extended to include information about locks. For the time being, we will make two assumptions:

- The class of anonymous locks is empty.
- There are no pointers to locks.

Both assumptions are nearly true in the systems we studied in the case studies in Chapter 7. Methods for handling a limited number of anonymous locks and pointers to locks are investigated later.

6.3.1 Extended Predicates

In order to analyze the behavior of locks, we need to keep track of locks that are used within the body of a function. To achieve this, we add an additional section with lock information to the description of predicates. For each lock we need a way to identify the lock within the predicate. Moreover, an aggregate description of the operations (*lock* and *unlock*) applied to each lock is associated with each of those locks in the description of a predicate.

A lock is identified by its class and if the class contains more than one lock (which is only possible for embedded locks) individual locks in each class are distinguished by a pointer to the surrounding structure. If necessary a variable containing the pointer must be added to the predicate. This way it is possible to use existing operations on the predicate in order to determine if two locks are actually equal. From the classification of locks described in Section 6.2 we know that two locks can only be equal if they are in the same class. For global locks this is sufficient to conclude that two locks are in fact equal. Given two embedded locks that fall into the same class there must be two variables p_1 and p_2 that are mentioned in the predicate such that p_1 and p_2 point to structures containing the first and the second lock, respectively. Thus, we can use the existing predicate operation that tests for equality of two variables to determine if p_1 and p_2 are equal. The result of this test extends to the locks themselves, i.e. the two locks are equal if p_1 equals p_2 and they are different if p_1 is known to be different from p_2.

There are two things that can be problematic in this context:

- A variable is used as the pointer to the structure containing an embedded lock but we cannot prove that the predicate remains multi-thread ready if the variable is added.
- When testing for the equality of two embedded locks, it is possible that the respective equality test on the pointer variables does not yield a result, i.e. it

is unknown if the pointers are equal or not. Hence, it may be impossible to decide if two locks are equal or not simply because we cannot decide this for the respective pointers.

Both cases can only arise with embedded locks. In the first case, a function performs an operation on a lock but the pointer to the structure surrounding the lock is stored in a variable that can potentially be modified by another control flow. If this is possible, the current control flow has no way to access the same lock again, i.e. it cannot be sure that other previous or subsequent operations on a lock accessed via the same pointer variable actually refer to the same lock instance. If a scenario like this is in fact possible it is almost certain that there is a bug in the code being analyzed. An example of such a bug was considered in the code fragment in Figure 2.6.

In the second case, the situation is similar. Here a function operates on two embedded locks that fall into the same class. However, the information stored in the relevant predicate is not sufficient to decide if the two locks are actually the same. Consider the code fragment in Figure 6.1 (a similar code fragment appears in the d_move function of the Linux kernel). In the true branch of the **if** statement, we know that target is less than dentry, i.e. the two pointers differ. However, in the false branch we can neither conclude that target and dentry are equal nor that they are

```
void d_move(struct dentry * dentry, struct dentry * target)
{
        /* ... */
        if (target < dentry) {
                spin_lock(&target->d_lock);
                spin_lock(&dentry->d_lock);
        } else {
                spin_lock(&dentry->d_lock);
                spin_lock(&target->d_lock);
        }

        /* Do something with dentry and target */

        spin_unlock(&dentry->d_lock);
        spin_unlock(&target->d_lock);
}
```

Figure 6.1: Incomparable locks (derived from Linux kernel's d_move function)

different. Thus we do not know whether the two spin_lock operations in the false branch refer to the same lock or to two different locks. The actual d_move function is never called with dentry equal to target. This immediately follows from the fact that the spin locks used in the Linux kernel are mutex locks, i.e. if dentry and target were equal, this would lead to a dead lock on dentry−>d_lock. This means that for this specific case, the problem can easily be solved by adding a pre-condition to the function which requires that the first and the second formal parameter are not equal. In general it is also possible to split the predicate into one where the two pointers are equal and another one where they are different.

Nonetheless, in many cases it is an indication of a bug or a non-obvious pre-condition if the analysis cannot decide whether two locks that are in the same class are equal or not. Therefore it is often useful to flag such cases as bugs and let the user add an appropriate pre-condition.

Lock Counter Semantics

From an analysis point of view we are not really interested in the absolute value of a lock counter. Instead we are only interested in the changes that are made to the lock counter by the current thread. This is a consequence of the requirement that each thread acquires and releases locks in a balanced way. Thus we will treat each lock as a counter that is local to the current thread. Only those changes to the lock counter that are performed by the current thread affect this local view on the lock counter. The locking primitives that are used to modify the lock counter ensure that the global value of the lock counter is within the constraints.

Treating the lock counter as a thread local variable allows us to include information about the lock counter in multi-thread ready predicates. However, for other variables the predicate contains *restrictions* on the *value* of variables, whereas lock counter information in a predicate describes the *changes* made to the lock counter during execution of a block of code or function. E.g. a predicate might state that the lock counter of a specific lock is increased by one in a block of code which does not tell us anything about the absolute value of the lock counter. Conceptually, this information can be expanded into a set of predicates – one for each initial value of the lock counter – that describes the status of the lock counter at the end of the block or function. This can be done in a way that is similar to the handling of function arguments. An actual implementation of predicates will combine all those predicates that differ only in the initial value of a lock counter into a single predicate that stores lock counter information relative to the initial value at the start of the function.

Lock Counter Information

For each lock, we store several integers that describe the behavior of the lock counter. These values are intended to give an aggregate description of the lock operations that are performed by a function. The following integer values are maintained for each lock:

cur The current value of the lock counter relative to the value of the lock counter at the start of the function.

max The maximum relative value of the lock counter that can be reached during execution of the function.

min The minimum relative value of the lock counter that can be reached during execution of the function.

req This value is either zero or one. If the value is one, this means that all the unlock operations accounted for in the *min* counter must not lead to a value of the lock counter that is zero. In other words, at the start of the block $min + req$ consecutive *unlock* operations must be possible without resulting in a negative value of the lock counter but the block of code in question performs at most *min* of these *unlock* operations.

The first three of these values are relative to the value of the lock counter at the start of the function. Only *req* is an absolute value.

We can immediately see that *max* cannot be negative whereas *min* cannot be positive. *cur* can be both positive and negative but must always be in the range defined by *min* and *max*. *req* is usually zero but it can be one as the result of an externally specified locking rule. E.g. if a function contains a write access to a variable that is protected by a specific lock as specified by an external locking rule, *req* will be set to one at that point. Section 6.3.2 describes the use of these values during analysis.

Notational Convention

Information about a lock that is included in a predicate is indicated by $Lock(\ldots)$. Inside the parentheses the lock itself is specified by means of its class and in case of an embedded lock a pointer to the surrounding structure. Additionally, the values of the four counters *min*, *max*, *cur*, and *req* are given unless they are zero.

6.3.2 Lock Specific Predicate Operations

```
void assign (entry * a, entry * b) {
                /* 1. {[]} */
        assert (a != b);
                /* 2. {[a ≠ b]} */
        lock (a->mutex);
                /* 3. {[a ≠ b, Lock(a−>mutex, max = cur = 1)]} */
        b->counter = a->counter;
                /* 4. {[a ≠ b,  Lock(a−>mutex, max = cur = 1),
                                Lock(b−>mutex, req = 1)]}           */
        unlock (b->mutex);
                /* 5. {[a ≠ b,  Lock(a−>mutex, max = cur = 1),
                                Lock(b−>mutex, min = cur = −1)]}     */
}
```

Figure 6.2: Function analysis with locks

The use of the lock counter information described in the previous section is best illustrated by means of an example. Let us assume that a structure type named entry contains two fields: an integer counter and an embedded mutex lock in the structure field mutex. Additionally an externally provided locking rule specifies that an access to the field named counter is only allowed if the mutex in the same structure is held by the current thread. With this knowledge, we analyze the function in Figure 6.2 and keep track of the status of the different locks involved.

In Step 3, a lock operation on a−>mutex is performed. As a consequence, information about that lock is added to the predicate. A lock operation increases the *cur* counter and if necessary *max* is adjusted accordingly. Step 4 is a bit more interesting: First we make use of the fact that at this point $a \neq b$. This means that a−>mutex and b−>mutex are different locks. From the externally provided locking rule, we also know that accesses to a−>counter and b−>counter require that a−>mutex and b−>mutex are held. For a−>mutex this is known to be true as a consequence of the previous locking operation. However, we know that the current function does not acquire b−>mutex before the access to b−>counter. Thus, the caller must acquire b−>mutex before the call to this function. In the predicate this is indicated by the fact that $req = 1$ for b−>mutex. All other counters for b−>mutex are zero because up to this point, the lock has not been modified in this function. Note that no information about the field counter is added unless it is known to be static. Finally, an unlock operation on b−>mutex is performed in Step 5. The effect of an unlock operation is similar to a lock operation: *cur* is decreased by 1 and *min* is adjusted as necessary. However, there is an important difference: If the new value of the lock counter reaches a new minimum (i.e. *min* is actually changed) and *req* is not zero the new value of *req* must be set to zero. This is necessary because the new decreased value of *min* already guarantees that the current thread still holds at least one instance of the lock if the lock counter reaches the old value of *min*.

More precisely, the following rules apply to locks in the analysis of a function:

Uniqueness A lock must not be mentioned more than once within a single predicate not even under a different name. If this cannot be guaranteed either the predicate must be split or analysis fails and an error is raised.

Lock A lock operation increases *cur*. Furthermore, *max* is set to the maximum of the new value of *cur* and the old value of *max*.

Unlock An unlock operation decreases *cur*. If *cur* is less than *min*, *min* is set to *cur*. If *req* is set and the value of *min* was changed, *req* is set to zero. Otherwise the value of *req* is not changed.

Required Lock Count If a certain operation requires that a lock is held, i.e. that the value of the lock counter is greater than 1, the *req* counter might need to be changed:

- If the value of *cur* is strictly greater than *min*, it is already clear that the current thread still holds at least one instance of the lock. In this case *req* is left unchanged.
- If *cur* equals *min*, the value of *req* is set to 1 independent of the old value of *req*.

Errors If the maximum value of the lock count for a specific lock is known and $max - min$ is greater than this value, this indicates an error because the lock counter will leave the permitted range in the scenario described by this predicate no matter what the initial value of the lock counter is.

6.3.3 Normal Predicate Operations and Locks

In the previous section, we have discussed how lock operations affect the modification of predicates. In order to use this information effectively in the analysis of a program we have to define how existing predicate operations and function descriptions are affected by the presence of lock information in a predicate. There are two different aspects that must be considered:

- In what way does the presence of lock information prevent the applicability or change the implementation of predicate operations like merging or special-of?

- How do we guarantee that changes to the information that is stored in a predicate does not lead to predicates where it is no longer possible to guarantee that any two locks mentioned in the predicate are different?

The latter point does not affect operations that add restrictions to a predicate, because the information stored in the predicate already guarantees all conditions necessary for the locks and adding additional restrictions cannot change this. I.e. those operations that add restrictions to a predicate while analyzing a statement are safe with regard to locks. However, removal of variables e.g. during the simplification step may lead to problems and must be considered separately. The following sections explain in detail how different predicate operations are affected by the presence of lock information.

