PATRICE LACROIX

RTL-CHECK: A PRACTICAL STATIC ANALYSIS FRAMEWORK TO VERIFY MEMORY SAFETY AND MORE

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en informatique
pour l'obtention du grade de maître ès sciences (M.Sc.)

Faculté des sciences et de génie
UNIVERSITÉ LAVAL
QUÉBEC

SEPTEMBRE 2006

Résumé

Puisque les ordinateurs sont omniprésents dans notre société et que, de plus en plus, nous dépendons de programmes pour accomplir nos activités de tous les jours, les bogues peuvent parfois avoir des conséquences cruciales. Une grande proportion des programmes existants sont écrits en C ou en C++ et la plupart des erreurs avec ces langages sont dues à l'absence de sûreté d'accès à la mémoire. Notre objectif à long terme est d'être en mesure de vérifier si un programme C ou C++ accède correctement à la mémoire malgré les défauts de ces langages.

À cette fin, nous avons créé un cadre de développement d'analyses statiques que nous présentons dans ce mémoire. Il permet de construire des analyses à partir de petits composants réutilisables qui sont liés automatiquement par métaprogrammation. Il incorpore également le modèle de conception (design pattern) du visiteur et des algorithmes utiles pour faire de l'analyse statique. De plus, il fournit un modèle objet pour le RTL, la représentation intermédiaire de bas niveau pour tous les langages supportés par GCC. Ceci implique qu'il est possible de concevoir des analyses indépendantes des langages de programmation.

Nous décrivons également les modules que comporte l'analyse statique que nous avons développée à l'aide de notre cadre d'analyse et qui vise à vérifier si un programme respecte les règles d'accès à la mémoire. Cette analyse n'est pas complète, mais elle est conçue pour être améliorée facilement. Autant le cadre d'analyse que les modules d'analyse des accès à la mémoire sont distribués dans RTL-Check, un logiciel libre.

Abstract

Since computers are ubiquitous in our society and we depend more and more on programs to accomplish our everyday activities, bugs can sometimes have serious consequences. A large proportion of existing programs are written in C or C++ and the main source of errors with these programming languages is the absence of memory safety. Our long term goal is to be able to verify if a C or C++ program accesses memory correctly in spite of the deficiencies of these languages.

To that end, we have created a static analysis framework which we present in this thesis. It allows building analyses from small reusable components that are automatically bound together by metaprogramming. It also incorporates the visitor design pattern and algorithms that are useful for the development of static analyses. Moreover, it provides an object model for RTL, the low-level intermediate representation for all languages supported by GCC. This implies that it is possible to design analyses that are independent of programming languages.

We also describe the modules that comprise the static analysis we have developed using our framework and which aims to verify if a program is memory-safe. This analysis is not yet complete, but it is designed to be easily improved. Both our framework and our memory access analysis modules are distributed in RTL-Check, an open-source project.

Avant-propos

Au cours des dernières années, j'ai eu le privilège de côtoyer des gens extraordinaires à l'Université Laval. J'ai connu certains en classe, d'autres dans le cadre de travaux d'équipe ou lors de sorties ; cet entourage a été à l'origine d'innombrables discussions enrichissantes. Au Chapitre ACM et dans l'organisation du Centinel, j'ai eu la chance de rencontrer un grand nombre de personnes dynamiques et passionnées. Voir l'implication d'autant d'individus motivés par la réalisation de projets diversifiés s'est avéré une source d'inspiration inestimable. C'est grâce à tous ces gens que mon passage à l'Université a été si agréable et que j'en conserve un si bon souvenir.

Aujourd'hui, ma maîtrise tire à sa fin et je tiens à remercier spécialement le professeur Jules Desharnais, mon directeur de recherche, pour m'avoir permis de travailler sur un projet fascinant et m'avoir si bien conseillé au cours des dernières années. Je remercie aussi tout le personnel dévoué du Département d'informatique et de génie logiciel ; que ce soit les professeurs, les chargés de cours, la direction, le personnel administratif ou technique, ils ont tous un rôle important à jouer pour faire du Département un milieu d'apprentissage stimulant. Je remercie également le Conseil de recherches en sciences naturelles et en génie du Canada pour l'appui financier dont j'ai bénéficié pendant ma maîtrise.

En terminant, je désire exprimer ma plus profonde reconnaissance à mes parents. Ce sont eux qui m'ont inculqué le goût d'apprendre et ils m'ont toujours encouragé dans ce que j'ai entrepris. Je les remercie de tout coeur pour tout.

À Lise et Serge,
mes parents,
pour tout.

The Guide *is definitive.*
Reality is frequently inaccurate.

Douglas Adam
The Restaurant at the
End of the Universe

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programs are not always apparent, but they are everywhere in our lives. We use them for entertainment; they are obviously an essential component of video games, but they are also very important for the creation and distribution of movies, music and books. Our economy depends on them; whether it is to automate manufacturing or to support the digital communications employed by businesses around the globe, they are unavoidable. We even rely on them in life-critical situations, e.g. in public air transportation, for emergency services and for medical treatments.

More often than we would like, programs have bugs. Their consequences vary greatly depending on the context in which they appear. When playing a video game, a bug can be frustrating. In a factory, a bug can cost millions of dollars in lost productivity. In a 911 call center, a bug can cost lives.

Moreover, in hostile environments such as the Internet, there are often malevolent people who are actively looking for bugs and ways to exploit them. New attacks and viruses are reported almost every day by web sites that specialize in tracking these threats.

In this context, more and more often we want assurance that the programs we use work as expected, i.e., they have no bug. However, it is difficult to prove that a program is correct, even under the best conditions. This task is further complicated by the fact that many languages do not even offer basic guarantees such as memory safety about the run-time behavior of their programs.

Static program analysis is an approach that can be used to obtain these guarantees and to prove that a program is correct with respect to its specification. This thesis is about static analysis of programs written in programming languages that are not memory-safe.

1.1 Memory Safety in C and C++

C and C++ are indisputably the two most popular programming languages that are not memory-safe. When a programmer uses them, he has complete control over memory. This is an important feature for writing the kernel of an operating system and device drivers, but for a normal application it causes more problems than it solves.

The problem is that when a program has a bug related to memory access, the bug often causes it to blindly overwrite some of its data instead of signaling the error and aborting. Depending on the nature of what is overwritten, the program may continue with corrupted data or crash before it completes.

Most often, invalid memory accesses are caused by a wrong assumption about some data. For example, a *buffer overflow* occurs when an array is smaller than the size the program assumes it has. The wrong assumption can also be that a pointer is initialized, that it is not null or that it points to data of a given type.

Sometimes, an invalid memory access can be traced to an error on the part of a programmer who confuses what some code actually does with what he expects it to do. This may be caused by ignorance, negligence, bad design or bad documentation of the interface of a function. For example, if a programmer expects `strncpy()` (from the C library) to always create a null-terminated string, it might induce an invalid memory access. Another common error is to pass a user-controlled string as the first argument to `printf()` instead of a proper format string. This is a *format string vulnerability* which can have disastrous consequences [Lac03].

In memory-safe languages such as Java and C#, these problems cannot happen. Such languages guarantee that they will catch every memory access error during the execution of a program. This ensures that the program will be stopped instead of continuing with corrupted data.

In C and C++, a memory access error can cause a program to generate a wrong result. Worse, it can often be exploited by malevolent people to gain unauthorized access to a computer or data [LD05].

To avoid these problems, one obvious solution is to avoid using unsafe languages such as C and C++. This is often a good idea, but it is not always possible for some low-level applications. Also, there is an immensely large collection of existing C and C++ programs which are useful and cannot be rewritten in another language overnight. For example, Debian 3.1, an open-source operating system, contains 170 millions lines of C and C++ source code [AIGBRMHT05].

Since C and C++ will continue to be in wide use for the foreseeable future, we are looking for means to mitigate the problems associated with the absence of memory safety in these languages.

1.2 Goals of the Project

Our main goal is to develop a tool that helps determining statically, i.e., without executing it, whether a C or C++ program is memory-safe or not. Because this problem is undecidable, it is not possible to write a tool that computes the right answer in all situations. However, it is possible to create a tool that gives a safe diagnostic in the sense that it will report at least one error for all programs that access memory incorrectly. We want to produce such a tool.

Nevertheless, we are attacking a difficult problem. There is an enormous amount of work ahead of us before we will be able to produce good diagnostics about the memory safety of C and C++ programs automatically. Since the problem is far too broad to be solved during a master's program, one of our objectives is to produce a tool that will continue to be useful and easy to improve after this period.

We specifically exclude from our project the usage of run-time checking to detect memory access errors. However, it is not unlikely that our tool or an extension of it could ultimately generate a diagnostic precise enough to allow automatic insertion of dynamic verifications into an unsafe program to render it memory-safe.

1.3 Contributions

Our main contributions can be found in RTL-Check[1], an open-source tool we created to analyze programs statically. This tool includes:

- An object model to represent RTL, the low-level intermediate representation for all languages supported by GCC, the GNU Compiler Collection;
- A static analysis framework that employs metaprogramming to facilitate the creation of possibly complex analyses (for C, C++ and other languages supported by GCC) from small reusable components;
- A promising and extensible analysis which aims to demonstrate memory safety of C programs using our framework;
- A generic method to automatically implement the visitor design pattern in Python programs.

1.4 Overview of the Thesis

Chapter 2 studies the techniques that can be used to detect memory access errors in low-level languages such as C and C++. Chapter 3 discusses the important choices that we made at the start of the project and that affected the development of our tool. Chapter 4 describes our first proof of concept which allowed us to validate some of our design decisions.

The following two chapters constitute the bulk of our contribution. Chapter 5 examines our static analysis framework. This framework makes it possible to build analyses from small reusable components. Chapter 6 details the inner workings of our memory safety analysis, which we created using our framework.

Chapter 7 shows that metaprogramming is a powerful programming technique to solve complex problems and how RTL-Check takes advantage of it. Among other things, this chapter depicts an automated way to implement an improved visitor design pattern. Chapter 8 explains the open source development process that we tried to apply to RTL-Check and the results that ensued from it. Finally, Chapter 9 presents our conclusion.

[1]The RTL-Check home page is at `http://rtlcheck.sourceforge.net/`.

Chapter 2

State of the Art

This chapter presents an overview of the techniques that can be used to protect programs against buffer overflows and memory access errors[1] in low-level languages such as C or C++. A very large number of different approaches have been proposed over the years. Some of them detect the errors themselves, but many only try to prevent their exploitation.

Sections 2.1, 2.2 and 2.3 present the main idea of the approaches based on static analysis, dynamic checks and miscellaneous other techniques respectively. We will explain in more details those that are nearest to the goal of our project, i.e., using static analysis to detect all memory access errors. For more in-depth information about the techniques presented in these sections, see this survey [LD05].

Section 2.4 presents some tools that can help implementing new static analyses.

2.1 Static Analysis

In this section, we describe the main ideas that can be used to protect code against buffer overflows without running it. First, we present general approaches to static analysis that are independent of the property analyzed.

Then, we explain more specific techniques to detect memory access errors. The

[1]"Memory access error" is a more general phrase than "buffer overflow". We use the latter a lot in this chapter because many approaches presented here target buffer overflows specifically.

different analyses are not all equivalent. Some of them guarantee that all errors are found, others do not. Some are completely autonomous; others take advantage of hints from the programmer. They also vary greatly by their speed and their rate of false diagnostics.

We distinguish between two kinds of false diagnostics. A false positive is when an analysis reports an error that cannot happen, and a false negative is when an analysis does not report a possible error. Since our problem is undecidable, every static analysis that always terminates gives a false diagnostic for a certain class of programs. We prefer analyses that never result in false negatives because our goal is to prove that a program is free of memory access errors.

Sections 2.1.2, 2.1.3 and 2.1.5 are based on a report [Lac03] written during the author's undergraduate studies.

2.1.1 General Approaches to Static Analysis

There are a few general approaches to static analysis that are independent of the properties to be analyzed. The following paragraphs present them briefly.

Monotone Data Flow Analysis Frameworks

Kildall, Kam and Ullman [Kil73, KU76, KU77], have established the basis of *monotone data flow analysis frameworks*. With this approach, which we describe in more details in Section 5.2.1, information about the program is kept for each program point during the analysis. This information is an element of a semilattice. Monotone functions on the semilattice describe how an instruction transforms the information from a given program point to the next. An iterative algorithm propagates the information along the control flow graph of the program and the analysis ends when no new information is propagated. The semilattice must satisfy some conditions to ensure that the analysis terminates.

Set-Constraint Based Analysis

In contrast, *set-constraint based analysis* [HJ91b, HJ94, Aik94] does not follow directly the control flow of the program to propagate information. Instead, only one pass over the program is used to extract set constraints. This gives rise to a system of inequations in which each constraint variable corresponds to the set of values a given program variable can take at a given program point. Constraints are created in such a way that any solution to the system of inequation is a safe approximation of the program. The smallest solution is the best approximation. The main limitation of set-constraint based analysis is that it ignores all inter-variable dependencies.

Constraint-Based Type Inference

A related approach is that of *constraint-based type inference* [AW93, Pot00, SP05]. The goal is to assign a type to variables and other syntactic constructions of programs. A type system expresses the relations between syntactic constructions and types. Type systems can be very simple and inspired from the underlying type system of the language, but they can also include more advanced concepts such as subtyping, polymorphic types, conditional constraints, indexed types, inductive types, and guarded algebraic data types. These concepts add expressivity to type systems, which in turn allow expressing more complex properties.

Abstract Interpretation

Cousot and Cousot developed *abstract interpretation* [CC77, CC79, CC92], a general framework for program analysis in which analyses are formally specified as an abstraction of the concrete semantics of a language. In its most common form, it is more general than monotone data flow analysis frameworks because, although it also use semilattices to represent information about a program, the conditions on the semilattices are relaxed a bit; the use of widening operators can compensate. Set-constraint based analyses and type inference algorithms can also be recast in an abstract interpretation framework, as shown in [CC95, Cou97]. One interesting aspect of abstract interpretation is allowing the creation of new analyses from existing ones.

2.1.2 Lexical Analysis

Lexical analysis is not a very powerful static analysis technique. Analyzers using this approach are looking for a sequence of tokens that often pose problems in the source code. For example, a tool can check whether there is a call to `strcpy()` and flag it unconditionally as a potential buffer overflow even if it can be used safely.

Tools [VBKM00, Fla] using lexical analysis have the advantage of being fast because the analysis does not require many calculations. They usually give many false positives, that is to say, they give indications that an overflow is possible when it is not. Also, they never ensure that no overflow is possible.

2.1.3 Annotation-Assisted Lightweight Static Checking

Evans and Larochelle [EL02, LE01] explain a method allowing, among other things, the detection of buffer overflows in C programs with some kind of lightweight static checking. Their method avoids interprocedural analysis with the use of annotations, also called semantic comments, added by the programmer to the prototype of the functions. These annotations (preconditions and postconditions) form a kind of contract for the behavior of a function. It is possible to check if the contract is respected by any party (the implementation and the callers) independently of others. Thus, with this approach, a whole program can be analyzed without resorting to more costly interprocedural analysis.

To check if a function can cause an overflow, it is assumed the values received as input respect the preconditions expressed in the annotations. Under these conditions, the analysis tries to verify whether the code ensures that the postconditions are met at the end of the function. If it does not, the programmer is informed.