Removal of Lock Descriptions

As with all other information stored in predicates, the easiest way to deal with lock information is to remove it from a predicate e.g. during finalization if it is no longer needed. A similar strategy has been used with information about variables that were determined to be dead by live range analysis. However, as opposed to variables, it is not that simple to find useful conditions that allow us to decide that information about a specific lock is no longer needed. In fact if the *cur* counter is not zero this means that the current block of code changes the lock counter and if the *req* counter is not zero, this means that the current block of code performs an operation that requires the lock to be held but the lock is not acquired within the block of code. In both cases, the lock information must be preserved to keep track of the changes to and requirements for the lock counter. Similarly, if the *min* counter is not zero this implies that the current block drops (and potentially reacquires) the lock. This information must be preserved, too. Thus the *max* counter is the only counter that can be allowed to be non zero in lock information, which is about to be removed. In fact we remove information about a lock if and only if all counters except the *max* counter are zero. Lock information with this property means that in the current block all instances of the lock that are acquired are also released. Additionally, if any operation is performed that requires an instance of the lock to be held, the lock is properly acquired within the block itself.

As a consequence of the lock information being removed, it is not possible to know exactly which locks are acquired on the path from the start of the block or function to the current statement if the relevant lock already has been released. This means that paths that potentially cause a deadlock in the current thread, because the thread acquires a lock that is already held a second time, might not be detected. However, as opposed to paths where a resource is accessed without holding the proper locks this is a smaller problem because compared to silent data corruption such a deadlock is relatively easy to debug. E.g. a watch dog can detect that the thread is stuck and obtain a backtrace. The backtrace usually shows quite clearly what sequence of events lead to the deadlock.

The removal of lock information according to the rules specified above does not have other adverse effects for the correctness of the analysis because lock information that pertains

- to changes to the lock counter that have not been reversed or
- to operations that had unsatisfied requirements for a lock

cannot be removed.

Removal of a Variable

Variables are removed from a predicate, e.g. if live range analysis has determined that the value stored in the variable is dead. This means that either the variable will not be used before it goes out of scope or the next operation is one that assigns a new value to the variable. It is still useful to remove dead variables from a predicate even in the presence of lock information. However, if the variable being removed is used as the pointer to identify an embedded lock there are two problems:

- As a consequence of the removal, an inequality affecting the removed variable might also be removed from the predicate. This in turn can make the lock incomparable to another lock also mentioned in the predicate which is not allowed.
- With the removal, the information stored about the lock becomes inaccessible, i.e. there is no way that this lock information will ever be changed. But then, it is not possible to simply remove the information because e.g. a non zero *req* counter must somehow be propagated to the calling function.

If there exists a variable that is equal to the variable that is about to be removed, there is a very simple and obvious solution to both of these problems: All occurrences of the dead variable in the lock information are replaced with an arbitrary variable that equals the variable that is about to be removed. Additionally, if the lock information will be removed anyway during finalization according to the rules described in the previous section, we can as well do this now.

Thus, we only need to consider those cases where the variable that is about to be removed is used as the pointer in the lock description of an embedded lock and the variable is the last and only one that is known to contain that pointer's value. This means that the associated lock information will definitely become inaccessible

```
void lock_page_zone (struct page * p)
{
        struct zone * z = p->zone;
        lock (z->lock);
}
```

Figure 6.3: Inaccessible Lock

to the analysis. As we have already determined that the lock information must be preserved, the inaccessible lock information is reported as a bug.

In order to explain why this is a reasonable way to deal with inaccessible locks, let us assume that the information stored in the predicate is accurate with regard to the inaccessibility of the lock information. This means that neither the function itself nor its caller can be sure that they are operating on exactly that lock that has just become inaccessible. Thus, it is impossible to bring a non zero *cur* counter back to zero. Consider for example the code fragment in Figure 6.3. After the lock operation, the variable z is dead unless the field zone in **struct** page is static (see Section 3.7). Hence, if zone is not a static field, the lock information about z−>lock becomes inaccessible at the end of the function. However, this inaccessibility is not limited to our analysis because the calling function does not have a way to access this exact lock, either, as p−>zone can change at any time. Thus, while it is possible for the code to access the lock, e.g. if p−>zone happens not to change at the wrong time, there is no *fail safe* way to access the lock. However, the current thread *must* access the lock because it is unbalanced. Consequently, the current thread cannot release the lock that it has acquired and we have found a bug.

Of course, it is still possible that our analysis is not accurate enough and believes that the lock is inaccessible while the way to access it is just not detected by the analysis. Nonetheless, in both cases manual intervention is needed either to fix the bug or to explain the behavior of the code to the analysis. Therefore it is generally safe to flag code like the one in Figure 6.3 as buggy. Similar considerations apply to other inaccessible locks.

With the presence of lock information, a problem similar to the one illustrated in Figure 4.1 on page 71 can occur. Thus, it is even more important that variables that hold the value field in a field-of relation are not removed from the predicate as long as the value can also be obtained by means of a component reference.

Finalization of Predicates

One of the most important operations performed on predicates is the simplification step done during finalization, i.e. the removal of information, that is no longer useful, from a predicate. We have already defined under what circumstances lock information can be removed from a predicate and in the previous section we have described how the removal of variables is affected by the presence of lock information. This allows us to describe the changes made to the finalization step of a predicate:

- Before anything else, lock information that can be removed according to the rules specified above is removed from the predicate.
- If removal of a variable that is still live but does not carry any useful information would render a lock inaccessible this variable is *not* removed from the predicate. This is different from the scenario discussed in the previous section where the variable potentially did carry information, but the variable itself was determined to be dead.
- Finally, the calculation of the predicate's hash can be extended to include lock information.

Otherwise the finalization is done as described for the analysis without lock information in Section 4.2.2.

Special Case

We are given two predicates p_1 and p_2 that may carry information about locks and we want to adapt the special case operation defined in Section 4.4.1 to this new situation. First of all, it is still necessary that p_1 is a special case of p_2 if the lock information is ignored. For the remainder of this section we will always assume that this is true for p_1 and p_2 because otherwise we already know that p_1 is not a special case of p_2.

It is clear that p_1 is in fact a special case of p_2 if the lock information carried by p_1 and p_2 is identical. Precisely, this means that each lock mentioned in p_1 can be paired with exactly one lock mentioned in p_2 such that

- both locks fall into the same class,
- if the class contains embedded locks the pointer variables for the two locks must be identical or equal in p_1, and
- corresponding counters in both locks have the same value.

The first two conditions ensure that the two locks in a pair actually refer to the same lock. This relies on the requirement that any two locks mentioned in a predicate must be mutually different. The last condition ensures that the information about each lock is the same in p_1 and p_2.

Merging Predicates

As with the special case operation, two predicates p_1 and p_2 can be merged if the lock information on both predicates is the same and p_1 and p_2 can be merged if the lock information is ignored. However, if p_1 and p_2 are equal ignoring lock information it is in certain cases possible and useful to merge lock information. Similar to the special case operation described in the previous section we still require that each lock mentioned in p_1 can be paired with exactly one lock mentioned in p_2. We allow the introduction of dummy locks that have all lock counters set to zero to complete the pairing, though. The same requirements as for normal locks apply for the introduction of such a dummy lock. Especially, it must be possible to prove from the information stored in the predicate that any two locks mentioned therein are different. In order to merge the two predicates p_1 and p_2 it is also necessary that the respective *cur* counters of paired locks in p_1 and p_2 have the same value.

In this case it is possible to merge the lock information of the two predicates as follows:

- The *max* counter of each lock in the merged predicate is the maximum value of the *max* counters of the corresponding locks in p_1 and p_2.
- The new value of the *min* and *req* counter are taken from the predicate with the smaller value of the *min* counter[1] for the corresponding lock. If both *min* counters are equal, the *req* counter is taken from the predicate with the higher *req* counter.

Such a merge does introduce a slight change in the semantic meaning of the lock information in predicates: E.g. the *max* counter of the lock information in a predicate does no longer guarantee that the lock counter reaches a relative value of *max* on each path from the start of the function. It only guarantees that there is no path along which the lock counter exceeds a relative value of *max*.

Now let us assume that only the requirement with regard to the *cur* counters prevents the merge of two predicates, i.e. p_1 and p_2 are identical except for lock information and there exists a lock for which the *cur* counters in p_1 and p_2 have different values. This means that there are two possible paths from the start of the function to the current statement for which the number of lock or unlock operations performed along the path differs and the information stored in the predicates is not

---

[1] This corresponds to the greater *absolute* value

sufficient to distinguish these paths. There are two possible explanations for such a situation:

- It is in fact possible to distinguish both paths but the relevant information to do so is not stored in the predicates.
- It is not possible to distinguish the two paths in a multi-thread safe way.

In the latter case it is almost certain that the code being analyzed has a bug because the current thread must at some point release all locks that it has acquired and there is no way to determine how many times the lock in question has been acquired by the current thread. In the former case, there is a similar problem, except that it is the analysis that cannot distinguish between the different cases. This means that subsequent analysis will sooner or later fail to deduct that all locks held by the current thread have been released.

In both cases, manual intervention is needed: Either the bug in the program must be fixed or the piece of code that is not understood by the analysis must somehow be excluded from the analysis and described manually. Often, the problem is in fact in the real code and not in the strength and accuracy of the analysis.

The consideration in this section ensures that adding lock information to predicates does not prevent the merging of predicates that could have been merged if lock information is ignored.

Function Descriptions

The building of function descriptions in the presence of lock information works exactly the same way as before, i.e. the set of predicates that are possible after a return statement form the function description. All variables that will go out of scope once the function returns are removed from the predicate and if any lock information becomes inaccessible due to this removal, an error is raised as described in the previous section on the removal of variables.

However, due to the relative nature of the counters in lock information, the semantic interpretation of lock information differs from the interpretation of normal restrictions on variables in a predicate: Normal variable restrictions are absolute, e.g. a variable restricted to the range $(0 \ldots 10)$ has an absolute value in that range. Conversely, the counters in a lock description refer to the value of the lock counter at the start of the function. Note that in this context, the value of the lock counter refers to the thread local view on the lock counter as described in Section 6.3.1. The lock description is actually a template that allows the construction of predicate instances that apply to a specific initial value of the lock counter.

Use of Function Descriptions

Recall, that in order to calculate the effect of a function call if there is no lock information involved, we first remove all restrictions that can be affected by the function call from the predicates in the *Pre*-set. The *Post*-set of the function call is then obtained by calculating the mapped intersection of each predicate in the modified *Pre*-set with each predicate in the function description of the call.

This does not change in the presence of lock information. However, in addition the lock information from the function description must be merged with the lock information from the *Pre*-set. This is done separately for each pair of predicates.