When the analysis encounters a function call, it checks whether the parameters passed respect the preconditions of the called function and it assumes that, after this point, the postconditions are met. If a precondition is not satisfied for a function call, the programmer is warned.

Splint is a tool using this approach. The analysis it implements has the advantage of being very fast. However, it can cause both false positives and false negatives. Splint is free software, but it would be difficult to modify it so that there are never false positives. The main idea that we will keep from Splint is that we can avoid costly interprocedural analysis by defining contracts between a function and its callers.

2.1.4 Taint Analysis

Alexander Ivanov Sotirov [Sot05] uses *taint analysis* to reduce the number of false positives that would be reported if the bounds of arrays were checked using a simple value range propagation algorithm. A variable is considered *tainted* if its value can be controlled by the user of the program. Annotations are used to specify whether a function returns tainted values.

This analysis reports errors only when a function that manipulates an array is called with a size argument that is tainted. The idea is that if the user can force the program to use an arbitrary value as the size of an array, he can cause an overflow. This analysis has been implemented in GCC and it can generate both false positives and false negatives. This is incompatible with our goal of detecting all array access errors.

2.1.5 Abstraction of Data

Wagner et al. [Wag00, WFBA00] consider null-terminated strings as an abstract data type manipulated by the functions of the C library. Every string is modeled by a pair of integers indicating its allocated size and its real length, i.e., the position of the null character that terminates the string. For each manipulation of a string, one or more constraints are associated to these two integers. Manipulations of integers also generate constraints.

After all the constraints have been extracted from a program, they form a constraint system that must be solved. The result is, for each string, a range of possible values for the memory allocated and another one for the size of the string. To be sure there is no overflow, the upper bound of the range for the size must be smaller than or equal to the lower bound of the range for allocated memory.

BOON is a tool that implements this technique. The analysis done to generate constraints has the advantage of being fast, but it lacks precision. For example, it does not take the control flow into account, i.e., control structures and the order of instructions are ignored. Also, this tool considers there is only one instance of each variable composing a structure instead of one for each instance of the structure. This can greatly enlarge the range of possible values seen by BOON as opposed to the real values it can take. Because of this, the number of false positives is rather high.

A more important problem with BOON is that it does not detect all overflows. First, it only detects overflows on strings, not on arbitrary arrays. Moreover, only manipulations of strings by the functions of the C library are considered and not those modifying a string as a character array or using pointer dereferences. The handling of pointers also poses problems because it is much simplified. For instance, BOON largely ignores the fact that two pointers can be aliases to the same string at the same time.

It also totally ignores doubly indirected pointers, arrays of pointers, function pointers and `union`.

All these deficiencies prevent the detection of some buffer overflows. They are not intrinsic to the approach but, since only some parts of BOON are free software, it is not an ideal starting point to build a better analysis.

The important thing to remember from BOON is the modeling of strings by pairs of integers. Such an abstraction of strings is easy to deal with. It would be much more difficult to implement a logic that can express the position of the first null character in a given array using more primitive constructs. Other abstractions could be created to deal with other kinds of data.

2.1.6 Context-Sensitive Analysis

Ganapathy et al. discuss [GJC+03] their improvements to the analysis described in the previous section. From our point of view, the most interesting improvement is that they do context-sensitive analysis, i.e., they take the context of the caller into account when analyzing a function. Context-sensitive analysis is important to reduce the number of false positives because many programs cannot be proved correct using only context-insensitive analysis.

They use two different techniques to achieve context-sensitive analysis. The first one is inlining constraints at call sites, but this cannot work with recursive function calls and a function must be analyzed for each call site independently. The second one is generating *summary constraints* for functions, i.e., a set of constraints that can be instantiated at every call site with elements taken from the call context. Both of these approaches have their advantages and their disadvantages.

2.1.7 Approximating Contracts with Integer Analysis

Dor et al. [DRS03] present an approach that can detect all string manipulation errors in C programs, i.e., without false negatives. They show how to reduce the problem of checking string manipulations to that of determining whether assertions can be violated in a program that manipulates only integers. This analysis is not interprocedural; it works on one procedure at a time. It can use annotations if they are available, but the authors also show how it is possible to compute an approximation of the strongest postcondition and the weakest liberal precondition from the program that manipulates integers. Thus, the contract of many procedures can be approximated automatically.

The tool that implements this analysis is called CSSV, but it is not publicly available. The results presented show that, in some cases, the time and memory required for the analysis and the number of false positives can be relatively high. However, it is difficult to tell how representative these results are, because they are based the analysis of on less than 1000 lines of code.

In any case, this analysis is very interesting both because it can generate a contract automatically and because it detects all string manipulation errors. However, it does not detect buffer overflows on data other than strings.

2.1.8 Statistical Belief Analysis

The tool presented by Xie et al. [XCE03], ARCHER, allows checking millions of lines of existing code because it is fast and it is tuned to suppress common sets of false positives. On the other hand, it makes absolutely no guarantee about false negatives, so the analysis itself is not that interesting to us. Also, the tool is not publicly available.

However, there are some elements of the analysis that are interesting. Contrary to the approach described in Section 2.1.7, where the implementation of a procedure is used to infer its contract, this one uses the context in which a procedure is called. This method is known as *statistical belief analysis* [ECH+01]. For procedures that receive a pointer to an array, it tries to compute the probability that each of its other parameters represents the size of the array. For this, it counts, among all call sites, how many times a given parameter is less than or equal to the size of the array (successes) and how many times it is more than the size of the array (failures). With the rate of successes and failures, it is possible to compute the probability that a given parameter represents the size of the array, and a contract for the procedure can be derived from that.

Statistical belief analysis could be one of many possible methods used to guess the contract of procedures in an analysis that never generates false negatives. The contract would still have to be verified at every call site and in the implementation of the procedure.

2.1.9 Abstract Program Representation

Livshits and Lam [LL03] present an abstract representation, IPSAA, an analysis that helps transforming a C program into this representation and a tool that tries to detect buffer overflows from this representation. IPSSA (InterProcedural Static Single Assignment) captures intraprocedural and interprocedural definition-use chains for both directly and indirectly accessed memory locations.

The main part of the analysis is a hybrid pointer alias analysis. The first component of this analysis is precise and it is path and context sensitive. It is used to track locations accessed from parameters and local variables by *simple paths*, that is to say, without iterated dereference or field access. The second component is more efficient but less precise because it is flow and context insensitive. It is used to track all other locations.

However, this analysis uses an unsound assumption, which is that pointers passed into a procedure and locations that can be accessed by applying simple paths to these pointers are all distinct from each other. Since this assumption is not verified, there can be false negatives. Still, this approach might be another way to guess the contract of a procedure.

2.1.10 Domain-Specific Abstract Interpretation

ASTRÉE [CCF+05, Mau04] is a tool specifically targeted at the verification of embedded real-time software written in C. It cannot verify all C programs, e.g., it supports neither dynamic memory allocation nor the use of the C library. Since embedded real-time software usually does not use these features, these limitations are often acceptable. These restrictions make it possible to verify many programs completely automatically, i.e., without annotation or help from the programmer. There are no false negatives.

ASTRÉE is not publicly available. It uses abstract interpretation and it implements many abstract domains; many of them where developed specifically to abstract some constructs that are often present in embedded real-time software. When running ASTRÉE on a given program, it is possible to specify which abstract domain to use for the analysis. If there are too many, the analysis might run for too long, and if one is missing, it might not be possible to verify that the program is correct.

The main thing to remember about this approach is that when we are dealing with a restricted class of programs, it is often possible to create abstractions that are precise enough to verify many programs, yet not too precise to the point of making the analysis too slow.

2.1.11 Predicate Abstraction and Abstraction Refinement

People at Microsoft Research developed SLAM [Bal04, BCDR04], a tool that implements a very interesting static analysis. The goal of this tool is to verify that drivers for the Windows operating system contain no error. It is currently available only as binary software in a beta version of Windows Driver Foundation, but BOOP [Wei03] is a publicly available open-source reimplementation of this tool.

The idea is to create a Boolean program, i.e., a program that manipulates only Boolean variables, from the C program to be analyzed and a set of predicates. This is *predicate abstraction*. The resulting Boolean program is an easier to analyze abstraction of the original C program. If a property holds on the abstraction, it means that it also holds on the original program.

However, if a property does not hold on the abstraction, it does not mean anything about the original C program. In this case, the analysis produces an execution trace in which the property does not hold on the abstraction. A theorem prover then checks whether the property holds on the original program for this specific trace. If it does not, an error is reported. If it does, the theorem prover identifies a new predicate to discriminate between executions where the property holds and those where it does not.

The new predicate is then added to the set of predicates used to build the abstraction of the original program, and the process starts over. This process is called *abstraction refinement*. An important problem with this process is that it does not ensure that the analysis terminates. Nevertheless, this approach can certainly give good results for many programs and we will keep it in mind.

2.2 Dynamic Checks

This section presents an overview of the different approaches that can be used to detect buffer overflows or their exploitation at run time. Some of them guarantee that they will work in all situations and some do not, either because they target some kind of overflow in particular or because they target some kind of exploitation techniques.

All the techniques presented here introduce an overhead at run time that varies from negligible to a slow-down factor of two or more.

2.2.1 Run Time Bounds Checking

Bounds checking, i.e., verifying whether an index is inside the bounds of an array before using it, is the most obvious form of dynamic check. The problem is that the C and C++ languages allow the use of pointers in contexts completely disconnected from the declarations (or allocations) to which they refer. It renders run-time bounds checking difficult for a compiler, but not impossible. There are many ways to achieve it and they are not all equivalent.

One way to implement array bounds checking is to change the representation of pointers so that they include information about the memory area they refer to. Such modified pointers are sometimes called bounded pointers. They allow the most precise detection of overflows and pointer error, i.e., with neither false positive nor false negative. This approach is presented in [ABS94, XDS04]. Since bounded pointers are larger, they are not compatible with some programs that assume that pointers have the same size as integers. An alternative is presented in [Jon95, JK97] to overcome this problem. Another alternative uses static analysis [NMW02] to avoid overhead when the compiler knows that the pointer is used safely.

The approach presented in [LC02] is less precise because it checks only that functions of the C library do not copy more data than were allocated for the destination memory area. The problem is that when a memory block is allocated, it can be used for a structure that contains one array and some other data. In this case, an overflow can overwrite the other data silently. It is possible to achieve similar results for all memory accesses by changing the way memory is allocated [Ele]. A variant of this technique ignores completely the size of actual data and compares directly the size of the source and destination memory areas [HB03]. This helps detecting more errors during the testing phase, but it also introduces false positives.

Another technique allows checking that every memory access [HJ91a] is in a memory area correctly allocated. Once again, this is not as precise as real array bounds checking. One advantage is that it does not require the source code. An alternative uses static analysis [YH03] to minimize the overhead.

2.2.2 Protecting the Return Address

The return address is important because it indicates where the execution continues when a procedure ends. If an attacker can overwrite it, he can instruct the program to continue its execution in code he chooses arbitrarily [Ale96]. This is why many techniques try to protect the return address more specifically.

One of them consists in modifying the compiler so that it inserts a *canary* [CPM$^+$98] before the return address. This is an arbitrary value that is checked before the control returns to the calling procedure. If the canary has changed, the program is aborted because an overflow might have overwritten it and the return address.

If the compiler cannot be modified, the functions that are known to be the cause of many overflows can be modified [TS01] to explicitly check that they will not overwrite a return address.

An alternative is to use an entirely different stack [Sta, CH01] for return addresses and local variables. Sometimes, it is also possible to implement this protection in the operating system [FS01] instead of the compiler.

2.2.3 Detecting Abnormal Behavior

Here we present techniques that do not detect overflows directly. Instead, they try to detect abnormal behavior that may be caused by overflows. The idea is that if an attacker uses a buffer overflow to take over control of a program, it will probably change its observable behavior.

The first thing that can be observed when running a program is the sequence of system calls it performs. One can verify that it corresponds to what the program is supposed to do. There are many ways to determine whether a sequence of system calls is acceptable or not. A simple approach is to run the program and extract the set of system calls [Pro03]. A more precise specification of a program can be built manually [SU99], by automatic learning [SBDB01] or by static analysis [Wag00].

Using a special interpreter, it is also possible to monitor every control flow transfer [KBA02] in a program to ensure that it is appropriate, as defined by a given policy. For example, it is possible to ensure that a return instruction returns to an instruction that follows a call, or that a library is called only through a declared entry point. This is called *program shepherding*.

The two previous techniques monitor some aspect of the execution of the program, but it is also possible to detect abnormal data that are inserted in a program. For example, *abstract payload execution* [TK02] checks whether data received from a client contains executable code. If it is the case, it may be a sign that the program is under attack. The attack can be stopped before an overflow is exploited so that no damage is done.

2.3 Other Approaches

Here we present miscellaneous approaches that aim to detect buffer overflows or prevent their exploitation and which do not fit in the previous sections.

2.3.1 Using Libraries with Safer Interfaces

Some functions of the C library do not accept the size of the buffer in which they put data. One can avoid these functions and use safer alternatives. For example, `strlcat()` and `strlcpy()` [MdR99] can replace `strcat()` and `strcpy()` respectively.

2.3.2 Encrypting Pointers

When an attacker exploits an overflow to gain control over a program, one step often requires giving an arbitrary value to a pointer. To avoid this, a compiler can be modified to encrypt pointers [CBJW03, PL02] using a secret key. Without knowing the secret key, which can change at every execution, the attacker can only give a random value to a pointer. It may crash the program, but encryption avoids more important consequences.

2.3.3 Preventing Execution of Arbitrary Code

Many attacks inject arbitrary code before forcing the program to execute it by exploiting a buffer overflow. Some programs need to generate and execute code at run time to work properly, but many do not. In the latter case, the operating system can ensure that the stack [Sol97b] and the heap [dR03, Mol03, Sta01, PaX00, PaX02] are not executable to thwart these attacks.

2.3.4 Moving the Stack and Libraries

Most attacks doing more than just crashing a program must know the exact, or at least approximate [Ale96], location of the code that is to be executed. The location of the stack [Ket03] and that of shared libraries [Woj01] can be randomized to make many attacks ineffective. Libraries can also be moved to addresses containing a byte having the value zero [Sol97a], which is more difficult to generate for an attacker in many situations.

2.3.5 Reordering Local Variables

A compiler can be modified to reorder [EY00] local variables so that pointers are before buffers on the stack. This way, if an overflow happens, the pointers can still be used safely. This can prevent many attacks from working correctly.

2.4 Existing Tools to Build Static Analyses

In this section, we present existing tools that can help building new static analyses and we evaluate their suitability for verifying the memory safety of a program.

2.4.1 Compilers

A compiler can be a good starting point to construct a new analysis if its internals are well documented and not too complex. This is because it already implements the parsing of the source code into more program-friendly structures and it usually incorporates some kind of static analysis (for optimizations) that can be reused. For example, Vulncheck [Sot05] is a tool that runs inside GCC to help detecting many vulnerabilities in programs. Its analysis uses a combination of taint analysis and value range propagation, but value range propagation is already present in GCC and it is reused as is.