Firstly, if the mapped intersection results in an empty predicate, i.e. the predicate from the *Pre*-set and the predicate from the function description are incompatible, there is no need to merge lock information. Secondly, we notice that the removal of information from the predicates in the *Pre*-set can render locks mentioned in the predicate incomparable. If this happens, an error is raised unless the lock information can simply be removed from the predicate. In the following, we will consider a single predicate p that has been derived from a predicate in the *Pre*-set and a predicate f from the function description. The result is a predicate r that initially consists of the mapped intersection of p and f where all lock information has been ignored. In a second step the lock information from p is added to r as is. This operation cannot lead to incomparable or equal locks in r because all locks are provably different in p and as an intersection of p and f the predicate r is a special case of p which means that the locks are also known to be different with respect to r.

In order to merge the lock information from f into r, it is necessary to associate each lock mentioned in f with exactly one lock that is mentioned in r such that both locks are known to be equal in all cases that are covered by r. As has been done in similar cases, it is possible to add locks with all counters set to zero to r as long as they are known to be different from all other locks mentioned in r. After all pairs have been formed, the lock information from each lock in f is merged with the corresponding lock in r.

For the sake of clarity we will assume that the relevant lock is fixed in all predicates and the counters that comprise the lock information of that lock in each of the predicates p, f, and r is denoted by $pred.counter$. This means that e.g. $f.min$ denotes the *min* counter of the lock in predicate f. With this notation, the lock counters of the lock in the merged predicate r are calculated as follows:

- The *cur* counters from the lock description in both p and f are added. The result is the *cur* counter of the merged lock description, i.e. $r.cur = p.cur + f.cur$.
- The minimum value that the lock counter can reach during execution of the function call is the sum of the *cur* counter in p and the *min* counter in f, i.e.

$p.cur + f.min$. If this value is less than the value of the *min* counter in p it becomes the new value of the *min* counter in r. Otherwise the *min* counter in r is the same as the *min* counter in p. Thus $r.min = \mathbf{min}(p.min, p.cur + f.min)$.

- Analogously, the value of the *max* counter for r is either taken directly from p or calculated as the sum of the *cur* counter from p and the *max* counter from f, whichever is greater, i.e. $r.max = \mathbf{max}(p.max, p.cur + f.max)$
- The value of the *req* counter is taken from the predicate that determines the *min* counter in r. In other words this means that we calculate $\mathbf{min}(p.min - p.req, p.cur + f.min - f.req)$ and if this value is less than the newly calculated value $r.min$, we set $r.req = 1$. Otherwise $r.req$ is 0.

The reasoning behind the last rule is as follows: If the minimum value of the lock counter is reached during the function call (i.e. $r.min = p.cur + f.min$) and at that point the current thread must still hold at least one instance of the lock (i.e. $f.req = 1$), we can conclude that at the point where the lock counter reaches $r.min$ the lock must still be held (i.e. $r.req = 1$). Similarly, if the minimum value of the lock counter is reached somewhere before the function call (i.e. $r.min = p.min$) and the predicate p states that at that point the lock must be held (i.e. $p.req = 1$) we can also conclude that $r.req$ must be 1. Of course it is possible that both of these cases apply at the same time. But then, if neither of them applies we can conclude $r.req = 0$ as the lock must not be held at any point where the counter reaches $r.min$.

Basically, the meaning of these rules is that all counters in f are interpreted relative to the *cur* counter in p.

Additional Error Conditions

The extended function descriptions introduced in the previous sections lead to additional requirements that are necessary for the function to be called. E.g. if the function description contains lock information with a non-zero *min* or *req* counter, the current thread must acquire that lock before it is allowed to call the function. Similarly, calling a function with a positive *cur* counter in its function description creates a responsibility for the calling thread to decrease the lock counter later on. In some sense this is similar to the pre-conditions of a function that have been discussed earlier. However, there is an important difference: Pre-conditions are checked immediately before the function call and if the pre-condition cannot be proven, an error is raised. Then again, the aforementioned conditions regarding locks do not lead to an error immediately. Instead, the lock information of the called function is merged into the lock information of the calling function. This way any pre-conditions that are not necessarily true are propagated and may result in pre-conditions for the calling function.

In general, this is desirable, e.g. consider the hypothetical example in Figure 6.4. The timed_unlock wrapper around the unlock call can only be called if ptr−>lock is held and calling the function will decrease the lock counter by one. The unlock_twice function uses timed_unlock twice in order to reduce the lock counter by two. This usage does not yield an error (unless ptr−>lock is a mutex lock) even though it is not possible to prove from the code of unlock_twice that ptr−>lock is held before each call of timed_unlock. Instead, the function description of unlock_twice simply indicates that unlock_twice can only be called if the current thread holds at least two instances of ptr−>lock.

Normally this behavior is desired. However, if the function being analyzed is an entry point, i.e. a function that can be called from the outside to spawn a new thread, this behavior is not the desired one. In Figure 6.4 the function syscall represents such a function, i.e. syscall can be called from the outside and the call to this function spawns a new control flow that terminates once syscall returns. In this case it is a bug if ptr−>lock is used in an unbalanced way as it is the case in Figure 6.4. Thus, if the function description of an entry point contains lock information this is marked as a bug, otherwise lock information is propagated.

6.4 Pointers to Locks

In general, pointers to locks do not work well together with the analysis methods described above. However, in the systems we have studied, the case of an embedded lock that is fully handled by the methods described in this chapter was by far the most common case of pointers to locks. In this section, we have a look at certain other constructs regarding pointers to locks and how these can be handled during analysis.

6.4.1 Local Pointers to Locks

A common use for pointers to locks occurs, if a local variable is initialized once and from then on points to a specific instance of an embedded lock. After initialization, the pointer variable is used as an abbreviation for the embedded lock. This situation can be detected and accesses to the lock pointer are translated to accesses to the embedded lock that the lock pointer points to.

```
int unlocktime = 0;
void timed_unlock (struct stat * ptr)
{
                /* 1. {[]} */
        int stime, etime;
                /* 2. {[]} */
        stime = time();
                /* 3. {[]} */
        unlock (ptr->lock);
                /* 4. {[Lock(ptr−>lock, min = cur = −1)]} */
        etime = time();
                /* 5. {[Lock(ptr−>lock, min = cur = −1)]} */
        unlocktime += etime - stime;
                /* 6. {[Lock(ptr−>lock, min = cur = −1)]} */
}
void unlock_twice (struct stat * ptr)
{
                /* 1. {[]} */
        timed_unlock (ptr);
                /* 2. {[Lock(ptr−>lock, min = cur = −1)]} */
        timed_unlock (ptr);
                /* 3. {[Lock(ptr−>lock, min = cur = −2)]} */
}
int syscall (struct stat * ptr)
{
                /* 1. {[]} */
        unlock_twice (ptr);
                /* 2. {[Lock(ptr−>lock, min = cur = −2)]} */
        return 0;
                /* 3. {[Lock(ptr−>lock, min = cur = −2)]} */
}
```

Figure 6.4: Lock counter propagation

6.4.2 Additional Locking Functions

Sometimes, a pointer to a lock is passed to a wrapper function that performs a well defined operation on the lock and does some other work. If this occurs, the locking operation performed in the wrapper function cannot be associated with the proper lock in the caller because the association of a lock pointer with an embedded lock cannot be represented by means of our predicates. Such a situation can be resolved by treating the wrapper function as a lock primitive in itself. This means that a function description for the wrapper function must be supplied by the user. This function description usually includes lock information for the lock operation performed by the wrapper function and no other restrictions on variables.

6.4.3 Container Locks

In some cases, each structure of a certain kind is assigned to a container upon creation. An example might be an I/O-Buffer that belongs to a certain device structure or an inode structure that is inserted into a hash bucket. In this situation, there often exists a lock in the container structure that protects some or all fields of all structures that belong to the container. Usually, there is a static field in the contained structures that points to the respective container. If this is the case, the normal analysis process can handle access to the lock in the container with the help of field-of relations.

However, sometimes each contained structure has a direct pointer to the lock inside the container instead of a pointer to the container itself. If this situation occurs, normal analysis does not know how to deal with the lock pointer. Then, it is often possible to treat each lock pointer in each structure as if it were a lock of its own. This way it is normally possible to verify that each of these logically separate locks is used in a balanced way.

If this is the case and the lock is of mutex type, it is not possible to release a lock via another name than the one that was used to acquire the lock. In order to illustrate this, consider two objects n1 and n2 that both contain a field lkptr that points to a mutex lock. Let us also assume that n1−>lkptr points to the same lock as n2−>lkptr. Although both locks are equal, they are logically treated as different locks by the analysis. Now, let us consider the following sequence of events,

```
lock(n1->lkptr);
/* ... */
unlock(n2->lkptr);
```

i.e. the pointer used to release the lock is different from the pointer used to acquired the lock. If the logical locks, i.e. the lock pointers, are used in a balanced way, there must be a call to lock(n2−>lkptr) somewhere before the call to lock(n1−>lkptr) that logically corresponds to the unlock call in the code fragment given above. Thus, at the time of the lock operation on n1−>lkptr, n2−>lkptr is already held by the current thread. Consequently, the sequence above cannot actually be executed because the lock operation already leads to a deadlock.

We can see that in case of mutex locks the fact that one of the logical locks is held always implies that the corresponding actual lock is also held. In many cases this is sufficient to verify commonly occurring locking rules. There are some drawbacks though:

- There are additional ways in which the thread can deadlock on itself. Many of those cannot be detected. However, as we have ignored the possibility of deadlocks of a thread on itself all along this is not much of an additional problem.
- From the fact that a logical lock is *not* held, it is not possible to conclude that the actual lock is not held. This can lead to additional spurious error reports. E.g. consider the two objects n1 and n2 above. If the current thread holds n1−>lkptr and n1−>lkptr equals n2−>lkptr it is legal to access all data that is logically protected by n2−>lkptr. However, from the analysis point of view n2−>lkptr is different from n1−>lkptr and probably not held. Thus these accesses generate spurious errors.

Despite theses drawbacks, the approach outlined in this section can often be used to verify locking rules in the presence of container locks.

6.5 Summary

In this chapter we have shown how the analysis methods developed in the previous chapters can be extended to analyze the synchronization in a multi-threaded environment. For this purpose we have defined a generic model for locks that can be used to describe the semantics of several different locking mechanisms commonly used in multi-threaded systems. Predicates and predicate operations have been extended in order to deal with locks. This has lead to an aggregate description of lock related operations performed by a function with the help of several counters that are maintained for each lock. Finally, we have outlined solutions to some situations involving lock pointers where the standard analysis method does not apply.

Chapter 7

Case Studies

In order to show the usefulness of the methods discussed in this work, a prototype has been implemented and case studies have been conducted. In this chapter we will first describe some practical issues of the implementation in more detail. Subsequently, we present results of the case studies, this includes time measurements, examples of actual bugs that have been discovered, and programming constructs that are problematic for our analysis.

7.1 Implementation

A patched version of the GNU C-Compiler has been used to generate a text representation of the decorated syntax tree for each translation unit. The decorated syntax tree generated in this process already has most of the properties that the syntax tree must have according to Chapter 2. The most important point is that the syntax tree is already in normalized form because the compiler does this normalization. This means that no additional rewriting is necessary. Additionally, each node in the abstract syntax tree already has an ID that is unique within the translation unit. This ID is also used within the text representation of the syntax tree to reference other nodes.