2.4.2 CQual

CQual [CQu] is an open-source tool that allows extending the syntax of C with user-defined type qualifiers and specifying rules on them. CQual then performs qualifier inference checks using constraint-based type inference. This approach has been used to verify some safety properties on C programs [STFW01, FTA02, Fos02], but it is not powerful enough to verify more complex properties such as memory safety.

2.4.3 Banshee

Banshee [Ban] is an open-source tool that automatically generates an efficient resolution engine for a constraint system, given its specification as input. It also supports backtracking and persistence of constraints, thus it can help creating efficient incremental analyses. To build an analysis, one only has to create a program that extracts constraints from source code and feed them to the resolution engine. Banshee was used to implement some points-to analysis [KA05], but it is not powerful enough for more complex context-sensitive analyses. Moreover, being constraint-based, it is not designed to support the implementation of many analyses created using the theory of abstract interpretation.

2.4.4 FIAT

FIAT [HMCCR94] is a framework for interprocedural analysis and transformation designed to implement optimizations during compilation of code. FIAT is not publicly available and our goal is not to optimize code, but this framework has a few interesting features. Firstly, it does not work directly on the code to analyze, but on an abstract representation of it. This allows FIAT to be used in different compilers. Secondly, it provides the logic that is common to all monotone data-flow analyses. Thirdly, an analysis can request the result of another analysis without specifying explicitly the ordering between the different analyses. Those are all useful features when building new analyses.

2.4.5 The Hob System

The Hob system [ZLKR04] and its successor Jahob can verify high-level properties to ensure data structure consistency by applying different verification techniques to different parts of a program. For example, they could use a simple form of static analysis for most of the program, and a theorem prover for a small part of it that is more difficult to verify. They are both open-source, but since they analyze languages that are already memory-safe, they are not compatible with the verification of memory safety in C and C++. However, some of the techniques they use might be useful to verify programs using complex data structures.

Chapter 3

Preliminary Choices

Some choices were made at the start of the project, before writing the first lines of code. They were about the openness of the development, the code that will be analyzed and the implementation language. This chapter discusses why they were made.

3.1 Open Source Development

Very early, we decided to create an open-source [Per98] project to implement our static analysis. This means that anyone can use, modify and redistribute our software. We have many reasons for doing so. The first one is that we want to show to the world the kind of research we are doing. Making our software available to all on the Internet and applying the adage *release early, release often* [Ray00a] contributes to that.

Another goal is attracting competent people in the project so that we can achieve better results faster. This also increases the chances that our software will continue to exist and improve after we stop working on it. This is important because the problem we are working on is hard and we knew from the beginning that we could not do alone everything we wanted in only two years.

A related goal is to help research on static analysis, and particularly its application to the detection of buffer overflows and the verification of memory safety. Often, when a paper describes an approach, no source code is provided. In this case, other researchers lose much time reimplementing it before they can improve it. With an open-source implementation of our analysis, the barrier to entry in the field is smaller.

Table 3.1: Example of an RTL jump instruction.

```
(jump_insn 14 13 15 (nil)
           (set (pc)
                (if_then_else (le (reg 17)
                                  (const_int 0))
                              (label_ref 17)
                              (pc)))
           -1 (nil) (nil))
```

This example shows the instruction of a function that is numbered 14. It is inserted between instructions 13 and 15 (instructions are not always numbered sequentially). It describes a jump to the label number 17 (i.e., the program counter, `pc`, is set) if the value of the register 17 is lower than or equal to the integer 0.

Chapter 8 describes in more details the open source development process we applied to our project and what ensued from it.

3.2 RTL as the Language to Analyze

The C and especially the C++ languages are complex and full of subtleties; we wanted to avoid parsing them and understanding their complete semantics. We knew that reusing some parts of a compiler could help us, so we studied the internals [GCC03] of GCC 3.3.2, the most recent version of the compiler at that time. We learned that there are two different intermediate representations of code used by GCC during the compilation of C and C++: trees and RTL.

Trees are the high-level intermediate representation for the C and C++ languages. They are structures close to the syntax of these languages, so they can save us only the parsing; much of the complexity still remains.

RTL, on the other hand, is the low-level intermediate representation for all the languages supported by GCC. This representation is much closer to an assembly language, yet it remains almost architecture-independent. In RTL, a function body is a sequence of RTL instructions.

Table 3.1 shows an RTL instruction that describes a jump as an example of how

RTL is structured. Every pair of parentheses delimits an *RTL expression* and an RTL instruction is itself an RTL expression. The first element of every RTL expression is the *RTL code*, which defines the meaning of its operands and the expression as a whole.

We decided to base our analysis on RTL because it has the following advantages over direct analysis of C and C++ source code.

- It saves us the parsing of C and C++, two languages with a complex grammar.
- It is language-independent; our analysis will not have to know anything directly related to C or C++, and eventually, we could also analyze other languages supported by GCC such as Java and Fortran.
- It is almost architecture-independent; our analysis will only need to know minimal information that varies depending on the target architecture.
- The semantics of RTL is simpler than that of C and C++ because it has fewer different constructs.
- RTL is not as ambiguous as C and C++. The C and C++ standards have many areas that are designated as *implementation defined*; GCC must disambiguate much of the code that falls in these areas before it is transformed into RTL.

Still, RTL is not perfect for our purpose; it is not completely architecture-independent and it does not contain type information about variables. However, we thought the advantages would outweight the disadvantages.

GCC 4.0 introduced *GIMPLE*, a third intermediate representation between trees and RTL. It was created for the new optimization framework of GCC based on static single assignment [CFR+91]. It might be a better representation than RTL for our purpose because it is designed to be straightforward to analyze and it keeps information about high-level attributes of data types. Since GIMPLE appeared too late for us, we did not study it in depth.

3.3 Python as the Implementation Language

The language we chose to implement our analysis is Python. This is a dynamically typed language that supports both object-oriented and functional programming paradigms.

Table 3.2: A simple Python program.

```
# Class A inherits from class object
class A(object):
    # Constructor with one (explicit) parameter
    def __init__(self, value):
        self._member = value

    # Method without (explicit) parameter
    def output(self):
        print "Class A with value", str(self._member)

# Create an instance of class A
a = A(42)

# Call a method on object a
# This will print: Class A with value 42
a.output()
```

It is not the ideal language for an analysis that will have to run on many large programs because it is not very fast, but Python is certainly a good choice for a prototype that needs to test different directions, because of its dynamic nature and its expressiveness. Chapter 7 demonstrates that these aspects of Python can be useful.

Some Python code will be presented in this thesis. It should be relatively easy to understand for people familiar with object-oriented and functional programming. An important thing to know is that in Python, a method receives the object to which it is applied in its first formal parameter, which is named `self` by convention. This parameter is the equivalent of `this`, which is passed implicitly in languages such as C++ and Java. The constructor of a class is named `__init__` and its first formal parameter is the object to initialize. In Python, variables and members are created automatically at the time of their first assignment; there are no declarations. Table 3.2 shows a simple Python program that defines a class and uses it.

Chapter 4

Proof of Concept

To check whether it was feasible to build an analysis given our preliminary choices, we decided to implement a trivial analysis of RTL using Python. The goal of this analysis was to flag all program points that access memory in a given function.

This chapter describes how we achieved that. In particular, we explain our modifications to GCC for dumping RTL, we describe how RTL is represented in Python and we present our trivial memory access analysis. The code developed for this proof of concept has been published as RTL-Check 0.0.1 and some of it is still in use in the most recent version of RTL-Check[1].

4.1 Dumping RTL from GCC

Since we want to implement our analysis in Python, not inside GCC, we need a way to dump the RTL representation of a program before we can analyze it. GCC already has some options to dump an ASCII representation of RTL at different stages during optimizations. For example, the option "`-dr`" creates a dump file when RTL is first generated. There are two problems with these dumps. First, there are some cases where not all the RTL instructions are dumped. Second, since the dump is in ASCII format, it must be parsed before it can be analyzed; we would like to avoid that, if possible.

To overcome these problems, we created a new binary dump of RTL. We modified

[1]The RTL-Check home page is at `http://rtlcheck.sourceforge.net/`.

the functions `open_dump_file()` and `close_dump_file()` defined in `gcc/toplev.c` so that our binary dump file is produced every time the usual ASCII dump file is requested. This means that we can use the already existing options of GCC to dump a binary representation of RTL at different stages during the compilation.

Our format for the dump file is close to the in-memory structures used by GCC to represent RTL expressions, but we cannot use a blind copy of these structures because they have a size that varies according to the RTL code of each RTL expression and because when an operand is a pointer to another RTL expression, the latter must also be dumped.

In GCC, every possible kind of RTL expression (RTL code) is defined in the file `rtl.def` using the macro `DEF_RTL_EXPR`. This macro as 4 parameters:

- The internal name of the RTL code;
- Its name as string;
- A string that describes the parameters for the expressions of this kind, if any;
- The class of the expressions of this kind.

The third parameter is what allows us to correctly dump RTL in a generic way, without having to encode additional knowledge about every RTL code. The file `dump-rtl.c` is the one implementing the binary dumping of RTL and it is still in use in current versions of RTL-Check.

4.2 Accessing RTL from Python

On the Python side, we created classes to represent dump files (`RtlFile`), functions (`RtlFunction`) and RTL expressions (`Rtx` and its subclasses). We also created a function that instantiates these classes from a given binary dump file. This code can be found in `rtl.py`. It has evolved somewhat since the first version, but much of it is still present in current versions of RTL-Check.

We also had to create a Python version of the low-level definitions from `rtl.def` so that, among other things, we could associate a meaning to the numbers we find in binary dump files. The corresponding Python definitions can be found in `rtldef.py`, which is created automatically using the C preprocessor and the files `rtl.def` and `rtldef.src`. The latter contains a special implementation of the macro `DEF_RTL_EXPR`.

Our classes are more interesting than the C structures used in GCC because they offer a type-safe interface for accessing the operands of RTL expressions. Moreover, having a different subclass for each RTL code allows us to use properties with significant names instead of accessing the operands using their position number. We did not define an `Rtx` subclass for all RTL codes; we defined only the subclasses that were needed for our proof of concept.

We decided to implement the visitor design pattern for our RTL classes to make it easier to create code that goes over the definitions of a function, e.g. to analyze it. Section 7.3 explains how the visitor design pattern works and how we implemented it.

The first use of the pattern was to create a visitor that simply prints an ASCII representation of RTL. It allowed us to check that the binary dump, its interpretation and our classes were all working properly. Since RTL can be dumped at various stages during optimizations, we also used this visitor to determine which one would be best to use for our analysis. We discovered that, as the compilation progresses, more architecture-dependant constructs are introduced in the RTL of a function. Thus, we decided to use RTL from the first generation.

4.3 A Trivial Memory Access Analysis

The last step was to implement the actual analysis. With the infrastructure described above, it was not difficult. The idea was to write a visitor that loops over all the instructions and that traverses their RTL subexpressions recursively to analyze them. The visitor flags a memory access only when it encounters an expression with the RTL code `MEM`.

Most of the code of the visitor implements the traversal of RTL expressions. It was written incrementally; when the visitor encountered an RTL code it did not know how to handle, it aborted with a message that indicated the next RTL code that had to be supported to continue the analysis of the function.

Our proof of concept was considered a success when the analysis correctly flagged all the memory accesses in two recursive versions of the factorial function. The visitor needed about fifty lines of code to do that.

Chapter 5

The RTL-Check Static Analysis Framework

This chapter describes the RTL-Check static analysis framework[1]. This framework makes it possible to build complex analyses from small reusable components. It is born from our first unsuccessful attempt to implement an analysis for checking whether some code has memory access errors. The problem was that the size of the code of the analysis grew rapidly while its precision improved slowly. The analysis was monolithic and more and more difficult to understand.

The main idea in the design of the RTL-Check framework is that a complex analysis should not be built in one piece. It should be composed of modules that are as simple as possible, and above all, the relations between them should be explicit and well defined. This is necessary so that an analysis can be extended easily. Another important aspect of this framework is that it integrates some patterns that are often repeated in different analyses. This makes it possible to implement an analysis by specifying only the part that is specific to it.

In spite of its name, the RTL-Check framework is independent of RTL. That being said, it provides some supplementary services related to RTL. It is also independent of the memory access analysis, which is described in Chapter 6. It is designed to support data flow analyses, and its suitability for other kinds of analysis has not been explored.

Section 5.1 describes the structure of the RTL-Check framework. It explains how the different kinds of modules it supports work together to perform an analysis. Section 5.2

[1] The RTL-Check home page is at `http://rtlcheck.sourceforge.net/` and its current version is 0.1.7 as of this writing.

discusses some theoretical results of static analysis and their relation with the RTL-Check framework. In particular, it shows how an instance of a monotone framework can be cast in our framework and how some aspects of abstract interpretation can be implemented.

5.1 Description of the Framework

As we already mentioned, modularity is a central concept in the RTL-Check framework. To render the creation of analyses possible from simple and reusable components, the framework is structured around the following four important kinds of modules.

- *Solvers*: They keep information about some properties of the program being analyzed. They can be queried and updated. An analysis can use many solvers to achieve its goal. Taken together, the solvers form what we call the *abstract state* (for a given analysis).
- *Interpreters*: They update the solvers according to the instructions of the program being analyzed.
- *Policies*: They define some logic that is required by other parts of the analysis. They can have a state but only if it is related to the whole analysis, not to a particular abstract state.
- *Analyses*: They implement the glue between all the parts of the analysis. They indicate which solvers, policies and interpreters will be used, and they include the main algorithm for the analysis.

In the framework, these modules are classes. Each of them represents an aspect that is either specific to an analysis or shared among many analyses. To create an analysis, one can create new modules or reuse existing ones from other analyses.

Figure 5.1 shows the class diagram of a fictive analysis implemented using the RTL-Check framework. This analysis (`UserAnalysis`) uses two solvers, two interpreters and two policies in addition to another analysis, which itself can use other modules not shown in the figure.

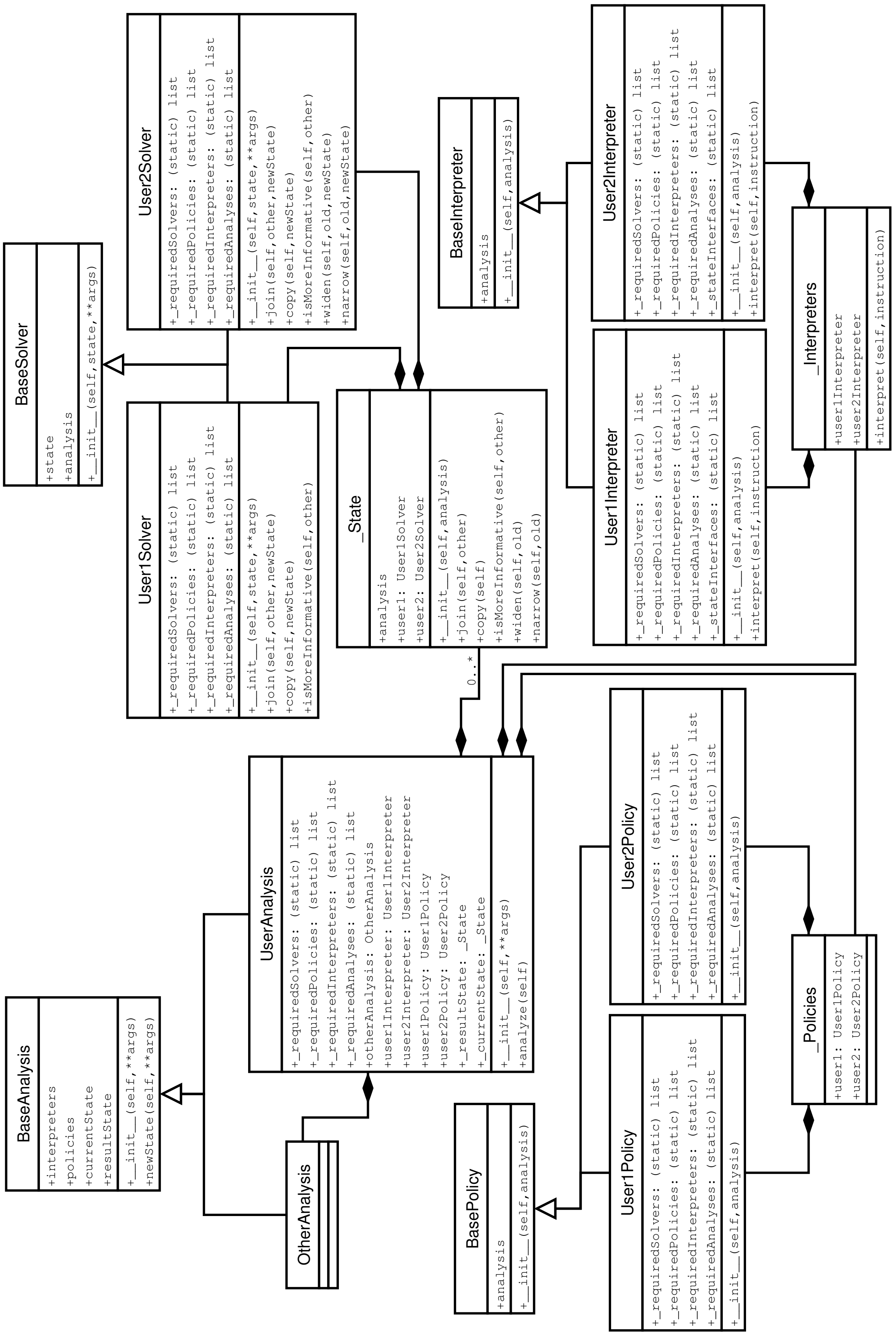
BaseSolver
+state
+analysis
+__init__(self,state,**args)
User2Solver
+_requiredSolvers: (static) list
+_requiredPolicies: (static) list
+_requiredInterpreters: (static) list
+_requiredAnalyses: (static) list
+__init__(self,state,**args)
+join(self,other,newState)
+copy(self,newState)
+isMoreInformative(self,other)
+widen(self,old,newState)
+narrow(self,old,newState)
User1Solver
+_requiredSolvers: (static) list
+_requiredPolicies: (static) list
+_requiredInterpreters: (static) list
+_requiredAnalyses: (static) list
+__init__(self,state,**args)
+join(self,other,newState)
+copy(self,newState)
+isMoreInformative(self,other)
_State
+analysis
+user1: User1Solver
+user2: User2Solver
+__init__(self,analysis)
+join(self,other)
+copy(self)
+isMoreInformative(self,other)
+widen(self,old)
+narrow(self,old)
0..*
BaseInterpreter
+analysis
+__init__(self,analysis)
User2Interpreter
+_requiredSolvers: (static) list
+_requiredPolicies: (static) list
+_requiredInterpreters: (static) list
+_requiredAnalyses: (static) list
+_stateInterfaces: (static) list
+__init__(self,analysis)
+interpret(self,instruction)
User1Interpreter
+_requiredSolvers: (static) list
+_requiredPolicies: (static) list
+_requiredInterpreters: (static) list
+_requiredAnalyses: (static) list
+_stateInterfaces: (static) list
+__init__(self,analysis)
+interpret(self,instruction)
_Interpreters
+user1Interpreter
+user2Interpreter
+interpret(self,instruction)
BaseAnalysis
+interpreters
+policies
+currentState
+resultState
+__init__(self,**args)
+newState(self,**args)
UserAnalysis
+_requiredSolvers: (static) list
+_requiredPolicies: (static) list
+_requiredInterpreters: (static) list
+_requiredAnalyses: (static) list
+otherAnalysis: OtherAnalysis
+user1Interpreter: User1Interpreter
+user2Interpreter: User2Interpreter
+user1Policy: User1Policy
+user2Policy: User2Policy
+_resultState: _State
+_currentState: _State
+__init__(self,**args)
+analyze(self)
OtherAnalysis
BasePolicy
+analysis
+__init__(self,analysis)
User2Policy
+_requiredSolvers: (static) list
+_requiredPolicies: (static) list
+_requiredInterpreters: (static) list
+_requiredAnalyses: (static) list
+__init__(self,analysis)
User1Policy
+_requiredSolvers: (static) list
+_requiredPolicies: (static) list
+_requiredInterpreters: (static) list
+_requiredAnalyses: (static) list
+__init__(self,analysis)
_Policies
+user1: User1Policy
+user2: User2Policy

Figure 5.1: UML diagram of an analysis implemented using the RTL-Check framework.

We use a notation derived from UML. A class is represented by a rectangle divided in three parts. The name of the class is in the top section, attributes and *get-properties*[2] are in the middle and methods are at the bottom. Static attributes are indicated explicitly. Arrows represent inheritance and black diamonds represent composition.

The diagram shows that the framework provides a base class for analyses, interpreters, solvers and policies. The framework also provides a class that groups solvers (to represent the abstract state), interpreters and policies. These classes, respectively `_State`, `_Interpreters` and `_Policies` in the diagram, are specifically adapted for the dependencies, direct or indirect, of each analysis.

Other aspects of the diagram specific to a given kind of modules will be explained in the following sections. Sections 5.1.1 to 5.1.4 describe solvers and abstract states, policies, interpreters and analyses in more details. Section 5.1.5 explains how it is possible to declare dependencies between modules and what it means.

5.1.1 Solvers and Abstract States

A solver is a class that represents some properties of a program at a given point. A property could be almost anything. For example, it could be the least possible value of a variable, whether a variable is odd or even, information about a pointer, information about the variables from which a value is derived, or the conjunction or disjunction of many properties.

A solver fulfills two main functions. Firstly, it collects information about the properties it is interested in. It is usually an interpreter (see Section 5.1.3) that gives to the solver the information it needs, but the information can also come from somewhere else. Secondly, it provides an interface that allows other modules to query it. The interface used to collect information is often the same for many solvers, but the query interface is always different because it is directly related to the property implemented by the solver.

A class that defines a solver should inherit from `BaseSolver` and its name should end with “Solver”. `BaseSolver` defines the following two get-properties for the solvers.

- `analysis`: The analysis object that uses the solver.

[2]We use the term get-properties (of a class) to avoid confusion with the program properties being analyzed.

- `state`: The abstract state of which the solver is part (see below).

In addition to their interfaces for collecting information and query, every solver must implement a common interface to be usable in the framework. This interface is composed of the following constructor and methods, which are called by the framework; they should not be called directly by the user.

- `__init__(self, state, **args)`: The constructor of the class that defines the solver will receive one parameter (in addition to `self`), the abstract state of which the solver is part. It may also receive other optional parameters in `args`, which is a dictionary. The solver can use these parameters to allow the creation of an instance that represents something other than the default property. If it does not know about some (or all) of the parameters that are passed, it must ignore them. It must call the constructor of its base class, `BaseSolver`, with the same parameters.
- `copy(self, newState)`: This method must return a new instance of the solver that is a copy of `self`, but which is part of `newState` instead of `self.state`.
- `isMoreInformative(self, other)`: This method must return whether the information contained in `self` is more informative than or as informative as that of `other`. Thus, it should be read as "`self` is at least as informative as `other`". When this method is called, the solver `other` is of the same type as `self`.
- `join(self, other, newState)`: This method must return a new instance of the solver that represents the merged information of `self` and `other`. When `join()` is called, the solver `other` is of the same type as `self`. It is called to merge the information of two program paths. When combining information from two or more paths, there can be a loss of precision, but there cannot be a gain of precision. Thus, assuming this method returns `returnedValue`,

  ```
  isMoreInformative(self, returnedValue)
  ```

  and

  ```
  isMoreInformative(other, returnedValue)
  ```

  should both be `True`.

There are two other methods, `widen()` and `narrow()`, that can be implemented optionally by a solver. These methods are discussed in Section 5.2.2.

Since the property that we are interested in when doing static analysis is often rather complex, using only one solver is usually not enough to build an analysis that gives the desired result. Thus, many solvers are combined to form an abstract state. The property represented by an abstract state is the conjunction of all the properties represented by the solvers it contains. For example, consider an abstract state composed of two solvers, one keeping information about the lower bound of variables and the other keeping information about pointers. The first one could represent the property "X is greater than 0 and Y is greater than 10" and the second one could represent the property "Z is either a pointer to X or a NULL pointer". Then, the abstract state would represent the property "X is greater than 0, Y is greater than 10 and Z is either a pointer to X or a NULL pointer".

It is important to note that a solver can implement a notion of disjunction (e.g. "either a pointer to X or NULL") but an abstract state is always a conjunction of solvers. This makes it easier to have solvers that are independent of each other.

Each analysis has a different class that describes its abstract state. This class is provided by the framework automatically and new instances can be created by calling the method `newState()` of the analysis class, see Section 5.1.4. The abstract state class defines the following methods which call the corresponding method of each solver it contains. These methods can be called by the user to implement an analysis.

- `__copy__(self)`: This method returns a new abstract state that is a copy of `self`. It can be called as `copy(abstractState)`.
- `isMoreInformative(self, other)`: This method returns whether the information contained in `self` is "at least as informative as that of `other`".
- `join(self, other)`: This method returns a new instance of the solver that represents the merged information of `self` and `other`. It can be called to merge the information of two program paths.
- `widen(self, old)`: This method is discussed in Section 5.2.2.
- `narrow(self, old)`: This method is discussed in Section 5.2.2.

Moreover, the abstract state class defines one get-property to obtain each of the solvers it contains. The name of the property is that of the solver with a lower case first character and no “Solver” at the end. For example, we access the `PointerSolver` of `abstracState` like this: `abstractState.pointer`.

At any moment during the analysis, there are two abstract states accessible from the analysis object. The first one is called `currentState` and it represents the state of the program before the instruction being analyzed. The second one, `resultState`, is the state processed by the analysis. This means that `resultState` is the same as `currentState` at the beginning of the analysis of an instruction, but it is gradually transformed by the analysis into the abstract state that represents the program after the instruction. Section 5.1.4 contains more details about this.

5.1.2 Policies

The main goal of the policies is to support static analysis by implementing some functions that can be accessed from any part of the analysis (solvers, interpreters and other policies). A policy can have a state that is global to one instance of an analysis, but it cannot have a state that is specific to a given abstract state. This kind of state must be kept in a solver. See Chapter 6 for some examples of policies.

In the framework, a class that defines a policy should inherit from `BasePolicy` and its name should end with “Policy”. `BasePolicy` has the following get-property.

- `analysis`: The analysis object that uses the policy.

Because the services provided are very different from one policy to another, the common interface of all the policies has only a constructor, which is always called by the framework.

- `__init__(self, analysis)`: The constructor of the class that defines the policy will receive one parameter (in addition to `self`), the analysis of which the policy is part. It must call the constructor of its base class, `BasePolicy`, with the same parameters.

The framework automatically provides a class that groups all the policies required for a given analysis. In Figure 5.1, `_Policies` plays this role for `UserAnalysis`. This class has one get-property to obtain each policy it contains. The name of the property is that of the policy with a lower case first character and no “Policy” at the end.

5.1.3 Interpreters

The job of an interpreter is to understand the instructions of the program and to transform `resultState` accordingly. To do this, an interpreter can use policies and solvers from both `currentState` and `resultState`. It can also use the result of other analyses, i.e., analyses in which it does not run.

The framework does not require any interpreter in order to have a complete analysis. For some very simple analyses, it might be easier to perform the job of the interpreter directly in the analysis class. On the other hand, a complex analysis might require more than one interpreter.

A class that defines an interpreter should inherit from `BaseInterpreter`, which has the following get-property.

- `analysis`: The analysis object that uses the interpreter.

The interface that the interpreter implements can be anything, but it usually needs only one method in addition to the constructor.

- `__init__(self, analysis)`: The constructor of the class that defines the interpreter will receive one parameter in addition to `self`, the analysis object of which the interpreter is part. It must call the constructor of `BaseInterpreter`, its base class, with the same parameters. Often, an interpreter does not even have to provide a constructor because it does not keep any state. In this case, the one of the base class is used automatically.
- `interpret(self, instr)`: It should modify `self.analysis.resultState` according to `instr`, the instruction that must be interpreted.

An interpreter also has a static attribute `_stateInterfaces` which is a list of strings that indicates to the framework the name of the methods the interpreter will use to modify `analysis.resultState`. The framework will provide an implementation of each of these methods in the abstract state class of the analysis that will call, in turn, the method of the same name for each solver that has such a method. This makes the interpreter independent of the solvers; new solvers can be added to an analysis without modification to the interpreter and the interpreter does not force the use of any specific solver.

An interpreter can also omit to declare any state interface and use directly the interface of a specific solver. However, this way it would be more difficult to extend analyses that use such an interpreter.

The framework automatically defines a class that represents the set of interpreters required for a given analysis. For example, in Figure 5.1, it is `_Interpreters` that represents the set of interpreters for `UserAnalysis`. Such a class provides the following method.

- `interpret(self, instruction)`: It calls the method `interpret()` of each interpreter in turn.

This class also provides one get-property to obtain each of the interpreters. The name of the property is that of the interpreter with a lower case first character.

5.1.4 Analyses

An analysis class is the glue between all the parts required to perform static analysis (solvers, policies, interpreters and other analyses). It implements the main loop of the algorithm and it calls the interpreter(s) as needed. It can also provide some services to other parts of the analysis while it runs (though it is probably better to have them in policies) and other services to report the results of the analysis when it is done.

A class that defines an analysis must inherit from `BaseAnalysis`. The former must provide the following constructor, method and attributes.

- `__init__(self, **args)`: The constructor must receive its parameters in a dictionary of optional parameters (`**args`) and it must ignore those it does not know about. It must also pass this dictionary to the constructor of its base class.
- `analyze()`: This method runs the analysis and should not take any parameter.
- `_currentState`: This attribute represents the abstract state before the interpretation of the current instruction. It must be set by the analysis before `interpret()` is called.
- `_resultState`: This attribute represents the abstract state that must be modified according to the instruction being analyzed. It must be set by the analysis before `interpret()` is called.

The use of a dictionary of parameters for the constructor is motivated by the fact that the framework allows one analysis to use the results of one or more other analyses. It is the framework that calls the constructor of each analysis (except of course for the top-level analysis) but it does not know what parameters are required for each of them. The dictionary of parameters solves this problem because it forces a common interface for all analyses. Thus, when instantiating an analysis, it is important to also pass the parameters for all dependent analyses, not only the parameters of the top-level one. In fact, when a top-level analysis depends on another, all the parameters of the latter become parameters of the top-level one and they should be documented as such.

Often, the constructor needs only one parameter named `function`: the function to analyze. This parameter is usually shared between all analyses. However, the framework makes it possible to define analyses that have more parameters.