All subsequent analysis is done by a set of scripts written in Perl[50]. First, the text representations of the syntax trees are transformed into a nested hash structure that can also be made persistent with the help of the Storable module in Perl. Nodes that are not used inside the translation unit and that are not visible outside of it are deleted. Due to the inclusion of header files, a typical translation unit in C can contain a large amount of declarations and definitions that can be deleted here. This step also assigns global Link IDs as described in Section 3.1. In the Perl representation, the syntax tree of a single translation unit does not contain any pointers to objects that are not part of the same syntax tree. Nodes in other translation units are referenced via their Link ID. This allows for easy on demand loading of syntax trees. In the following, the basic analysis steps described in Chapter 3 are performed. Each of these steps adds additional annotations to the syntax trees.

Finally, the actual analysis is performed one function at a time. If possible, the functions are processed in topological order, i.e. whenever possible a function is only processed after all the functions it calls have already been processed. If necessary, this is repeated until no more changes occur. This step consumes the vast majority of the total time required for the analysis.

7.2 Case Studies

7.2.1 Overview

The algorithms presented in this work have been applied to several different versions of the Linux and NetBSD kernels. In the following, we will present results for three different scenarios where detailed measurements of the runtime have been made. It should be noted, that especially in case of the Linux kernel it is not yet feasible to run the algorithms with a complete configuration that includes all possible options and drivers. The following configurations have been used:

linux-small A minimal configuration of the Linux kernel version 2.6.19. This configuration includes only the bare minimum needed to build a kernel and does not include support for networking, disk file systems, or special hardware.

linux-big A moderately sized configuration of the Linux kernel version 2.6.20 that is actually usable on a standard PC. This is still far from a configuration that includes everything that is supported. However, this configuration does reach the limits of what is still feasible with the current implementation and hardware used to conduct the analysis.

netbsd A medium sized version of the NetBSD kernel (snapshot of netbsd-current from April 2006). This configuration includes a large subset of the features available in the NetBSD kernel.

The following table summarizes the size of these three configurations:

	linux-small	**linux-big**	**netbsd**
Number of source files	300	1,450	1,300
Number of unique functions	7,000	28,000	20,000
Number of statements	300,000	2,000,000	1,700,000
Lines of code (excluding header files)	200,000	1,200,000	1,000,000

The number of source files only counts `.c` files, header files are not included. The number of functions only counts those functions that are actually analyzed. Functions that are removed from the syntax tree of a translation unit because they are `static` and never used are excluded. This type of functions is actually very common because of `static inline` functions declared in header files. Such a function is defined independently in every translation unit that includes the header file. E.g. in case of the **linux-small** configuration about 90% of all functions are of this type and can be removed. Even if such a function is used in some of the translation units where it is independently declared, it is only counted once in the table above.

Likewise, the number of statements only counts statements in those functions that are actually analyzed. This number is roughly proportional to the total number of lines of code. However, it is not directly comparable because the statement count given above does not include declarations and comments at all. But then, normalization of the syntax tree increases the number of statements. We give the number of statements in the normalized AST because, as opposed to lines of code, this number does not depend on coding style whitespace and comments.

7.2.2 Time and Performance Measurements

As said before, the following measurements and graphs always refer to a single run of the final analysis step. Each function is looked at exactly once. If the system in question is recursive (which all of the tested systems are), a couple of iterations (5-15 depending on the size of the system) are needed until a stable set of function descriptions is obtained.

Function Analysis Convergence

The main loop of the algorithm that analyzes a single function iterates until no more changes happen. Each iteration considers the effects of a single statement. We have counted for each statement how often it is considered in this inner loop, i.e. how often does the *Pre*-set of a statement change during analysis of the function. The table in Figure 7.1 shows that with very few exceptions this number is small. In fact the table

Number of Times	1	2	3	4	5	≥ 6
linux-small	86.1%	10.8%	1.8%	0.8%	0.2%	0.3%
linux-big	87.3%	10.4%	1.2%	0.8%	0.1%	0.2%
netbsd	84.4%	13.0%	1.2%	1.0%	0.1%	0.3%

Figure 7.1: Percentage of Statements that are considered n times in the inner loop of the function analysis algorithm

shows that only about 3% of all statements must be considered more than twice and only about 0.3% must be considered more than five times.

Runtime and Number of Predicates

We have previously suggested that the main factor that determines the runtime of our analysis is the number of predicates involved. This is supported by the graphs in Figures 7.2, 7.3, and 7.4. Each graph shows the total time needed for each function dependent on the number of predicates that are required for the analysis. The least squares fitting method has been used to find the best fitting line also shown in these graphs.

The graphs show that the time required for the analysis of a function grows roughly, linearly with the number of predicates required.

Number of Predicates and Function Size

Now that we have seen that the number of predicates required for a function is the main factor that determines the time required for the analysis, we would like to show that this number is linear in function size. Figures 7.5, 7.6, and 7.7[1] show the average number of new predicates generated at each statement dependent on the size of the function being analyzed. We see that the this value is bounded by about 40 and for the vast majority of functions it is less than 10. Additionally it is worth noting that the value tends to be significantly lower for larger functions. However, comparing the results for `linux-small` with those of the bigger configurations we also see that the absolute values are larger in the bigger configurations.

Thus, we can observe that within a system the number of predicates required for a function appears to grow at most linear with the size of the function. However, the overall system size influences the constant factor for this linear bound. A possible explanation for this phenomenon is that function descriptions tend to become slightly more complex in larger systems.

Time per Statement and Function Size

From the previous sections we can conclude that the average time required for a *single* statement during analysis does not grow with function size. Figures 7.8, 7.9, and 7.10 support this, i.e. the time required for analysis of a function grows linearly with the size of the function within a given system.

---

[1] Note that in these figures the value given on the y-axis is the total number of predicates required for the function *divided by the number of statements* in the function.

Conclusions

The runtime for the analysis of a function can be expected to be proportional to the number of predicates required for the analysis. This number in turn grows roughly linearly with the size of the function. This indicates that the total time required for analysis grows roughly linearly with the overall size of the system being analyzed. For very large functions, the measurements indicate that the growth in runtime is even sub-linear. However, the constant factors in these linear relationships differ depending on the system and they seem to grow slightly with the overall size of the system. Finally, it should be noted that it is certainly possible to construct functions where the runtime of our analysis grows exponentially, however, the empirical results presented in this section suggest that such functions are extremely rare in practice.

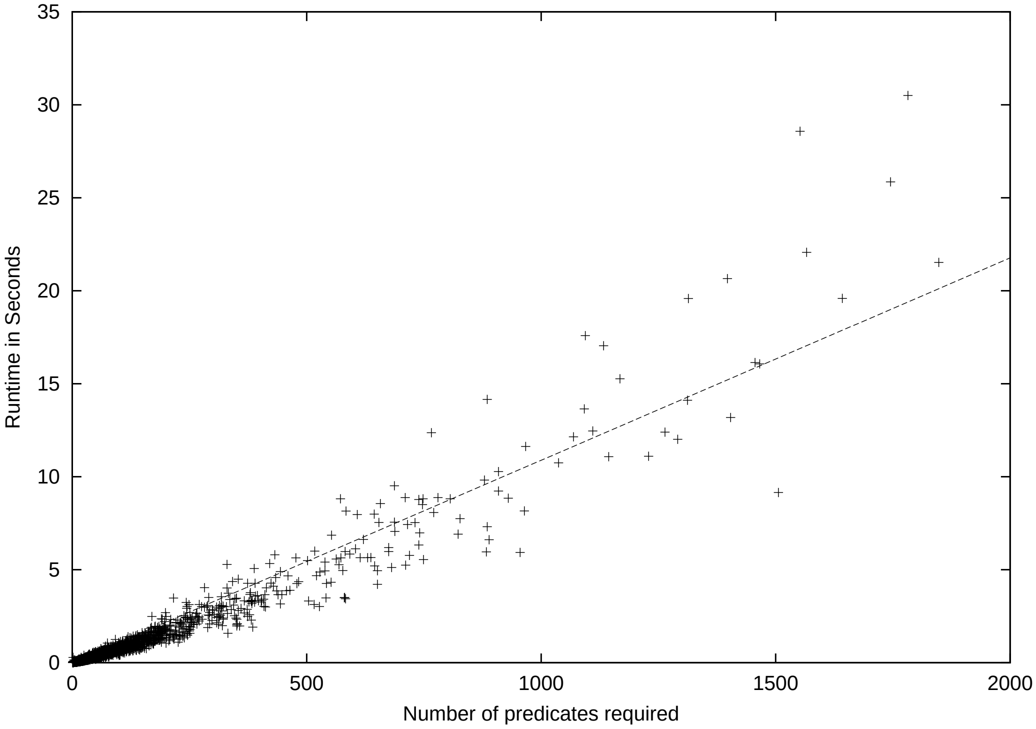

Figure 7.2: **linux-small**: Runtime for functions by number of predicates required

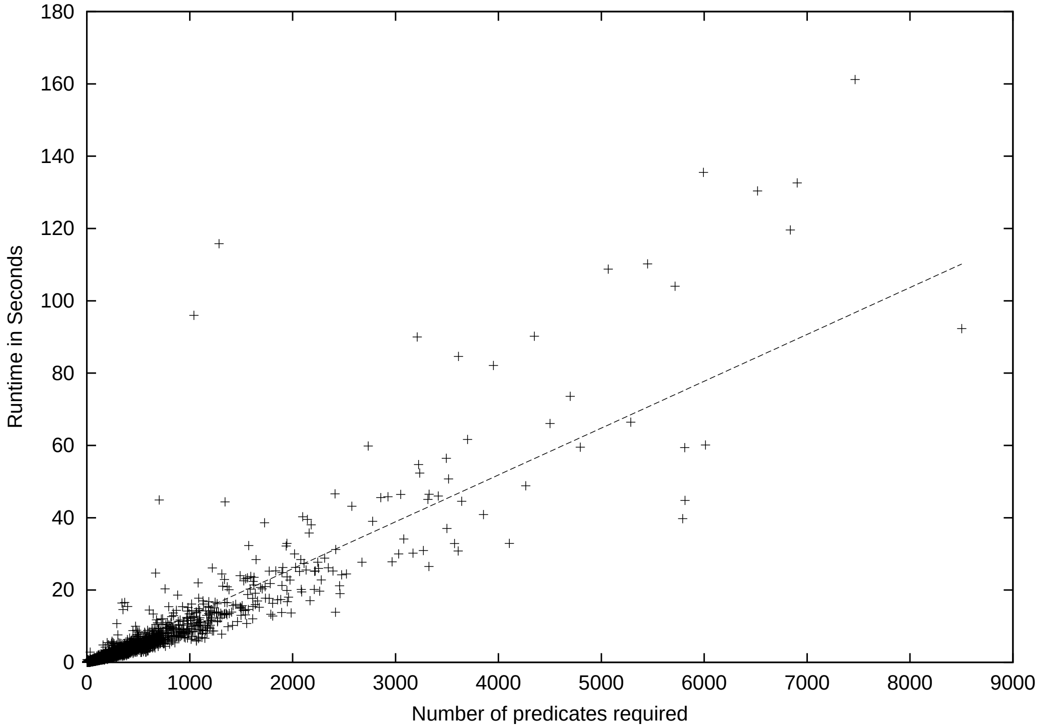

Figure 7.3: **linux-big**: Runtime for functions by number of predicates required

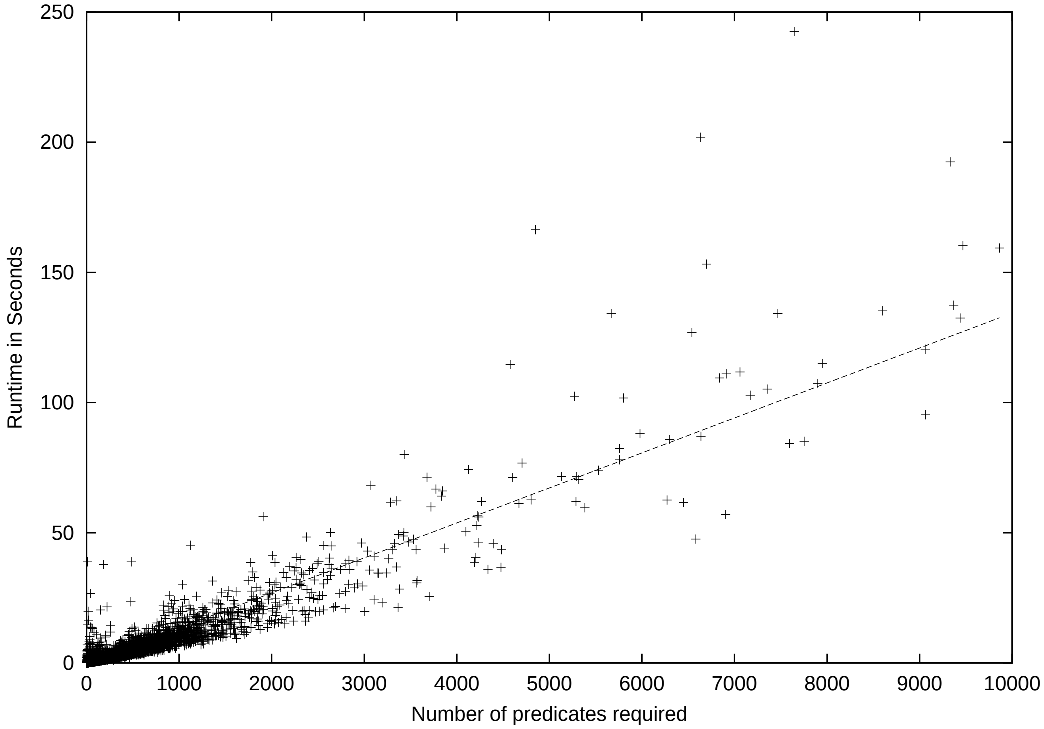

Figure 7.4: **netbsd**: Runtime for functions in by number of predicates required

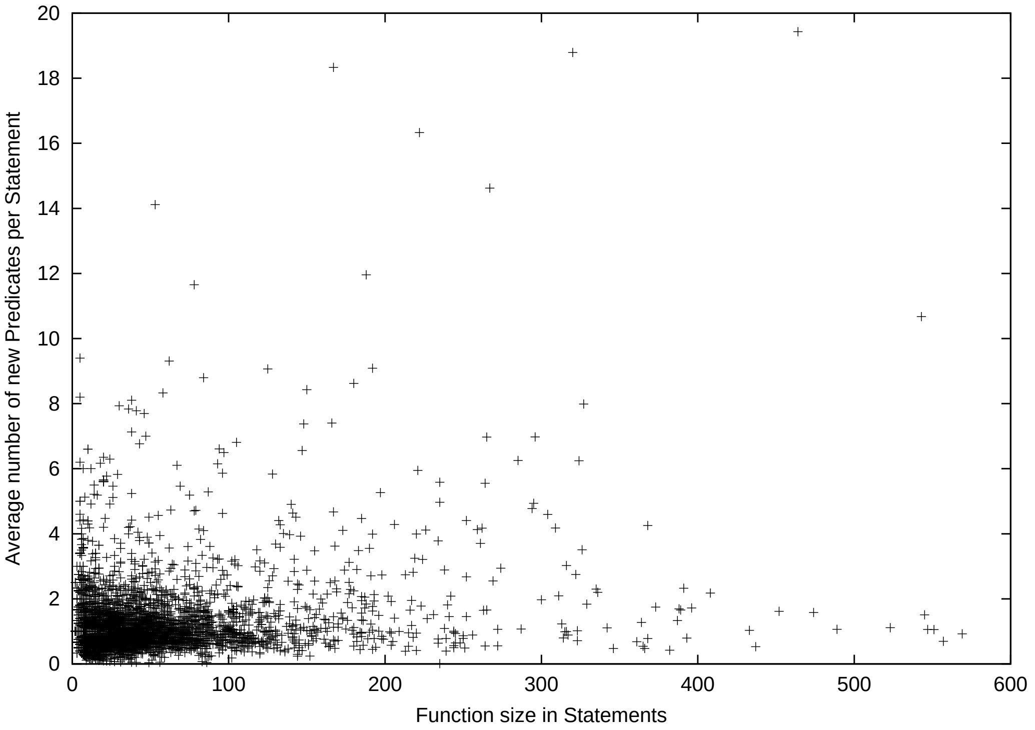

Figure 7.5: **linux-small**: Average number of predicates per statement by function size

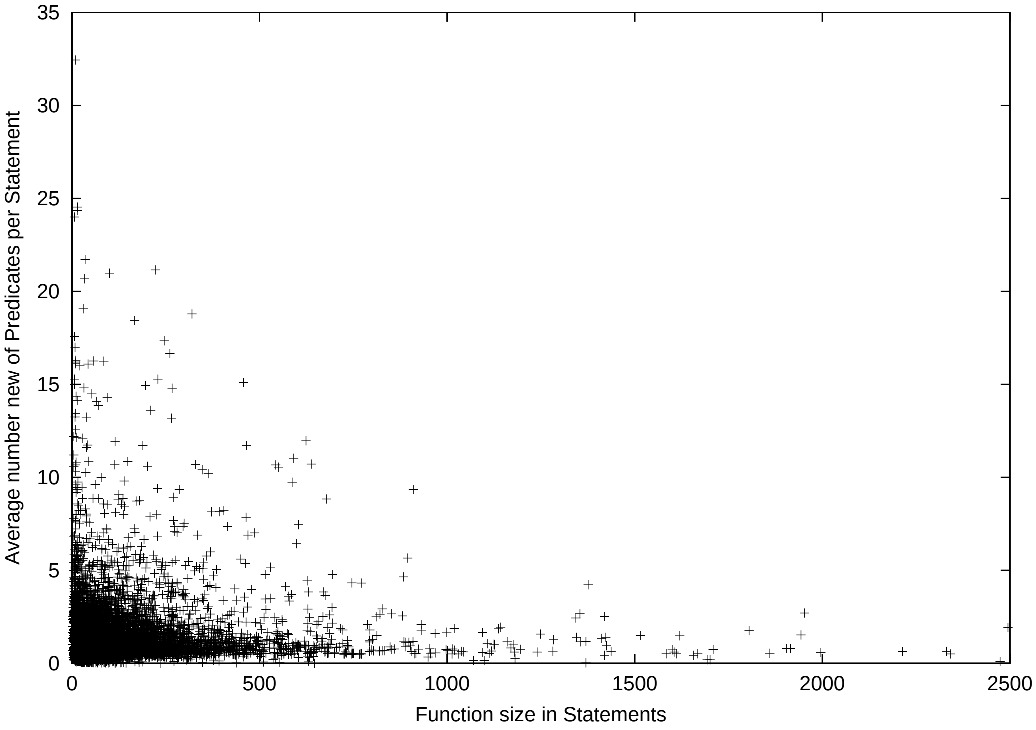

Figure 7.6: **linux-big**: Average number of predicates per statement by function size

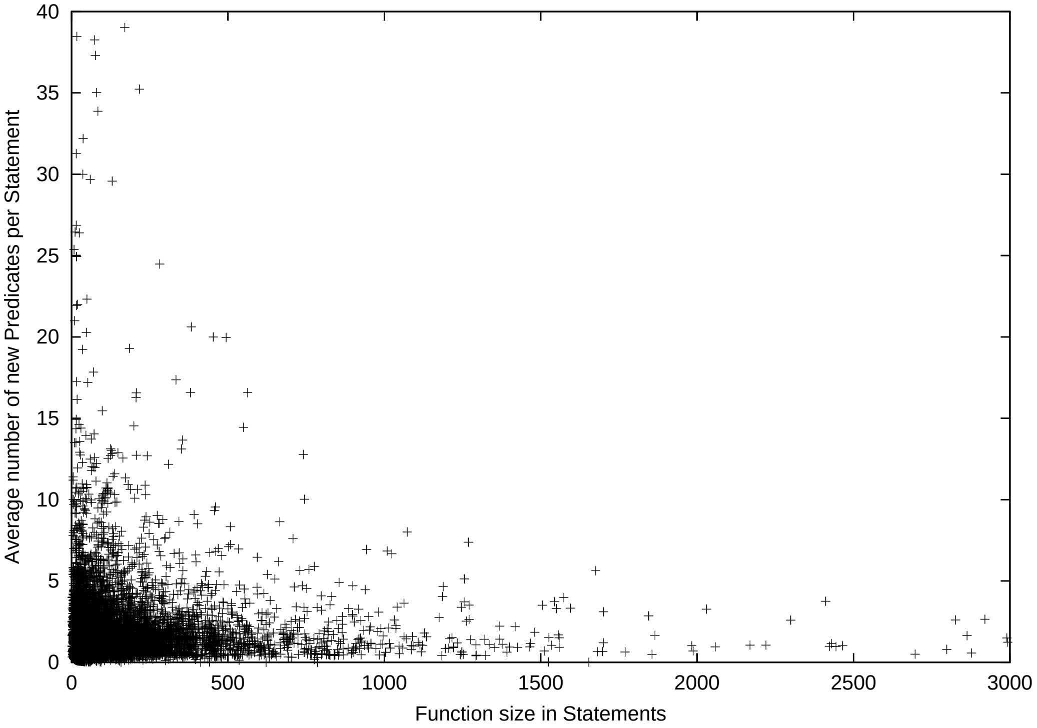

Figure 7.7: **netbsd**: Average number of predicates per statement by function size