The class `BaseAnalysis` also provides the following get-properties and methods.

- `newState(self, **args)`: This method creates an instance of the abstract state with the dictionary of optional parameters (`**args`) passed to the constructor of each solver.
- `interpreters`: This get-property returns the object that represents all the interpreters of the analysis.
- `policies`: This get-property returns the object that represents all the policies of the analysis.
- `currentState`: This get-property has a meaning only while the analysis calls an interpreter. It returns the abstract state before the interpretation of the current instruction.
- `resultState`: This get-property has a meaning only while the analysis calls an interpreter. It returns the abstract state that must be modified according to the instruction being analyzed.

Moreover, the framework adds one get-property for each policy and interpreter that the analysis uses. In both cases, the name of the get-property is that of the class that defines the policy or the interpreter with a lower case first character.

5.1.5 Dependencies between All Modules

The different kinds of modules (solvers, policies, interpreters and analyses) can use each other almost arbitrarily. However, the framework must know about the relations between all of them to implement some of its aspects. For example, it must know all the policies needed to run an analysis, in what order the analyses must be run if there is more than one, and in what order the solvers of an abstract state should do their job. The general rule is that if module X might use a service from module Y, then module Y should do its job before module X. Thus, every module has to declare its dependencies. Also, the framework cannot work if there is a cyclic dependency between a set of modules because it would not know which one to run first. For example, if Analysis1 uses Policy2, which in turn uses Analysis3, then Analysis3 can use neither Analysis1 nor Policy2.

The interface used to declare dependencies is the same for all kinds of modules. A class that defines a module can use the following four static attributes to declare its dependencies.

- `_requiredSolvers`: The list of solvers used by this module.
- `_requiredPolicies`: The list of policies used by this module.
- `_requiredInterpreters`: The list of interpreters used by this module.
- `_requiredAnalyses`: The list of analyses used by this module.

In all cases, the lists must contain only the class objects that define the required modules, not instances of these classes. The four attributes are optional, i.e., if a module does not use any solver, it does not have to declare `_requiredSolvers` and the same is true for policies, interpreters and analyses.

The framework traverses the dependencies recursively from the top-level analysis. This means that a module does not have to redeclare the dependencies of the modules it uses. It should only list the modules it uses directly. For example, if "Analysis1" uses only "Interpreter2", which uses "Policy3", then "Analysis1" should not declare a dependency on "Policy3". The framework also looks at the base classes of a module when it searches for its dependencies. Thus a module does not have to redeclare them, even if it calls a method of its base class that uses them. Once again, it should declare only its direct dependencies.

Every kind of module can use a solver, a policy or an analysis, so there is no restriction on dependencies between these kinds of modules, except for cyclic dependencies. Concerning interpreters, they should be used directly only by analyses but any kind of module can still declare a dependency on such module to make sure the interpreter will run.

Once the framework has collected all the dependencies, it can determine the order in which the solvers, the interpreters and the analyses will run. It is always possible to build these orders because no circular dependency is allowed between the modules.

The order of analyses is used when the user creates an instance of an analysis. At this time, all the dependent analyses are created and run, starting with the ones that have no dependencies. The order of solvers is used by the abstract state class. It calls the different solvers in the right order when an operation is executed on the abstract state. The order of interpreters is used only if the analysis calls `self.interpreters.interpret( insn)`. If it does, the `interpret()` method of each interpreter is run, starting with the ones that have no dependency on other interpreters.

5.2 RTL-Check compared to Theoretical Analysis Frameworks

This section explains the relation between RTL-Check, monotone frameworks and abstract interpretation. It also discusses general considerations about the design of an analysis to be implemented using the RTL-Check framework.

Principles of Program Analysis [NNH99] is an excellent reference for more information about many of the concepts presented in this section.

5.2.1 Monotone Frameworks

Classical data flow analyses are often categorized by the combination of the direction of the analysis (forward or backward) and the size of the wanted result (smallest or largest). Thus, there are four different kinds of analysis but they share a lot of similarities. Monotone frameworks abstract the differences between all these analyses and provide a way to look at them in a generic way.

A *monotone framework* is defined by:

- L, the property space of the framework, a partially ordered set $(L, \sqsubseteq)$;
- $\mathcal{F}$, a set of monotone functions over L.

The property space L must have a least element that we will note $\bot_L$ and it must satisfy the *ascending chain condition*, which means that each ascending chain $(l_n)_{n \in \mathbb{N}}$ eventually stabilizes, i.e., $\exists n : l_0 \sqsubseteq l_1 \sqsubseteq \cdots \sqsubseteq l_n = l_{n+1} = \cdots$. The property space must also have a combination operator $\bigsqcup : \wp(L) \to L$ which is the least upper bound operator. This implies that L is a complete lattice.

A *monotone function* f is one that respects $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$. The set of monotone functions $\mathcal{F}$ determines the functions that the framework will have to deal with. It must contain all the functions that the analysis will use and the identity function, and it must be closed under function composition. This set can be simply the set of all monotone functions over L or it can be more restrictive.

We assume that every instruction of the program has a distinct label. An *instance* of a monotone framework (a given analysis) is defined by:

- L, the property space of the monotone framework;
- $\mathcal{F}$, the set of function of the monotone framework;
- F, a finite flow of the program to analyze;
- E, a set of extremal labels;
- $\imath \in L$, an extremal value;
- $f_.$, a mapping from labels to transfer functions of $\mathcal{F}$.

The *flow* F is the set of edges (pairs of labels) of the control flow graph of the program. For backward analyses, edges are reversed. The set of *extremal labels* E contains either the label of the first instruction (for forward analyses) or the labels of instructions where the program ends (for backward analysis). The extremal value $\imath$ is the value of the property at extremal labels. This is often $\bot_L$ or $\bigsqcup L$, but it could be something else.

Table 5.1: Algorithm to solve an instance of a monotone framework.

```
INPUT:   $(L, \mathcal{F}, F, E, \imath, f_\cdot)$: an instance of a monotone framework
OUTPUT:  Before: mapping of the value of the property before each instruction.
         After: mapping of the value of the property after each instruction.
TEMP:    W: the list of edges to process
METHOD:  W := new List
         for all edges $(l, l')$ in $F$ do
             add(W, $(l, l')$)
             Before[$l$] := $\bot_L$
             Before[$l'$] := $\bot_L$
         for all labels $l$ in $E$ do
             Before[$l$] := $\imath$
         while not empty(W) do
             $(l, l')$ := pop(W)
             if $f_l$(Before[$l$]) $\not\sqsubseteq$ Before[$l'$] then
                 Before[$l'$] := $\bigsqcup\{$Before[$l'$], $f_l$(Before[$l'$])$\}$
                 for all $l''$ such that $(l', l'') \in F$ do
                     add(W, $(l', l'')$)
         for all labels $l$ such that $l \in E$ or $\exists l' : (l, l') \in F$ or $\exists l' : (l', l) \in F$ do
             After[$l$] := $f_l$(Before[$l$])
```

The mapping $f_\cdot$ indicates the transfer function that must be used for each label. A *transfer function* describes how an instruction transforms the property analyzed. Table 5.1 shows the algorithm that can be used to solve an instance of the monotone framework.

Using the RTL-Check framework, it is easy to implement a monotone framework in three parts:

- A solver class defines the property space;
- An interpreter class implements the mapping from labels (instructions) to monotone functions and the monotone functions themselves;
- An analysis class defines the links between the solver, the interpreter and the instance of the framework (flow, extremal labels and extremal value) and it provides an interface to make available the results of the analysis.

Defining a solver for a property space is straightforward. By default, the constructor should create a solver that corresponds to $\bot_L$, the least element of L. It can also allow optional parameters that make it possible to create a property different from $\bot_L$. For example, it is often useful to have a solver that corresponds to $\bigsqcup L$. The method `isMoreInformative()` is the operator $\sqsubseteq$ and the method `join` is the operator $\bigsqcup$. There must be a method `copy()` that returns a copy of the solver. The solver must also implement the interface used by the interpreter to transform the property and an interface to query its state. Table 5.2 presents a simple working solver for the liveness of registers.

The interpreter does not have to follow the formal definition of a monotone framework as closely as the solver does. There is no need to have a real mapping from labels to transfer functions. In fact the monotone functions do not have to be represented directly. Instead, the interpreter can "interpret" the program, one instruction at a time, and modify `resultState` accordingly. This is the purpose of the method `interpret()`, which receives the instruction to interpret as parameter. Instructions are usually represented by instances of the class `RtxGenInsn` and there is no real need for labels. The interpreter can also use `currentState`, which corresponds to the value that would be passed to the monotone function in the monotone framework. At the moment `interpret()` is called, `currentState` and `resultState` are equal.

An interpreter can use two ways to modify `resultState`. The first one is using directly a solver and its interface. In this case, the interpreter should declare that it depends on this solver with `_requiredSolvers`. The second one requires the interpreter to declare in `_stateInterfaces` the methods it wishes to use to communicate with solvers. It can use these methods on `resultState` directly, which will then dispatch the call to all the solvers that implement this interface. The first way prevents the interpreter from being used with a different solver. The second one is preferred because it is more extensible. Table 5.3 presents an (incomplete) interpreter for the liveness of registers.

The analysis class completes the implementation of an instance of a monotone framework. It is greatly simplified by the fact that the RTL-Check framework provides a class called `MonotoneAnalysis` that implements the algorithm of Table 5.1 and it provides an interface to obtain the abstract state before and after the interpretation of a given instruction. The analysis class only has to indicate the following items in addition to its interface for querying the results of the analysis:

- The solver that defines the property space (`_requiredSolvers`);
- The interpreter that transforms the property space according to the instructions

Table 5.2: Solver for the liveness of registers.

```
class LivenessSolver(BaseSolver):
    # Common interface for all solvers
    def __init__(self, state, **args):
        BaseSolver.__init__(self, state, **args)
        # The least element is the empty set
        # (it is consistent with join and isMoreInformative)
        self._live = set()

    def copy(self, newState):
        new = LivenessSolver(newState)
        new._live = set(self._live)
        return new

    def join(self, other, newState):
        new = LivenessSolver(self._state)
        new._live = self._live.union(other._live)
        return new

    def isMoreInformative(self, other):
        # Less live variables is more precise
        return self._live.issubset(other._live)

    # Interface of this solver for modifications
    def addLive(self, regs):
        self._live.update(regs)

    def removeLive(self, regs):
        self._live.difference_update(regs)

    # Interface of this solver for query
    # Make it possible to iterate over live registers
    def __iter__(self):
        return iter(self._live)
```

Table 5.3: Interpreter for the liveness of registers.

This table shows two implementations of an interpreter. The first one directly depends on a solver and the second one (preferred) uses an abstract state interface.

```
class LivenessInterpreter1(BaseInterpreter):
    _requiredSolvers = [LivenessSolver]

    def interpret(self, insn):
        self.use = set()
        self.def = set()
        # Code to "interpret" insn not shown
        # (it populates self.use and self.def)
        . . .
        self._analysis.resultState.liveness.removeLive(self.def)
        self._analysis.resultState.liveness.addLive(self.use)

class LivenessInterpreter2(BaseInterpreter):
    _stateInterfaces = ['addLive', 'removeLive']

    def interpret(self, insn):
        self.use = set()
        self.def = set()
        # Code to "interpret" insn not shown
        # (it populates self.use and self.def)
        . . .
        self._analysis.resultState.removeLive(self.def)
        self._analysis.resultState.addLive(self.use)
```

Table 5.4: Analysis class for the liveness of registers.

```
class LivenessAnalysis(MonotoneAnalysis):
    _requiredInterpreters = [LivenessInterpreter]
    _requiredSolvers = [LivenessSolver]

    def __init__(self, **args):
        MonotoneAnalysis.__init__(self, **args)
        func = args['function']
        self.flow = ReverseFlow(func)
        # For liveness analysis, the extremal value
        # is the least element
        self.extremalValue = self.newState()

    def liveBefore(self, insn):
        # The flow is reversed
        # before the instruction is after interpretation
        return iter(self.after(insn).liveness)
```

(`_requiredInterpreters`);

- The flow and extremal instructions of the function to analyze, which are encapsulated in a single class (`self.flow`);

- The extremal value (`self.extremalValue`).

The framework provides two basic flow classes that can be used for monotone analyses, `StraightFlow` for forward analyses and `ReverseFlow` for backward analyses. Both of them need the function to analyze as parameter to their constructor. The flow object must be stored by the analysis in `self.flow` and the extremal value in `self.extremalValue`. There is no need to provide an `analyze()` method because `MonotoneAnalysis` has it already. Table 5.4 shows the class that completes the implementation of liveness analysis for registers.

5.2.2 Abstract Interpretation

Abstract interpretation is a very general framework in which analyses are calculated. It does not mandate the use of a given algorithm. Instead, new analyses are created from the specification of existing analyses. Since it is possible to specify an analysis that is very precise but uncomputable, the goal of abstract interpretation is often to create an analysis that is computable, i.e., one that will not run forever. Other frequent uses of abstract interpretation include creating analyses that terminate in fewer iterations or that use less memory.

One important aspect of abstract interpretation is that the method for constructing a new analysis ensures that its correctness follows from the correctness of the original analysis. Galois connections, for example, are very useful for this purpose. However, the RTL-Check framework does not know about the correctness of an analysis and it offers no support for calculating analyses. These tasks must be done manually. Nevertheless, the RTL-Check framework offers some support for implementing analyses created using abstract interpretation.

Widening and Narrowing

A concept often used in abstract interpretation is that of widening operators. A widening operator can turn any ascending chain, which may increase infinitely, into an ascending chain that eventually stabilizes. First we define:

$$l_n^{\triangledown} = \begin{cases} l_n & \text{if } n = 0 \\ l_{n-1}^{\triangledown} \triangledown l_n & \text{if } n > 0. \end{cases}$$

The operator $\triangledown : L \times L \to L$ is a *widening operator* if it respects the following conditions:

- $a \sqsubseteq a \triangledown b \sqsupseteq b$, i.e., $\triangledown$ is an upper bound operator;
- Given any ascending chain $(l_n)_{n\in\mathbb{N}}$, the ascending chain $(l_n^{\triangledown})_{n\in\mathbb{N}}$ eventually stabilizes, i.e., $\exists m : l_0^{\triangledown} \sqsubseteq l_1^{\triangledown} \sqsubseteq \cdots \sqsubseteq l_m^{\triangledown} = l_{m+1}^{\triangledown} = \cdots$.

Note that $(l_n^{\triangledown})_{n\in\mathbb{N}}$ is an ascending chain because $\triangledown$ is an upper bound operator. Also note that $\triangledown$ does not have to be commutative and monotone, and usually it should not

Table 5.5: Example of a widening operator.