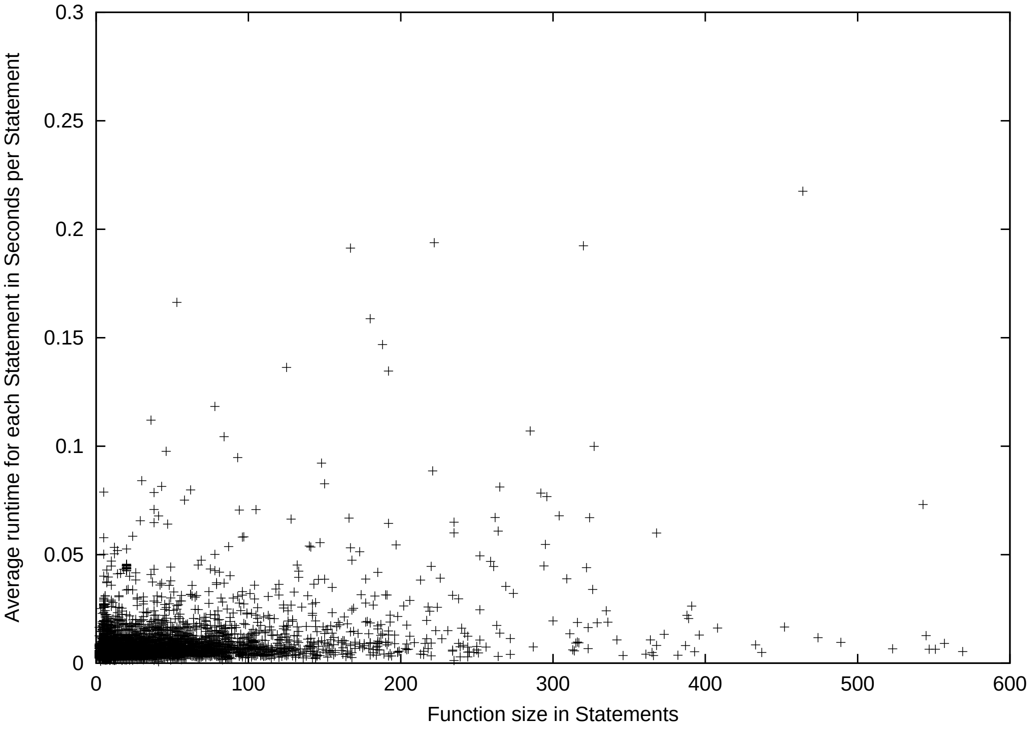

Figure 7.8: **linux-small**: Average runtime per statement by function size

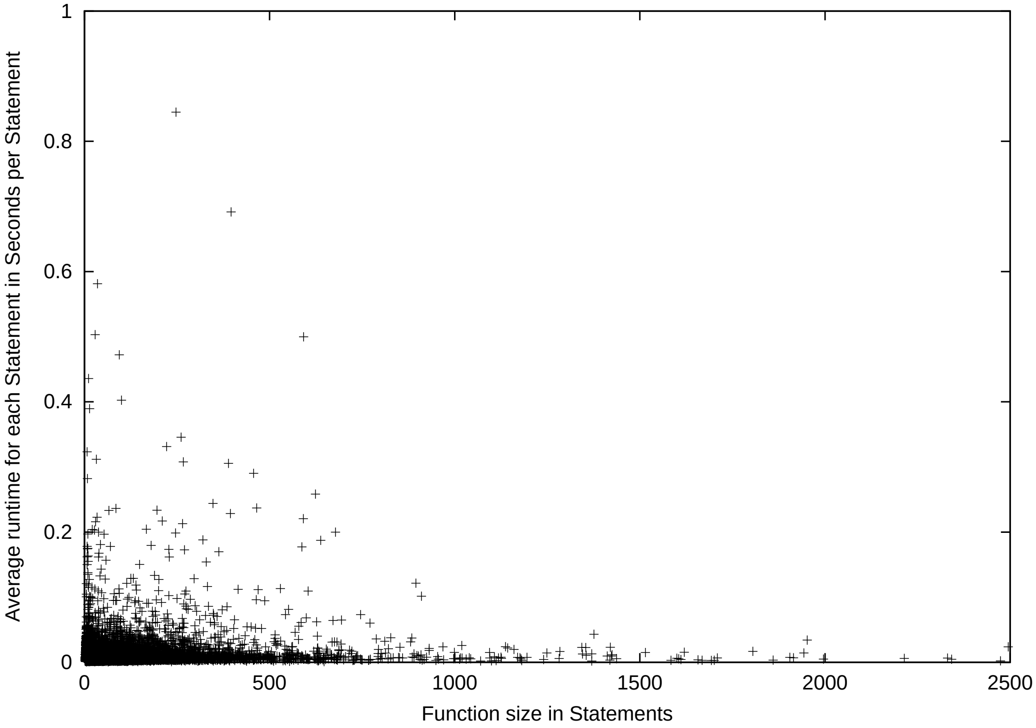

Figure 7.9: **linux-big**: Average runtime per statement by function size

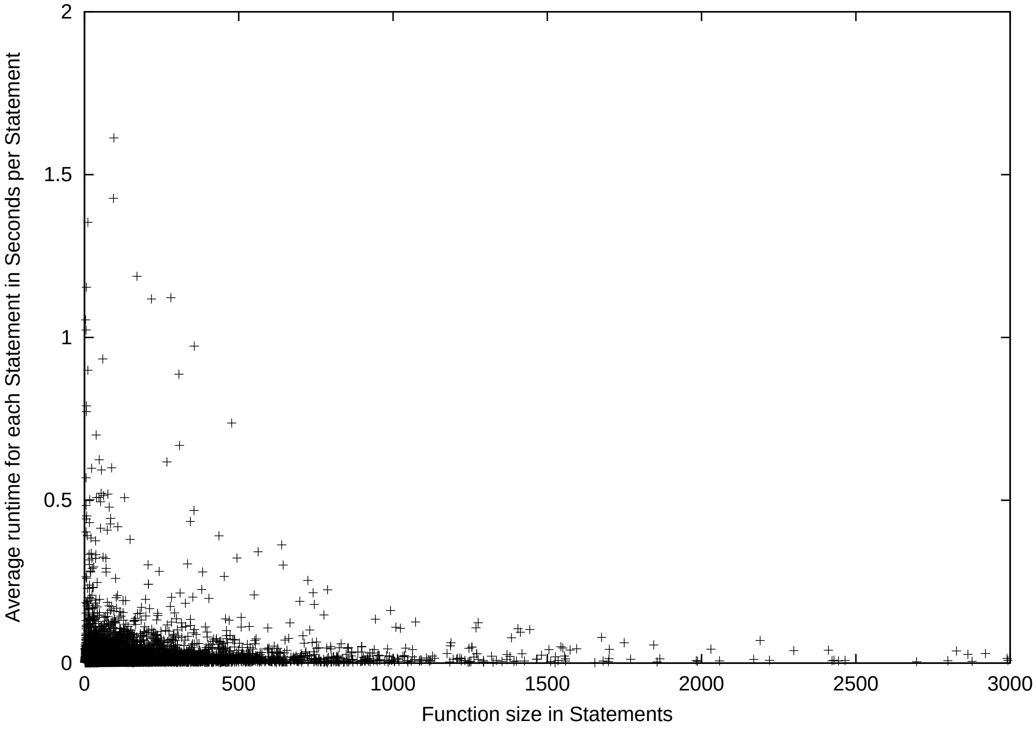

Figure 7.10: **netbsd**: Average runtime per statement by function size

```
unsigned long badness(struct task_struct *p)
{
        unsigned long points;
        struct mm_struct * mm;

        task_lock(p);
       mm = p->mm;
        if (!mm) {
                task_unlock(p);
                return 0;
        }
        if (p->flags & PF_SWAPOFF)
                return ULONG_MAX;   /* Missing unlock */
        points = mm->total_vm;
        task_unlock(p);
        /* ... */
        return points;
}
```

Figure 7.11: Excerpts from the badness function in the Linux kernel version 2.6.19

7.3 Examples of Actual Bugs

The analysis methods presented in this work have uncovered several real bugs both in the Linux and in the NetBSD kernels. This section shows some examples of bugs that have been found.

7.3.1 Random Number Pool

One of these bugs has already been shown in Figure 2.6. The problem in this code fragment is that the global variable random_state can change its value at any time. We have been able to completely crash a system running the affected kernel by putting it under a workload that stresses both the path that uses random_state under the assumption that it cannot change and the path that does change it.

7.3.2 Out of Memory Badness Function

The Linux kernel contains routines that start to kill processes in case the system really runs out of memory. Naturally, it is desirable that the system kills the process that actually hogs the memory first. The function badness calculates a score for a

```
int flags = ap->a_flags;
/* ... */
if (flags & LK_INTERLOCK) {
        simple_unlock (&vp->v_interlock);
        flags &= ~LK_INTERLOCK;
}
/* ... */
return (lockmgr(&vp->v_lock, ap->a_flags|LK_RELEASE,
        &vp->v_interlock));
```

Figure 7.12: Bug in NetBSD's layer_unlock function

process that indicates how likely it is that this process is responsibly for the memory shortage. The relevant fragments of this function are shown in Figure 7.11.

The actual bug is the missing task_unlock(p) before the marked line in Figure 7.11. This bug is not detected immediately while analyzing the function itself because it is possible that the behavior is in fact correct, e.g. because the function should return with the task's lock held if PF_SWAPOFF is set in p−>flags. However, analysis of the calling function will notice that badness may return with the task lock of the task given as the first parameter held. But then, the code of the calling function is not prepared to the fact that the lock on the task may be held. Thus, this lock will sooner or later become unreachable in the calling function and this event is reported as an error.

The net effect of this bug is that the system crashes completely if it ever runs out of memory because the functions that are intended to fix the out-of-memory problem contain a bug. It is worth noting that this particular bug is unlikely to be discovered by normal testing and in fact that particular bug remained in the source tree for more than six months and two stable kernel releases.

7.3.3 NetBSD Bugs

The first run of the analysis software on the NetBSD kernel turned up about 30 error reports. Some could be identified as false positives and were suppressed. The remaining 14 suspected errors were reported to the kernel developers and as bug number 33280[2]. One error was confirmed and fixed immediately, several others were fixed silently and the rest was confirmed as actual errors after some time.

One example for a particularly subtle bug that was found is illustrated by the code fragment in Figure 7.12. The code fragment is entered with vp−>v_interlock held

---

[2] `http`//www.netbsd.org/cgi-bin/query-pr-single.pl?number=33280: (last viewed 30 Apr 2007)

```
        if ((fp = fd_getfile(fdp, SCARG(uap, fd))) == NULL)
                return (EBADF);
        if ((fp->f_flag & (FREAD | FWRITE)) == 0) {
                error = EBADF;
                goto out;
        }
        FILE_USE(fp);
        /* ... */
out:
        FILE_UNUSE(fp, l);
        return error;
```

Figure 7.13: FILE_USE bug in NetBSD

and the code should release the lock on all paths if and only if LK_INTERLOCK is set in ap−>a_flags. The lockmgr function will release the lock passed as the third argument if and only if the flag LK_INTERLOCK is set in the second argument. This means that the lock is dropped twice if LK_INTERLOCK is initially set in flags, once explicitly by the call to simple_unlock and once implicitly by lockmgr. The problem is that the second argument of lockmgr should be flags|LK_RELEASE instead of ap−>a_flags|LK_RELEASE. The analysis reports the duplicate unlock as an error. This bug shows the usefulness of the analysis because this particular bug is likely to be missed by manual inspection as well unless the kind of problem we are looking for is known precisely.

The somewhat surprising semantics of the FILE_USE macro led to another noteworthy bug that has similar characteristics. The relevant code fragment is shown in Figure 7.13. The function fd_getfile returns either NULL or a file pointer fp and the embedded simple lock fp−>f_slock is held. FILE_USE in turn increases the reference count of the file pointer and *releases* fp−>f_slock. FILE_UNUSE must be used to reduce the reference count but FILE_UNUSE does not release fp−>f_slock. In the error path the jump to the label out skips the call to FILE_USE and fp−>f_slock is never dropped.

7.4 Problematic Program Constructs

Function analysis is usually accurate enough, i.e. the function descriptions accurately represent the relevant effects of a function. However, some functions are not fully understood. These functions either need hints in order to obtain precise enough results from the analysis or a function description must be provided manually by the user. In this section, we show some programming constructs that have actually caused problems in our case studies.

7.4.1 Function Pointers

The way we deal with function pointers is one reason that can cause problems for the analysis. Problems that arise from inaccurate alias analysis for function pointers have been very rare but they do exist. The major problem that arises results from the fact that function pointers in a certain structure are often not independent. E.g. consider a virtual file system layer that supports several physical file systems. Some of these physical file systems support transactions while others do not. Let us further assume that each physical file system provides two function pointers start_transaction and end_transaction in the kernel representation of its super block. A typical use of such a super block in the virtual file system layer might look like this:

```
(*sb->start_transaction)();
/* Do some file system operations. */
(*sb->end_transaction)();
```

In case of a file system that does not support transactions, both function pointers point to functions that do nothing. However, if transactions are supported, the implementation of a physical file system might acquire some implementation specific lock in start_transaction that is released in end_transaction later on. This works correctly in practice if the construction of the super block guarantees that end_transaction matches start_transaction.

Unfortunately, our analysis cannot detect this reliably at the moment. This means that the analysis assumes that the call to start_transaction may or may not acquire a lock depending on the function that is actually called. Likewise, the call to end_transaction may or may not release that lock. There is currently no way to detect that the end_transaction function that in fact releases the lock is only called if the previous call to start_transaction acquired it.

The current implementation does not have a solution for this problem. It might be possible to define sets of functions for different function pointers in the same structure. The first call to one of those functions would determine which set of functions is used for future calls to function pointers in the same structure. A solution to this problem would probably allow us to extend the methods described in this work to object oriented programming languages.

7.4.2 Flags

Some functions have a flag argument that determines the way the function behaves. E.g. the lockmgr function in NetBSD that has already been used in Figure 7.12 drops the lock if and only if the flag LK_INTERLOCK is set in the second argument. While it is possible to represent this fact with a normal function description, the dependency on the flag is not detected automatically during analysis. If a flags argument actually affects the behavior of a function in a way that is relevant to the analysis, a hint must be given. This can be achieved by starting the analysis with two predicates that differ only in the value of the relevant bit in the flags argument.

7.4.3 Non-Static Fields

In some cases, a lock that is modified by a function is only reachable via several levels of indirection from the parameters of the function. E.g. consider a function like the following:

```
void doit (struct task * p) {
        spin_lock (&p->mm->page_table_lock);
        /* ... */
        spin_unlock (&p->mm->page_table_lock);
}
```

An error is reported for such a function unless mm is a static structure field (see Section 3.7) because p−>mm might change between the lock and unlock operation. Unfortunately, it is possible that a structure field like mm in the above example is not static but while a certain lock is held that protects the field it can only be modified by the control flow that holds the lock. By means of manually provided locking rules, we can guarantee that the relevant lock is held at all accesses to mm and treat the field as if it were static while this is the case. The difference compared to real static fields though is that the current thread itself might change p−>mm between the lock and the unlock operation or the lock that protects the field might be dropped. To make things worse, this can happen in a different function that is called from the function currently being analyzed.

For a limited number of such fields this problem can be solved by keeping track of those fields that are potentially modified by a function. If such a function is called, all field-of relations that are related to this field declaration in the calling function must be invalidated. In case of the **linux-big** configuration special rules like this for about 40 field declarations are needed. Unfortunately, each of these fields must be identified manually as a result of a false positive error report. Additionally, the fact that we must keep track of which functions can modify which of these fields directly or indirectly tends to increase the number of passes that are needed until no more changes to the function descriptions happen.

7.4.4 Complex Synchronization Semantics

In a few very rare cases, the actual semantics are too complex to be expressed within the limited predicates used in this work. The RCU architecture[33] in the Linux kernel sometimes results in such a situation.

7.5 Summary

In this chapter we have described the case studies that have been conducted. Time measurements and detailed results for three different systems configurations of significant size have been presented. Within a single system the time required for the analysis of a function has been observed to be almost linear in the function size. Additionally, examples of bugs in theses systems have been presented that were uncovered by the analysis. Finally, we also gave examples of constructs where the current analysis methods yield results that are not accurate enough.

Altogether, the case studies presented in this chapter show that conducting an analysis with the methods developed in this work is both feasible and useful in finding bugs.

Chapter 8

Summary and Perspective

Bugs are a very widespread problem in today's software systems and many methods to find and prevent them have been proposed and are actively in use. These methods range from classical testing to verification with the help of a formal calculus. The latter is not feasible for a moderately large software system at least with today's technologies while the former cannot show the absence of bugs.

Multi-threaded systems further complicate the situation for both formal proofs and testing: In a formal proof one must make sure that the effects of concurrent threads do not influence the correctness of the proof. Testing techniques are still valid but the significance of a passing test is reduced considerably in multi-threaded environments.

Therefore, supplementary techniques to avoid bugs in multi-threaded systems are of great importance. In the introduction, classes of bugs have been described by means of characteristics that influence the applicability of bug prevention techniques. Bugs that are hard to reproduce while potentially having severe consequences for the integrity of critical data are identified as suitable candidates for techniques other than testing. It is argued that multi-threaded systems are particularly prone to this kind of bugs.

Given that a complete formal proof of correctness is often not feasible, this work has presented a sound and conservative analysis algorithm suitable for multi-threaded systems. The main focus of the case studies conducted in the context of this work is on operating system kernels written in the C language. However, the methods are in principle applicable to other types of systems written in other programming languages as well. The prerequisites that a system must have were described in Chapter 2. In particular the system's source code is given in the form of decorated abstract syntax trees. Additionally, the syntax trees must adhere to effective type and aliasing rules that are similar to those required by the C Standard[25].

The core analysis algorithm depends on information that can mostly be obtained with the help of traditional analysis techniques that are well known in the context of optimizing compilers. Among other things this information can be used to identify variables and other memory locations that can actually contain non-volatile information about the state of the current thread. In turn, this information is used to identify so-called multi-thread ready predicates (sets of data space states) which describe the state of a single thread in a way that is independent of the execution of concurrent threads. This allows for a conservative analysis of a single thread such that the results obtained are still valid in the presence of arbitrary concurrent threads. In order to extend the analysis methods beyond a single block of code in an efficient way, function descriptions are used to give aggregate descriptions of the effects of a function.

Furthermore, it is argued that the precision of the general analysis method is sufficient to obtain meaningful results. This is supported by case studies conducted on different versions of the Linux and NetBSD kernels. In these case studies, the analysis method has been applied to synchronization problems and we have been able to find several bugs in these systems. Apart from that, the case studies show that it is actually feasible to conduct a conservative analysis as described in this work on large multi-threaded real life systems. Due to its conservative nature, the analysis is also meaningful for the quality of the software system if no bugs are actually found.

Contributions

Recapitulating, the main contributions of this thesis are:

- the identification of semantic restrictions that enable an accurate and conservative analysis of multi-threaded systems.
- a solution to avoid the linking of structure types across translation unit boundaries by replacing it with the linking of field declarations.
- adaption and customization of well known techniques for alias analysis to the resolution of calculated calls and optimization of these methods for large systems consisting of multiple translation units.
- the observation that many structure fields do not change after initialization and that this information is useful in the analysis of multi-threaded systems.
- the definition of a restricted set of predicates and efficiently implemented operations on these predicates as the main tool to describe the state of the system.
- the definition of multi-thread ready predicates and sufficient criteria that can be used to show that a predicate is multi-thread ready.

- an algorithm that can be used to obtain conservative and accurate descriptions of the effects of a block of code in a multi-threaded system.
- the observation that the results of the aforementioned algorithm can be used to obtain function descriptions and the use of these function descriptions to extend the results of the analysis to the inter-procedural case.
- an application of this general analysis approach to the validation of synchronization semantics.
- case studies that show the feasibility of the approach for systems with several hundred source files and its usefulness by finding several long standing bugs in widely used software systems.

Perspectives

Objects and Methods

Currently the analysis does not fully support systems written in object-oriented programming languages. Most of the syntactic sugar introduced by object-oriented languages is hidden from the analysis by the requirement that the program is given as a normalized AST. Nonetheless, dynamically bound object methods pose a problem. Currently, the only option is to treat these calls like function pointers. This assumes that the binding in different method calls happens independently. This is often not precise enough because the binding assumed for the first method call on an object usually determines the binding for all subsequent method calls on the same object.

It should be possible to maintain a list of all virtual function tables that can occur within the system. The method chosen for a method call on an object would then limit the set of virtual function tables for that object to those consistent with the method called.

Parallel and Incremental Analysis

While the software systems being analyzed can be parallel, the analysis itself is completely sequential. While some sequence points in the analysis cannot be avoided, it should be possible to accelerate the analysis on modern multi-processor systems.

Additionally, a small change in the source code requires a completely new analysis. This is suboptimal because a large part of the information that is recalculated is redundant. This is particularly important because it also applies to changes to the compile-time configuration. With some effort, it could be possible to start with the analysis of a core part of the system and additional modules could be added one at a time with only small changes to the information obtained from the initial 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] Frances E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, 1970.

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[5] Takao Asano and Tomio Hirata. Edge-Deletion and Edge-Contraction Problems. In *STOC '82: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 245–254, New York, NY, USA, 1982. ACM Press.

[6] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, TX, USA, 1992.

[7] David Callahan, Alan Carle, Mary Wolcott Hall, and Ken Kennedy. Constructing the Procedure Call Multigraph. *IEEE Trans. Softw. Eng.*, 16(4):483–487, 1990.

[8] G. J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105, New York, NY, USA, 1982. ACM Press.

[9] Huo Yan Chen, Yu Xia Sun, and T. H. Tse. A Strategy for Selecting Synchronization Sequences to Test Concurrent Object-Oriented Software. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, page 198, Washington, DC, USA, 2003. IEEE Computer Society.

[10] Andy Chou, Benjamin Chelf, Dawson Engler, and Mark Heinrich. Using Meta-Level Compilation to Check FLASH Protocol Code. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 59–70, New York, NY, USA, 2000. ACM Press.

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[12] Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 1990.

[13] 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.

[14] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, New York, NY, USA, 2002. ACM Press.