Consider an analysis that computes the greatest possible value of a variable. The property space could be the complete lattice $(\mathbb{Z} \cup \{-\infty, +\infty\}, \sqsubseteq)$, where

$$a \sqsubseteq b = \begin{cases} a \leq b & \text{if } a, b \in \mathbb{Z} \\ \textbf{true} & \text{if } a = -\infty \text{ or } b = +\infty \\ \textbf{false} & \text{otherwise.} \end{cases}$$

We can define the following simple widening operator on this property space:

$$a \bigtriangledown b = \begin{cases} a & \text{if } b \sqsubseteq a \\ b & \text{if } a = -\infty \\ +\infty & \text{otherwise.} \end{cases}$$

This operator is not commutative because $1 \bigtriangledown 2 = +\infty$ and $2 \bigtriangledown 1 = 2$; the left operand can be seen as the "old" value, i.e., the result of the last iteration of the analysis. Also, it is not monotone in its left operand because it would mean that

$$a \sqsubseteq b \Rightarrow a \bigtriangledown c \sqsubseteq b \bigtriangledown c.$$

For $a = 1$, $b = 3$ and $c = 2$, we have $1 \bigtriangledown 2 = +\infty$ and $3 \bigtriangledown 2 = 3$; we observe that $+\infty \not\sqsubseteq 3$.

be; see Table 5.5 for an example. Since $(l_n^{\bigtriangledown})_{n \in \mathbb{N}}$ is guaranteed to eventually stabilize, the widening operator can be used to perform an analysis on a property space that does not satisfy the ascending chain condition.

A related concept is that of narrowing, which makes it possible to recover some of the precision lost by the use of a widening operator and still make sure that the analysis will not iterate forever. Narrowing is not the dual of widening. See [NNH99] for the definition of narrowing.

In the RTL-Check framework, each solver can define its widening and narrowing operators by implementing the following methods, which are optional.

- `widen(self, old, newState)`: This method must return a new instance of the solver that is the result of the widening between `self` and `old`.
- `narrow(self, old, newState)`: This method must return a new instance of the solver that is the result of the narrowing between `self` and `old`.

The RTL-Check framework provides two methods for widening and narrowing in the abstract state class.

- `widen(self, old)`: This method returns a new abstract state in which each solver that implements `widen()` is created by calling this method, and each solver that does not implement it is a copy of the solver `self`, i.e., there is no widening.
- `narrow(self, old)`: This method returns a new abstract state in which each solver that implements `narrow()` is created by calling this method, and each solver that does not implement it is a copy of the solver `self`, i.e., there is no narrowing.

Combining Analyses

Another approach used to create new analyses with abstract interpretation is the combination of two or more existing analyses. Sometimes the precision of the result can be improved when there is some overlap between the property spaces of the analyses. Other times, the goal is just to perform two or more independent analyses in parallel.

The RTL-Check framework is particularly well suited for the combination of analyses. We have seen in Section 5.1.5 how to indicate to the framework the solvers that must be part of the abstract state. These solvers can receive their data from the same interpreter or from different modules. If two analyses use the same algorithm, e.g. the one from `MonotoneAnalysis`, the only thing needed to perform a parallel analysis is to add the solvers and the interpreters of one of them to the list of dependencies of the other.

If the algorithms are not exactly the same, they must be adapted. Of course there must be some similarity in the algorithms for this to be possible, but it is often the case.

When the goal is to improve precision, it is usually not enough to run two independent analyses in parallel. In this case, it might be more appropriate to have a solver that depends on one or more other solvers. It can then combine its own information with the information from other solvers to give a more precise result.

5.2.3 General Considerations

It is possible to use the RTL-Check framework to design analyses that are neither a monotone framework nor the result of abstract interpretation. In this section, we will see the possible impacts of changing some aspects that are requirements for monotone frameworks and abstract interpretation.

Complete Lattices

A central concept of both monotone frameworks and abstract interpretation is that the property space is a complete lattice, or at least a join-semilattice with a unit element, which corresponds to the least element of a complete lattice. It is natural to define such a property space even if the RTL-Check framework does not mandate it. Being a *join-semilattice* ensures that any two elements have a least upper bound, which means that it is possible to combine information from two program paths safely. The *unit* (or least) element is useful when no information is available yet for a program point. It allows using a general algorithm that does not have to keep track explicitly of which program paths have been analyzed or not. It is also useful that all solvers create this least element by default, to allow algorithms to be independent of the solvers they work with.

The method `join()` of solvers should be associative, commutative and idempotent, otherwise this method would not implement the least upper bound operator, and thus an analysis that uses it will probably be less precise or unreliable. Associativity and commutativity also ensure that information from different paths can be combined in any order without affecting the result of the analysis.

The method `isMoreInformative()` of a solver should describe the relation “is at least as informative as”. Since the goal of an analysis is usually not only to find a correct result, but the most informative one, doing so will be much more difficult without a way to accurately compare results. Moreover, the implementation of `isMoreInformative()` should be consistent with that of `join()`, i.e., at the very least the following properties should hold for all `a` and `b` to represent the fact that we cannot gain information by merging that of two paths:

`a.isMoreInformative(a.join(b)) == True` and
`b.isMoreInformative(a.join(b)) == True`.

For the case where `join()` is the least upper bound operator, we have

`a.isMoreInformative(b)` if and only if `a.join(b)` equals `b`.

Monotone Functions

The use of monotone functions for transfer functions is also natural in program analysis. In the context of static analysis, the formula $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$ means that if we have more information about the input of an instruction, we cannot get less information about its output. Using monotone functions also ensures that the transfer functions have a least fixed point and that it is possible to reach it in a finite number of iterations under certain circumstances, see Termination and Correctness below. The least fixed point is important because it represents the best possible solution to an analysis.

Termination and Correctness

When designing an analysis, it is important to make sure that it will not run forever. In monotone frameworks, the use of complete lattices that satisfy the ascending chain condition and monotone transfer functions ensures termination. In abstract interpretation, when using widening and narrowing operators, we are not sure to reach a fixed point of the transfer functions, let alone the least one. However, we are sure that the analysis will end, even though it might not give the best possible result.

When creating an analysis in RTL-Check, if the property space is not a complete lattice that satisfies the ascending chain condition and there is no widening, or if the transfer functions are not monotone, there should still be something that ensures it will eventually terminate.

In fact the analysis should not only terminate, it should be correct with respect to the semantics of the program. Proving that might turn out to be difficult if the analysis deviates too much from established results.

Chapter 6

The Memory Access Analysis

This chapter describes the memory access analysis which is built using the RTL-Check framework and distributed with it[1]. Its goal is to detect memory access errors in programs. This analysis is rather complex, it uses two interpreters, many solvers and many policies in addition to the liveness analysis described in Section 5.2.1. This analysis is not interprocedural and it does not support function calls yet.

6.1 Description of the Analysis

Figure 6.1 shows the dependencies (dashed arrows) between the classes that make up the memory access analysis. Some of them, which have no dependency, appear many times to prevent arrows from crossing. Also, two policies, `AbstractVariablePolicy` and `SafeMemoryVariablePolicy`, have been grouped to make the diagram clearer. The former depends on the latter, but in practice, they complete each other.

The different solvers of this analysis record information about the program state. The main interpreter and `SafeMemoryVariablePolicy` use this information to detect possible memory access errors. When such an error is detected, it is logged using `ErrorLogPolicy`. The list of errors is shown to the user at the end of the analysis. The following sections describe the modules in more details and they explain the most important relations between them.

[1] The RTL-Check home page is at `http://rtlcheck.sourceforge.net/` and its current version is 0.1.7 as of this writing.

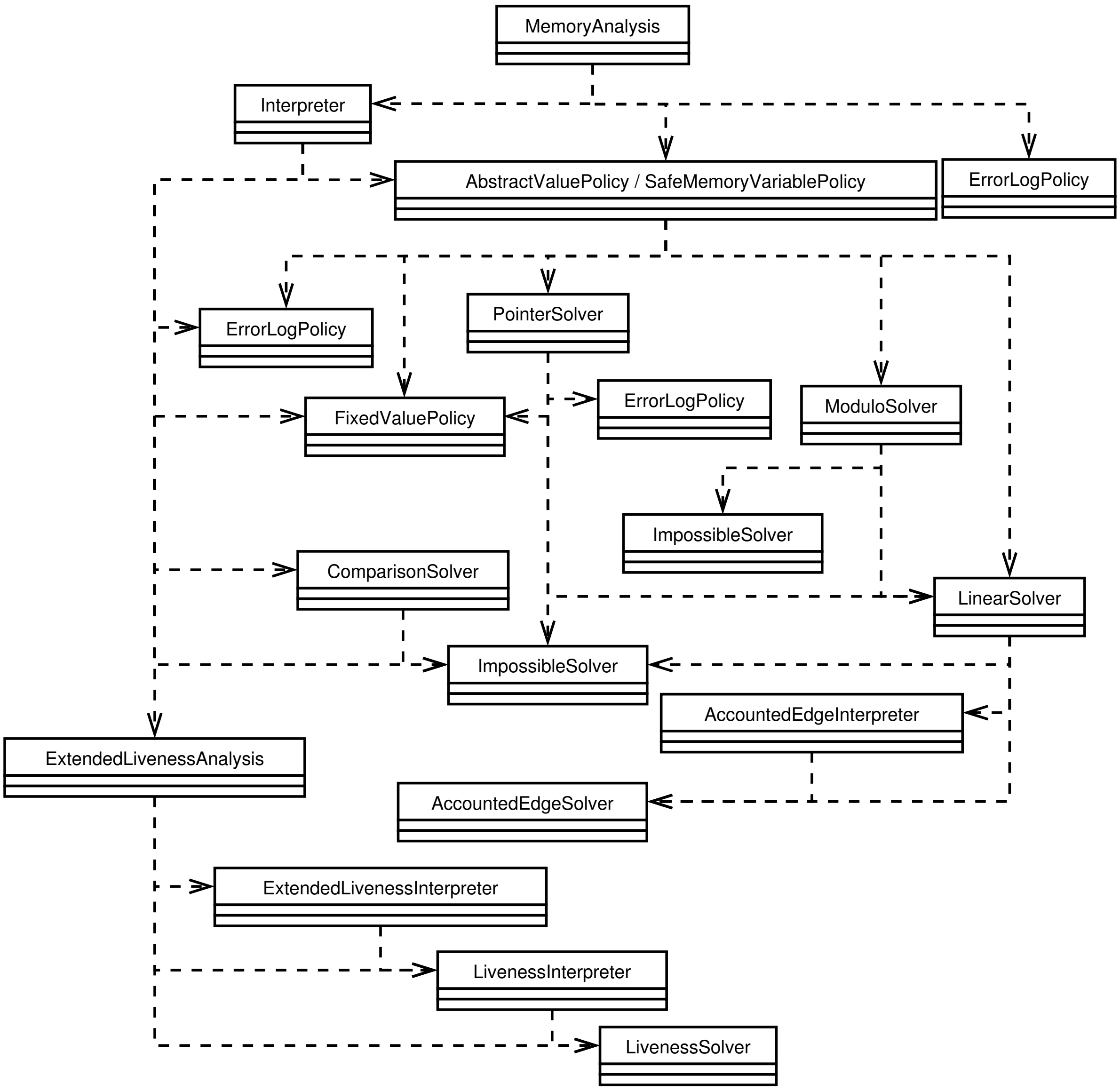

Figure 6.1: Overview of the memory access analysis.

6.1.1 The Main Interpreter

The main interpreter is implemented as an RTL visitor, a design pattern described in Section 7.3. Its `interpret()` method takes a pair of instructions, instead of only one as discussed previously. The first one is the instruction to interpret and the second one is the following instruction. Because of this, conditional jumps must be interpreted twice but the interpreter knows the outcome of the test, which can improve precision. It is the class `EdgeFlow` (instead of `StraightFlow`) that provides this kind of flow.

The main interpreter uses two methods to modify `resultState` when it interprets an instruction. The first one is `setValue(self, value, expression)`. The parameter `value` will be an *abstract variable*, i.e., an instance of a class that inherits from `AbstractVariable`, which is defined in `value.py`. When the interpreter calls `setValue()`, it means that this abstract variable, which represents a register or a memory word, gets a new value. The parameter `expression` represents this new value. It is an *abstract variable expression*, which can be an instance of one of the following classes.

- `UnknownValue`: It is used when a value is unknown. It can be the result of an operation that is not implemented precisely enough by the interpreter or a solver, but it can also be because `value` could be an alias to another value that was just modified. The interpreter also uses it in some cases when `value` is dead, i.e., `value` will not be used again later in the program before being assigned again. For some solvers, "forgetting" a value can save both memory and execution time.
- `Integer`: It is used when an exact integer value is known.
- `AbstractVariable`: It is used to represent a value that is unknown to the interpreter, but that could be referred from elsewhere in the program. An abstract variable has not necessarily been assigned a value by the interpreter before it is used in an abstract variable expression.
- `BinaryOp` (`Plus`, `Minus`, `Times`): They are used when a value can be described by two other abstract variable expressions, which are combined using an arithmetic operation.
- `AbstractAddress`: It is used to describe a value that represents a memory address. An `AbstractAddress` is composed of a *zone* (an abstract memory area) and an offset, which itself is an abstract variable expression, i.e., it may not be known precisely.

Different zones are considered distinct and non-overlapping. However, a single zone might represent more than one concrete memory area at the same time. Each zone has three *zone attributes* that describe it. For the different solvers that do not treat them in a special way, a zone attribute is just another kind of `AbstractVariable`. The zone attributes are the following.

- `NumInstances`: It represents the number of distinct concrete memory areas represented by the zone.
- `StartOffset`: It represents the offset at which the zone starts. It can be positive or negative. A memory access to the zone at an offset less than this offset is wrong.
- `EndOffset`: It represents the offset at which the zone ends. It can be positive or negative. A memory access to the zone at an offset greater than or equal to this offset is wrong.

All of these classes are defined in `value.py`.

The second method that the interpreter uses to modify `resultState` is `addConstr(self, constr)`. When it calls this method, it means that the *abstract variable constraint* `constr` is satisfied at this point in the analysis. This parameter is an instance of one of the following classes, which all inherit from `BinaryComp`. Each of them has two attributes, `left` and `right`, which are abstract variable expressions.

- `Equal`: It means that `left` is equal to `right`.
- `NotEqual`: It means that `left` is not equal to `right`.
- `LessThan`: It means that `left` is less than `right`.
- `LessThanEqual`: It means that `left` is less than or equal to `right`.
- `GreaterThan`: It means that `left` is greater than `right`.
- `GreaterThanEqual`: It means that `left` is greater than or equal to `right`.

To do its work, this interpreter relies mainly on `AbstractVariablePolicy` and `SafeMemoryVariablePolicy`.

The first policy, `AbstractVariablePolicy`, is used by the interpreter to convert RTL code and zone attributes to abstract variables and abstract variable expressions. The second policy, `SafeMemoryVariablePolicy`, is used to get the list of possible aliases of an abstract variable that represents a memory word when it is modified. These policies will be described in Sections 6.1.7 and 6.1.8 respectively.

This interpreter does not understand all RTL expressions, but it knows enough to analyze many procedures. If it encounters an unknown expression, it prints an informative message and aborts.

The main RTL instruction that is handled by this interpreter is `Set`. It is used for both assignments and jumps. For simple jumps, the interpreter does nothing because no register or memory is modified. For a conditional jump, nothing changes, but we gain information because the interpreter knows if the jump is taken or not, and thus whether the condition is true or false. Thus it uses `AbstractVariablePolicy` to convert the RTL expression of the condition into `constr`, an abstract variable constraint. It then calls `resultState.addConstr(constr)`.