[15] Edsger W. Dijkstra. *Co-operating Sequential Processes, F.Genuys (ed.), Programming Languages: NATO Advanced Study Institute.* Academic Press, London, 1968.

[16] Edsger W. Dijkstra. The Humble Programmer. *Commun. ACM*, 15(10):859–866, 1972.

[17] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

[18] GNU Compiler Collection Online Documentation. Reporting Bugs. `http://gcc.gnu.org/bugs.html#dontwant` (last viewed 30 Apr 2007).

[19] Dawson Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.

[20] Dawson Engler, David Yu Chen, and Andy Chou. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[21] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Problems. In *STOC '74: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pages 47–63, New York, NY, USA, 1974. ACM Press.

[22] Laurie J. Hendren, Chris Donawa, Maryam Emamim, Guang R. Gao, Justiani, and Bhama Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Springer-Verlag, LNCS 757, 1993.

[23] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

[24] Coverity Inc. Company homepage. `http://www.coverity.com` (last viewed 30 Apr 2007).

[25] International Organization for Standardization. *ISO/IEC 9899 Programming Languages – C*, 1999. ISO/IEC 9899.

[26] International Organization for Standardization. *ISO/IEC 9126/1 Software Engineering – Product Quality – Part 1: Quality Model*, 2001. ISO/IEC 9126/1:2001.

[27] S. C. Johnson. Lint, a C Program Checker. Unix Programmer's Manual, AT&T, Bell Laboratories, 1978.

[28] Matt Kaufmann and J. S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Trans. Softw. Eng.*, 23(4):203–213, 1997.

[29] Donald E. Knuth. Frequently asked questions. `http://www-cs-faculty.stanford.edu/~knuth/faq.html` (last viewed 30 Apr 2007).

[30] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation Exploitation in Error Ranking. *SIGSOFT Softw. Eng. Notes*, 29(6):83–93, 2004.

[31] John M. Lewis and Mihalis Yannakakis. The Node-Deletion Problem for Hereditary Properties is NP-Complete. *J. Comput. Syst. Sci.*, 20(2):219–230, 1980.

[32] David Maier. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM*, 25(2):322–336, 1978.

[33] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[34] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*, 2004(117):3, 2004.

[35] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[36] L.F. Menebra and Ada Augusta, Countess of Lovelace. Sketch of the Analytical Engine invented by Charles Babbage with Notes by the Translator Ada Augusta, Countess of Lovelace. *Scientific Memoirs, vol. 3*, 1843.

[37] Jason Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the GCC Developers Summit May 25–27, 2003, Ottawa, Ontario Canada*, pages 171–193, 2003.

[38] Martin Middendorf. More on the Complexity of Common Superstring and Supersequence Problems. *Theor. Comput. Sci.*, 125(2):205–228, 1994.

[39] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability: Measurement, Predication, Application.* MacGraw-Hill, New York, 1987.

[40] Glenford J. Myers. *Methodisches Testen von Programmen.* Oldenbourg Verlag, München Wien, 2001.

[41] Paul E. Black, Hellen Gill and W.Bradley (co-chairs), Elizabeth Fong (editor). *NIST Special Publication 500-262: Proceedings of the Static Analysis Summit.* National Institute of Standards and Technology, Boston, MA, USA, 2006.

[42] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[43] Kari-Jouko Räihä and Esko Ukkonen. The Shortest Common Supersequence Problem over Binary Alphabet is NP-complete. *Theor. Comput. Sci.*, 16:187–198, 1981.

[44] Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Compass '95: Proceedings of the 10th Annual Conference on Computer Assurance*, pages 151–162, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.

[45] Barbara G. Ryder. Constructing the Call Graph of a Program. *IEEE Trans. Software Eng.*, 5(3):216–226, 1979.

[46] Helmut Seidl, Varmo Vene, and Markus Müller-Olm. Global Invariants for Analyzing Multi-Threaded Applications. In *Proceedings of the Estonian Academy of Sciences Physics and Mathematics*, volume 54, pages 413–436, 2003.

[47] Bjarne Steensgaard. Points-to Analysis by Type Inference of Programs with Structures and Unions. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 136–150, 1996.

[48] Jan Tretmans. Testing Concurrent Systems: A Formal Approach. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65, London, UK, 1999. Springer-Verlag.

[49] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[50] Larry Wall. *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[51] Niklaus Wirth. The Programming Language Oberon. *Software – Practice and Experience*, 18(7):671–690, 1988.

Zusammenfassung

Softwarefehler sind ein weit verbreitetes Problem in modernen Softwaresystemen. Die zu Ihrer Vermeidung vorgeschlagenen Methoden reichen von klassischen testbasierten Ansätzen auf der einen Seite bis hin zu Korrektheitsbeweisen mit Hilfe eines formalen Kalküls auf der anderen Seite. Allerdings ist letzteres für größere Softwaresysteme mit den heute zur Verfügung stehenden Mitteln nicht praktikabel, während mit ersterem nicht die Abwesenheit von Fehlern gezeigt werden kann.

In parallelen Systemen kommen bei beiden Ansätzen weitere Probleme hinzu: Bei einem formalen Korrektheitsbeweises müssen zusätzlich die möglichen Auswirkungen von parallel laufenden Threads im Beweis berücksichtigt werden. Testbasierte Ansätze hingegen sind zwar auch in parallelen Systemen sinnvoll und möglich, allerdings reduziert sich die Aussagekraft eines fehlerfrei ablaufenden Tests in solchen Systemen.

Daher sind in parallelen Systemen zusätzliche Techniken zur Vermeidung von Fehlern notwendig. In der Einleitung zu dieser Arbeit wurden zunächst Eigenschaften von Fehlern beschrieben, die einen Einfluss auf die Anwendbarkeit von bestimmten Techniken zur Vermeidung von Fehlern haben. Insbesondere Fehler, die nur schwer zu reproduzieren sind, gleichzeitig aber zu schwerwiegenden Problemen bezüglich der Integrität kritischer Daten führen können, eignen sich besonders für die Anwendung von Techniken zur Vermeidung von Fehlern, die über das Testen hinausgehen. Es wird argumentiert, dass parallele Systeme besonders anfällig für solche Fehler sind.

Basierend auf der Annahme, dass ein vollständiger Korrektheitsbeweis mit Hilfe von formalen Methoden häufig aus praktischen Gründen nicht durchführbar ist, wurden in dieser Arbeit Algorithmen zur Analyse von parallelen Systemen entwickelt, die sowohl konservativ als auch hinreichend genau sind. Die in diesem Zusammenhang durchgeführten Fallstudien konzentrieren sich dabei auf in der Sprache C geschriebene Betriebssystemkerne. Die Methoden selbst sind aber ebenso auf andere Arten von Softwaresystemen anwendbar, auch dann, wenn diese in einer anderen imperativen Programmiersprache geschrieben wurden. Die grundlegenden Voraussetzungen, die ein Softwaresystem für die Anwendbarkeit der vorgestellten Methoden erfüllen muss, sind in Kapitel 2 dargestellt. Insbesondere muss das Programm in Form von abstrakten Syntaxbäumen gegeben sein, und es müssen Regeln bezüglich der effektiven

Datentypen von Objekten im Speicher und bezüglich Aliasing eingehalten werden, wie sie in ähnlicher Form auch vom C-Standard für gültige Programme gefordert werden.

Das Herzstück der vorgestellten Analyse benötigt einige Informationen, die weitgehend mit Hilfe von etablierten, aus dem Compilerbau bekannten Analysemethoden gewonnen werden können. Unter anderem können die so gewonnen Informationen verwendet werden, um Variablen und andere Speicherstellen zu identifizieren, die nicht flüchtige Informationen über den Zustand des aktuellen Kontrollflusses enthalten. Basierend darauf werden Prädikate (Mengen von Zuständen des Datenraums) definiert, die den Zustand eines Kontrollflusses so beschreiben, dass die Beschreibung unabhängig von parallel ablaufenden Kontrollflüssen richtig ist. Als Hilfmittel zur interprozeduralen Analyse werden die Auswirkungen einer Funktion in aggregierter Weise beschrieben.

Darüber hinaus wird argumentiert, dass die vorgestellten Analysemethoden präzise genug sind, um aussagekräftige Ergebnisse zu erhalten. Diese Behauptung basiert auf Fallstudien, die an verschiedenen Versionen des Linux- und NetBSD-Kernels durchgeführt wurden. Dabei wurden die Analysemethoden auf Synchronisationsprobleme angewandt. Im Zuge der Fallstudien konnten einige Fehler in diesen Systemen gefunden und in der Folge beseitigt werden. Darüber hinaus zeigen die Fallstudien allgemein, dass es praktisch möglich ist, eine konservative Analyse von tatsächlich eingesetzten parallelen Systemen mit Hilfe der in dieser Arbeit entwickelten Methoden durchzuführen. Dadurch, dass die Analyse konservativ ist, lässt Ihre Durchführung auch dann Rückschlüsse auf die Qualität der untersuchten Software zu, wenn dabei keine Fehler gefunden werden.

Zusammenfassend sind die wesentlichen Beiträge dieser Arbeit die folgenden:

- Die Identifizierung von semantischen Einschränkungen, die eine konservative und dennoch hinreichend genaue Analyse von parallelen Systemen erlauben.

- Eine Lösung, um das Linken der Deklarationen von Strukturdatentypen über Modulgrenzen hinweg zu vermeiden. Statt dessen werden lediglich die Deklarationen von Strukturfeldern gelinkt.

- Die Übertragung und Anpassung von existierenden Methoden zur Aliasanalyse an das Problem, indirekte Funktionsaufrufe aufzulösen und die Optimierung dieser Methoden für die Analyse von Softwaresystemen, die aus einer Vielzahl von Modulen bestehen.

- Die Beobachtung, dass sich der Wert vieler Felder in Datenstrukturen nach der Initialisierung nicht mehr ändert und dass diese Information in der Analyse von parallelen Systemen von Nutzen sein kann.

- Die Definition einer eingeschränkten Menge von Prädikaten und der zugehörigen, effizient implementierbaren Operationen als zentrales Hilfsmittel, um den Zustand eines Softwaresystems zu beschreiben.
- Die Definition der Eigenschaft *multi-thread ready* für Prädikate und die Angabe von hinreichenden Kriterien, mit denen gezeigt werden kann, dass diese Eigenschaft für ein Prädikat erfüllt ist.
- Ein Algorithmus, mit dem eine konservative and hinreichend präzise Beschreibung der Auswirkungen eines Codeblocks in einem parallelen System bestimmt werden kann.
- Die Erkenntnis, dass die Ergebnisse dieses Algorithmus verwendet werden können, um die Auswirkungen einer Funktion zu beschreiben. Diese Funktionsbeschreibungen werden genutzt, um den Algorithmus zu einer interprozeduralen Analyse zu erweitern.
- Die Anwendung der Analyse auf die Prüfung von Synchronisationsmechanismen.
- Fallstudien, die die Anwendbarkeit des vorgestellten Ansatzes in Systemen mit mehreren hundert Quelldateien und seine Nützlichkeit durch das Auffinden von einigen teilweise bereits seit längerer Zeit existierenden Fehlern zeigen.