For assignments, it uses `AbstractVariablePolicy` to transform both the source and the destination RTL expressions into abstract variable expressions `src` and `dst` respectively, and it calls `resultState.setValue(dst, src)`. If the destination is in memory (i.e., not a register) it also calls `resultState.setValue(a, UnknownValue())` for each alias `a` of `dst` according to `SafeMemoryVariablePolicy`.

It is not possible to tell whether the transfer functions implemented by this interpreter are monotone with only the description above, because it depends much on the definition of `setValue()` and `addConstr()` of each solver. These methods must be monotone in `self` for the whole abstract state, not only internally.

Also, since the list of aliases known by `SafeMemoryVariablePolicy` can grow during the analysis, the transfer functions do not depend only on the abstract state, but also on this policy. This means that transfer functions could give a less informative result for a second call with the same abstract state. It would mean that the functions are not monotone, but this case cannot happen in the current analysis because, if `SafeMemoryVariablePolicy` does not know about an alias, it also means that no solver has seen this alias yet, and thus no solver can have any information about it.

6.1.2 The Impossible Solver

`ImpossibleSolver` is a very simple solver defined in `impossible.py`, which only keeps track of whether or not it is possible to reach a given point of the program. It is used by some solvers that keep constraints on abstract variables to represent their "most precise" state. For these solvers, an impossible state can be interpreted as a state where all possible constraints are respected at the same time, including those that are contradictory. This is obviously not possible for any given execution of any program. Since an abstract state becomes more precise when constraints are added, it is easy to understand why the impossible state is the most precise state.

`ImpossibleSolver` does not implement the method `setValue()`, it only implements `addConstr(self, constr)`. It ignores every constraint it sees, except those that are instances of `Impossible` or one of its derived classes declared in `constraint.py`. In this case, it makes the constraint accessible by its get-property `impossible` because it could be useful for some other part of the analysis to know why some code is unreachable. The memory access analysis however does not use it currently.

`Impossible` constraints are not generated by the interpreter directly, they are generated by other solvers when they detect that two constraints cannot be satisfied at the same time.

`ImpossibleSolver` follows the rule that, by default, a solver is created in its most precise state. It means that other solvers using this one to represent their most precise state also follow this rule automatically. To create an abstract state that is not impossible, one must pass the optional parameter `bottom=False`.

Another advantage of using this solver is that a solver can optimize some of its operations when an abstract state is determined to be impossible. Yet another advantage is that when a new solver that reports constraints that cannot be satisfied is added to an analysis, all existing solvers using `ImpossibleSolver` become more precise because they share information with the new solver, even if they do not know its existence.

The definition of `copy()` for this solver is trivial. The method `join()` returns an impossible solver only when both `self` and `other` are impossible. The method `isMoreInformative()` returns `False` only if `other` is impossible and `self` is not, otherwise it returns `True`.

6.1.3 The Linear Constraints Solver

`LinearSolver` is inspired by the description of ARCHER [XCE03]. It is implemented in `linear.py`. It keeps information about the integer value of each abstract variable and linear relations between them. More precisely, for each abstract variable that is not known to be constant, this solver keeps the following information:

- An integer lower bound;
- An integer upper bound;
- A list of integers that cannot be equal to the abstract variable;
- A list of linear derivations that are lower bounds for the abstract variable;
- A list of linear derivations that are upper bounds for the abstract variable;
- A list of linear derivations that cannot be equal to the abstract variable.

Here, a *linear derivation* is an expression that has the form $(a \cdot \alpha + b)/c$, where a, b and c are integers (with $c \neq 0$) and α is a *solver symbol*. A solver symbol represents an unknown value, but it is useful for expressing relations between abstract variables.

For efficiency and precision, the information enumerated above is not kept for each abstract variable individually, but only for solver symbols. Thus, abstract variables are associated to either an integer value or a linear derivation. One advantage of this approach is that when an abstract variable is assigned a new value that can be transformed into a linear derivation, there is no need to copy the information about upper bounds, lower bounds and values that are not equal. This information can all be retrieved when needed by following the chain of linear derivations between symbols, which are indexed in both ways. But the main advantage is that if a constraint that improves the lower or upper bound of an abstract variable is added later, all other abstract variables that derive directly or indirectly from the same symbol as this abstract variable automatically get improved bounds at the same time.

To implement this behavior, this solver tries to transform abstract variable expressions received by both `setValue()` and `addConstr()` into linear derivations with the following rules, which are defined in a value visitor (see Section 7.3):

- $a \mapsto$ `Integer`(i) (if $a \in$ `AbstractVariable` and the solver knows a equals i);

- $a \mapsto l$ (if $a \in$ `AbstractVariable` and the solver knows a equals $l \in$ `LinearDerivation`);
- $a \mapsto l$ (if $a \in$ `AbstractVariable` and the solver does not know about a; l is a linear derivation of a fresh symbol);
- `Plus`$(i, j) \mapsto i + j$ (if $i, j \in$ `Integer`);
- `Plus`$(i, l) \mapsto i + l$ (if $i \in$ `Integer` and $l \in$ `LinearDerivation`);
- `Plus`$(l, i) \mapsto l + i$ (if $i \in$ `Integer` and $l \in$ `LinearDerivation`);
- `Minus`$(i, j) \mapsto i - j$ (if $i, j \in$ `Integer`);
- `Minus`$(i, l) \mapsto i - l$ (if $i \in$ `Integer` and $l \in$ `LinearDerivation`);
- `Minus`$(l, i) \mapsto l - i$ (if $i \in$ `Integer` and $l \in$ `LinearDerivation`);
- `Times`$(i, j) \mapsto i \cdot j$ (if $i, j \in$ `Integer`);
- `Times`$(i, l) \mapsto i \cdot l$ (if $i \in$ `Integer` and $l \in$ `LinearDerivation`);
- `Times`$(l, i) \mapsto l \cdot i$ (if $i \in$ `Integer` and $l \in$ `LinearDerivation`).

Note that it is trivial to define operators $+$, $-$ and $\cdot$ between integers and linear derivations. For example,

$$\begin{aligned} & \texttt{Integer}(x) \cdot \texttt{LinearDerivation}((a \cdot \alpha + b)/c) \\ = \; & \texttt{LinearDerivation}((a \cdot x \cdot \alpha + b \cdot x)/c). \end{aligned}$$

The operator / can also be defined precisely when the first operand is a linear derivation and the second one is an integer different from 0. Also note that the result of these operations on two linear derivations is not a linear derivation in general. For example, `LinearDerivation`$(\alpha_1) +$ `LinearDerivation`(α_2) is not a linear derivation.

When the transformation succeeds, its result is either an integer or a linear derivation. In the case of `setValue()`, the result is assigned directly to the abstract variable and the old information is lost. For `addConstr()`, constraints of the type `LessThan` and `GreaterThan` are first converted to `LessThanEqual` and `GreaterThanEqual` respectively by adding one to the "b" of the smallest linear derivation. For these cases, there are four possibilities, but three of them are handled about the same way.

- If both `left` and `right` are integers, we verify that the constraint is an arithmetic fact. If it is not, we call `addConstr()` on the abstract state to indicate that we are in an impossible state.

- If at least one of `left` and `right` is a linear derivation, we use the following procedure.
  - Without loss of generality, we will assume that `left` $=$ `LinearDerivation`$((a \cdot \alpha + b)/c)$.
  - We first normalize the constraint to have the form: `left`$' = \alpha$ and `right`$' =$ (`right` $* c - b)/a$.
  - The constraint must be reversed if $a{*}c$ is negative, e.g. `LessThanEqual(left, right)` becomes `GreaterThanEqual(left′, right′)`.
  - The division used to compute `right`$'$ is exact if `right` is a linear derivation, but it cannot be if it is an integer. In this case, it must be the "floor division" if the final constraint is of the form `LessThanEqual(left′, right′)` and the "ceiling division" for the form `GreaterThanEqual(left′, right′)`. If another division was used, the result would be less precise.
  - When the constraint is in its final form, it is added to the internal state of the solver for the symbol α.
  - If `right`$'$ is a linear derivation, its upper or lower integer bound is also taken into account for `left`$'$. If `right`$'$ is an integer, this bound is propagated to all symbols that can be linked to `right`$'$ through existing linear derivations. This ensures that the lower and upper bounds of each symbol are always up-to-date in the solver.

If the type of the constraint is `Equal`, it is treated as both `LessThanEqual` and `GreaterThanEqual`. If it is of type `NotEqual`, it is almost like the case of `LessThanEqual` or `GreaterThanEqual`, except that if the remainder of the division is not zero, we do not gain any information because symbols represent integer values.

If the transformation fails for `setValue()`, the solver deletes all the information it has about the abstract variable that is being assigned. If it fails for either abstract variable expression of `addConstr()`, it ignores the constraint.

The fact that abstract variables and symbols represent integers allows this solver to deduce information that is not always obvious at first sight. For example, assume it receives the following information:

```
setValue(b, Times(a, 10)),
addConstr(GreaterThan(b, -10)),
addConstr(LessThan(b, 10)).
```

From this, the solver can correctly deduce that both abstract variables `a` and `b` are equal to 0. Another example is

```
setValue(b, Times(a, 5)),
addConstr(Equal(b, 4)).
```

In this case, the solver correctly deduces that it is impossible, which means that the path being analyzed cannot be executed.

The implementation of `copy()` for this solver is straightforward, but `join()` and `isMoreInformative()` are less obvious. Nevertheless, they share a lot.

The method `join(self, other)` first uses `ImpossibleSolver` to check if `self` is impossible and it returns a copy of `other` in this case. Likewise, it returns a copy of `self` if `other` is impossible. Otherwise, for each abstract variable that is present in both solvers, the set of constraints of which it is part is computed in both solvers using the graph of linear derivations. The intersection of these sets is added to a new solver, which is returned when the method ends. This defines an upper bound operator, but not the least upper bound. Thus, there is room for improvement in this implementation.

The method `isMoreInformative(self, other)` also checks if `self` is impossible by using `ImpossibleSolver` and returns `True` in this case. If `other` is impossible, it returns `False`. Otherwise, for each abstract variable from `other`, the set of all constraints of which it is part is computed using the graph of linear derivations. `False` is returned if one of these constraints is not satisfied in `self`. `True` is returned otherwise.

This solver also offers many other methods to query its state. They allow, among other things, checking whether a constraint is known to be satisfied by the solver, determining whether an abstract variable is constant, its lower bound and its upper bound.

This solver does not satisfy the ascending chain condition. For example, we can build an infinite chain of more informative constraints: $a > 0, a > 1, a > 2, \ldots$ However this solver has a useful, but incomplete, implementation of widening, as discussed in Section 6.1.11.

6.1.4 The Modulo Constraints Solver

`ModuloSolver` is defined in `modulo.py` and it tracks the remainder of the division by a given divisor for each abstract variable. It is useful to know the remainder of the division in addition to the lower and upper bounds of an abstract variable to gain precision when some code deals with an array or a structure. For example, if we can determine that the remainder of the division by 8 of the offset of a memory access is 0, and the remainder of the division by 8 of another memory access is 4, we are sure that the two abstract variables representing these memory accesses cannot be aliases.

To do its work, this solver uses a class named `Congruence`. To create an instance, this class uses two parameters, `mod` and `rem`, the latter being 0 by default. An instance of this class represents the subset of the integers $\{x \in \mathbb{Z} | x \equiv \texttt{rem} \pmod{\texttt{mod}}\}$. We use `mod == 0` to represent the empty set and `mod == 1` to represent $\mathbb{Z}$.

Defining $\sqsubseteq$ on `Congruence` is natural; it corresponds to the subset operator $\subseteq$ on the set of integers represented by the congruence. Note that, with this partial ordering, `Congruence` can be seen as a complete lattice[2].

`Congruence` has many methods, which are the following.

- `join(self, other)`: The binary least upper bound operator. This method returns a `Congruence` that represents all integers that are represented by either `self` or `other`. It also works if `other` is an `Integer`. This operation is not precise, which means that the resulting `Congruence` might represent some integers that were represented by neither `self` nor `other`.
- `meet(self, other)`: The binary greatest lower bound operator. It returns a `Congruence` that represents only integers that are represented by both `self` and `other`. This operation is precise, which means that the resulting `Congruence` represents exactly the integers that are represented by both `self` and `other`. It also works if `other` is an `Integer` in which case it may also return an `Integer`.
- `__neg__(self)`: Abstract negation. It returns a `Congruence` that represents the integers which are the negation of an integer represented by `self`. This operation is precise.
- `__add__(self, other)`: Abstract addition. It returns a `Congruence` that represents all the integers which are the sum of an integer represented by `self` and

[2]In the rest of the text, we will use the concept of class and that of mathematical structure interchangeably when appropriate. For example, we can say that `Congruence` is a complete lattice.

an integer represented by `other`. This operation is not precise. It also works if `other` is an integer, in which case it is precise.

- `__mul__(self, other)`: Abstract multiplication. It returns a `Congruence` that represents all the integers which are the product of an integer represented by `self` and an integer represented by `other`. This operation is not precise. It also works if `other` is an integer, in which case it is precise.
- `__contains__(self, i)`: It returns whether `self` represents the integer `i`.
- `isCongruent(self, other)`: It returns whether `self` represents a subset of the integers represented by `other`. It is the operator $\sqsubseteq$ discussed above. It also works if `other` is an `Integer`.
- `isDisjoint(self, other)`: It returns whether there exists no integer represented by both `self` and `other`.
- `isNothing(self)`: It returns whether `self` represents no integer.
- `isAnything(self)`: It returns whether `self` represents all integers.
- `after(self, i)`: It returns the least integer represented by `self` that is greater that integer `i`.
- `before(self, i)`: It returns the greatest integer represented by `self` that is less that integer `i`.

`ModuloSolver` keeps a `Congruence` for each abstract variable it knows. It receives its information from `setValue()` and `addConstr()`. Much like `LinearSolver` does for linear derivations, this solver tries to transform abstract variable expressions into congruences. For this purpose, it uses the following rules, defined in a value visitor that uses `LinearSolver` to improve precision when a constant integer value is known:

- $a \mapsto$ `Integer`(i) (if $a \in$ `AbstractVariable` and the `LinearSolver` knows a equals i);
- $a \mapsto c$ (if $a \in$ `AbstractVariable` and this solver knows a is represented by $c \in$ `Congruence`);
- `Plus`$(i, j) \mapsto i + j$ (if $i, j \in$ `Integer` $\cup$ `Congruence`; addition is defined by the method `__add__` described above);
- `Minus`$(i, j) \mapsto i + (-j)$ (if $i, j \in$ `Integer` $\cup$ `Congruence`; addition and negation are defined by the methods `__add__` and `__neg__` described above);

- Times$(i, j) \mapsto i \cdot j$ (if $i, j \in$ Integer $\cup$ Congruence; multiplication is defined by the method __mul__ described above);
- $x \mapsto c$ (if x is something else; c is the Congruence that represents all integers).

Because of the last rule, this transformation cannot fail and it always results in a Congruence or an Integer. For setValue(av, x), if the transformation of the expression x results in a Congruence, the solver uses it directly as the new value of the abstract variable av. If it is an Integer, information about av is cleared from the solver. This is not a loss of information because LinearSolver still knows that av is constant.

For addConstr(), the solver ignores all constraints except those of type Equal. In this case, if the transformation of at least one of the expressions (left-hand side and right-hand side of the constraint) result in a Congruence, it calls meet() between both of them. If this last result is a Congruence that represents the empty set, addConstr() is called on the abstract state to indicate that we are in an impossible state. If it is another Congruence, and the left side is an abstract variable, it is assigned the result of meet(). Similarly, if the right side is an abstract variable, it is assigned the result of meet().

Here is an example of a situation that this solver can determine to be impossible, whereas LinearSolver cannot:

```
setValue(b, Times(a, 16)),
setValue(d, Times(c, 8)),
addConstr(Equal(d, Plus(b, 4))).
```

The implementation of copy() for this solver is straightforward. The methods join() and isMoreInformative() are also pretty simple. They first check if an abstract state is impossible, like LinearSolver. Then they iterate over all abstract variables and use the methods join() and isCongruent() of Congruence to compute their result.

This solver satisfies the ascending chain condition because given a Congruence, strictly less precise constraints can only be created by removing a factor from the modulo, and there is a finite number of factors for any given integer. Thus, this solver does not need widening.

6.1.5 The Pointer Solver

`PointerSolver` is defined in `pointer.py` and it tracks the relations between abstract variables and abstract addresses. To do this, it associates an abstract address to every abstract variable it knows. Much like `LinearSolver` and `ModuloSolver`, this solver receives its information from calls to `setValue()` and `addConstr()` and it uses simple rules, implemented in a value visitor, to transform abstract variable expressions into abstract addresses. They are:

- $v \mapsto a$ (if $v \in$ `AbstractVariable` and this solver knows v equals $a \in$ `AbstractAddress`);
- $v \mapsto i$ (if $v \in$ `AbstractVariable` and the `LinearSolver` knows v equals $i \in$ `Integer`);
- $\texttt{Plus}(\texttt{AbstractAddress}(z, o), v) \mapsto \texttt{AbstractAddress}(z, \texttt{Plus}(o, v))$;
- $\texttt{Plus}(v, \texttt{AbstractAddress}(z, o)) \mapsto \texttt{AbstractAddress}(z, \texttt{Plus}(o, v))$;
- $\texttt{Minus}(\texttt{AbstractAddress}(z, o), \texttt{AbstractAddress}(z, p)) \mapsto \texttt{Minus}(o, p))$ (if there is only one instance of z);
- $\texttt{Minus}(\texttt{AbstractAddress}(z, o), v) \mapsto \texttt{AbstractAddress}(z, \texttt{Minus}(o, v))$.

After transformation, if the result is an abstract address, it can be used by this solver. For calls to `setValue(av, x)`, the goal is to associate the abstract address to `av`. For calls of the form `addConstr(Equal(av, x))` or `addConstr(Equal(x, av))`, the goal is the same except that if there is an existing association for `av`, the new one is ignored. Other calls to `addConstr()` with other kinds of constraints are ignored.

Unlike `LinearSolver` and `ModuloSolver`, this solver does not transform all abstract variables. This means that the offset of an `AbstractAddress` can still be an abstract variable that is not a constant. One more transformation rule is applied to ensure that this solver will always remember the right abstract address, even if the abstract variable is changed by a subsequent instruction:

- $v \mapsto f$ (if $v \in$ `AbstractVariable`; $f \in$ `AbstractVariable` is the fixed value for v at the current instruction according to `FixedValuePolicy` (see Section 6.1.9)).

After this last transformation, the association between `av` and the abstract address can be saved. `FixedValuePolicy` does not attempt to modify any state to reflect the fact that an abstract variable and its corresponding fixed value are equal. It is this solver that calls `setValue`(f, v) on the `resultState` to indicate that when it uses a fixed value.

The definition of `copy()` for this solver is straightforward. The methods `join()` and `isMoreInformative()` are much less obvious, and they are currently incomplete. In both cases, the analysis is aborted if two addresses with a nonconstant offset must be compared. A complete implementation would use `SafeMemoryVariablePolicy` to check whether a pointer may point to a subset of the addresses that may be pointed to by the other.

This solver, by itself, does not satisfy the ascending chain condition. However, the current version does not need widening because of the imprecision in `join()` and `isMoreInformative()`. When these methods will be more precise, there will probably be no need for widening either because:

- A strictly less precise state (for an ascending chain) means that an abstract variable may point to more addresses;
- Each abstract variable is associated to at most one abstract address;
- An abstract address points to only one zone;
- An abstract address has an offset that is represented by only one abstract variable expression;
- The constraints placed on an abstract variable expression are kept by the solvers `LinearSolver` and `ModuloSolver`;
- `LinearSolver` implements widening;
- `ModuloSolver` satisfies the ascending chain condition.

Thus, as long as the memory model implemented by this solver and the policies described in Sections 6.1.7 and 6.1.8 remain as restrictive, there should not be any need for widening in this solver. The memory model was inspired by *Alias Types* [SWM00], which is an extension of TAL [MCGW02]. There are, however, many important differences. What is called a *location* in the first article corresponds sometimes to a `MemoryVariable`, and sometimes to a `Zone` in this analysis. In the article, pointers

Table 6.1: Conditional jump in RTL.

```
(set (reg 17)
     (compare (reg 60)
              (mem (reg 59))))

(set (pc)
     (if_then_else (lt (reg 17) (const_int 0))
                   (label_ref 20)
                   (pc)))
```

are to a location, and in this analysis they are to a `Zone` and an offset. This makes it possible to support pointer arithmetic. The article talks about *nonlinear constraints*, which correspond to constraints on `Zone` that have `NumInstances` > 1. This solver provides no support for *option types*, which are possibly NULL pointers. There is also currently no support in the analysis for allocating and freeing memory.

6.1.6 The Comparison Solver

`ComparisonSolver` is defined in `comparison.py`. To understand the usefulness of this solver, it is necessary to look at how conditional jumps are represented in RTL. Table 6.1 shows one. There are two instructions. The first one compares the content of register 60 to that of the memory word pointed to by register 59. The comparison is in fact a subtraction. The result is stored in register 17, which is a “flag” register. The second instruction is the conditional jump and `pc` stands for the current instruction pointer. It means that if the content of register 17 is less than 0, the program jumps to the instruction after label 20, otherwise it continues at the next instruction.

The effect of these two instructions is that the jump is taken if and only if the content of register 60 is less than that of the memory word pointed to by register 59. This condition can be expressed by `LessThan(avr60, avm59)`, where `avr60` and `avm59` are abstract variables for register 60 and the memory word pointed to by register 60 respectively. This constraint would be understood correctly by `LinearSolver`.

The problem is that `LinearSolver` cannot understand the call `setValue(avr17, Minus(avr60, avm59))` that is generated for the first instruction, except in the rare case where `avr60` or `avm59` is known to be constant. This is because it only understands linear relations between two abstract variables, and there are three in this case. Thus, when the second instruction is interpreted, `LinearSolver` does not know the relation between register 17 and the two other abstract variables, and thus it cannot deduce that `LessThan(avr60, avm59)` when the jump is taken, and that `GreaterThanEqual(avr60, avm59)` when it is not.

`ComparisonSolver` tries to improve the situation by keeping information about the last call of the form `setValue(av, Minus(v1, v2))`. It clears its state for any other call to `setValue()`. Then, if there is a call of the form `addConstr(Comparator(av, 0))`, this solver calls `addConstr(Comparator(v1, v2))`. Here, `Comparator` can be any class that inherits from `BinaryComp`.

Since the two instructions that this solver targets to improve the precision of conditional jumps are always next to each other and there is never another instruction that jumps directly to the second one, the implementation of `isMoreInformative()` and `join()` does not have to be very precise. The method `isMoreInformative()` can return `False` for all nontrivial cases, and `join()` can return a solver with no information for these cases. The method `copy()` is simple, as usual.

This solver satisfies the ascending chain condition because it only keeps information about one abstract variable at a time, and `isMoreInformative()` is not precise enough to tell whether an abstract variable expression is more informative than another.

6.1.7 The Abstract Variable Policy

`AbstractVariablePolicy` is implemented in `abstractvariable.py`. It exists mainly to create abstract variable expressions from other representations of values, namely RTL expressions and concrete variable expressions. Abstract variable expressions have already been described in Section 6.1.1 and RTL expressions were described in Sections 3.2 and 4.1. *Concrete variable expressions* are an intermediate state between RTL expressions and abstract variable expressions. They are much like abstract variable expressions, except that abstract variables can be replaced by one of these classes:

- `Register`;
- `ZoneSlice`;
- `ZoneAttr` (`StartOffset`, `EndOffset` and `NumInstances`).

A `Register` is described by its register number, a `ZoneAttr` is described by its type (`StartOffset`, `EndOffset` or `NumInstances`) and a zone, and a `ZoneSlice` is described by a zone, an offset and a length.

Concrete variable expressions form an intermediate representation that has many uses. The first one is to make it possible for an analysis to express constraints (e.g. for preconditions) in a language similar to that of abstract variable expressions, The language of abstract variable expressions is not appropriate because it lacks a way to describe registers and memory accesses. The language of RTL expressions is not appropriate either because it is too different and it lacks the concept of memory zone. Concrete variable expressions allow doing this all at once.

The second use is for debugging. Since abstract variables are not meaningful by themselves, they must often be converted to a more human-readable representation during debugging. Concrete variable expressions are well suited for this purpose and this policy provides a method to convert abstract variable expressions back to concrete variable expressions.

`AbstractVariablePolicy` implements five public methods. The first of them is named `abstractVariableExpr()` and it converts a concrete variable expression to an abstract variable expression. It uses these simple rules:

- $r \mapsto v$ (if $r \in$ `Register`; $v \in$ `RegisterVariable` represents r);
- $a \mapsto v$ (if $a \in$ `ZA` and `ZA` inherits from `ZoneAttr`; $v \in$ `ZoneVariable` represents a);
- $s \mapsto v$ (if $s \in$ `ZoneSlice`; $v \in$ `MemoryVariable` represents s according to `SafeMemoryVariablePolicy`).

Here, `RegisterVariable`, `ZoneVariable` and `MemoryVariable` are all classes that inherit from `AbstractVariable`. Compared to instances of `Register`, `ZoneAttr` and `ZoneSlice`, which are concrete variables, abstract variables have the advantage that they are easier to compare and index for the solvers.

For the first two rules, this policy creates an association between the concrete variable and a new abstract variable if the concrete variable is seen for the first time. For the third rule, things are more complicated because there can be aliases and the offset may not be constant. This is why it resorts to `SafeMemoryVariablePolicy`, which is described in the next section.

The second method, named `abstractVariableExprForRtx()`, converts RTL expressions to abstract variable expressions. The RTL expression is first converted to a concrete variable expression. Much of this conversion is straightforward (e.g. an RTL register is mapped to a `Register`, an RTL addition is mapped to `Plus`...) thus only the rules that are less obvious are shown here.

- `rtl.SymbolRef`$(s) \mapsto$ `AbstractAddress`$(z, 0)$ (where $z \in$ `Zone` is the zone for the symbol s, see `zoneForSymbol()` below).
- `rtl.Ashift`$(x, y) \mapsto$ `Times`$(x, 2^z)$ (if `LinearSolver` of `currentState` knows that $y = z \in$ `Integer`).
- `rtl.Mem`$(x)^l \mapsto$ `ZoneSlice`(z, o, l) (if $a =$ `abstractVariableExpr`(x) and a can be transformed into `AbstractAddress`(z, o) using the rules described in Section 6.1.5 for `PointerSolver`, and l is the size of the memory word in bytes).

The first rule means that each static variable or array is represented by its own zone. The constraints for this zone (start offset, end offset...) must be provided independently. See Section 6.1.12 for how this is done in this analysis.

The second rule is for "arithmetic shift", which is exponentiation. When possible, it is transformed into a multiplication so that solvers have fewer operations to deal with.

The third rule is for pointer indirection and it is by far the most elaborate. Its implementation requires the use of `abstractVariableExpr()` (described above) and code from `PointerSolver`.

If there is some part of an RTL expression for which no conversion rule applies, this policy can return an `UnknownValue` safely, but since it is currently incomplete, it aborts when it encounters an RTL expression it does not know. This is so that we know which RTL expression should be supported next to improve the analysis.

The second step in method `abstractVariableExprForRtx()` is converting the resulting concrete variable into an abstract variable. This is exactly what the method `abstractVariableExpr()` does.

The third method of `AbstractVariablePolicy` is `abstractVariableForRtx()`. It does the same thing as `abstractVariableExprForRtx()`, except that it aborts if the result is not a simple abstract variable. It is used mainly for debugging.

The fourth method is `zoneForSymbol()`. It simply returns the zone that corresponds to a given (static) symbol. If the symbol is seen for the first time, a new zone is created and the association is saved. This method can be used for both real symbols that come from the compiler, and "fake" symbols that represent other memory area. For example, the fake symbol `*locals*` can be used to obtain the zone that corresponds to local variables.

The fifth method is `concreteVariableExpr()`. It converts an abstract variable expression back to a more meaningful concrete variable. However, this conversion is imprecise. For example, a `MemoryVariable` cannot always be converted back to a `ZoneSlice`. It is used only for debugging.

6.1.8 The Safe Memory Variable Policy

`SafeMemoryVariablePolicy` is implemented in `safememoryvariable.py` and it defines three methods to deal with memory accesses. The first method is `mvForSlice()`, which returns a `MemoryVariable` for a given `ZoneSlice`. There are two kinds of `MemoryVariable`: `PreciseMemoryVariable`, which represents memory words at a constant offset in a zone that has a single instance, and `ImpreciseMemoryVariable`, which represents memory words when the offset is not known precisely or when there can be many instances of the zone.

Internally, this policy uses `ModuloSolver` and `LinearSolver` of `currentState` to discover constraints on the offset of a `ZoneSlice`. The class `SubZone` helps putting all this information together. It takes a lower bound, an upper bound, a congruence and a length of the memory word. The operations implemented by `SubZone` are the following.

- `isPrecise()`: Returns whether the offset is constant.
- `isNothing()`: Returns whether the `SubZone` represents no offset at all.
- `overlaps()`: Returns whether two `SubZone` overlap at least partly.
- `isSubSet()`: Returns whether a `SubZone` represents a subset of another.
- `intersections()`: Returns an iterator of `SubZone` that are part of the intersection between two given `SubZone`.

Each `MemoryVariable` is associated with a `Zone` and a `SubZone`, and each `Zone` has a list of associated `MemoryVariable`. When a memory variable is requested for a

`ZoneSlice`, it is first converted into a `SubZone`. The list of existing `MemoryVariable` is then searched for a `SubZone` that is a superset of what is requested. If it is found, it is returned, otherwise an association with a new `MemoryVariable` is created.

This policy also keeps a mapping of possible aliases between all `MemoryVariable`. Thus, when a new `MemoryVariable` is created, the list of existing `SubZone` is scanned and an alias is added for each existing `MemoryVariable` that overlaps the new one. For a `MemoryVariable` in a zone for which `NumInstances` may be greater than 1, an alias is also added between the new `MemoryVariable` and itself. It is important to note that aliases may overlap only partly, e.g. a word of length 4 at offset 0 partly overlaps a word of length 4 at offset 2.

The second method is `sliceForMV(mv)`, which returns the `ZoneSlice` for a given `PreciseMemoryVariable`. It is used only for debugging.

The third method is `aliasesForMV()`, which returns the list of aliases for a given `MemoryVariable`. Note that the "aliases" relation is not transitive. For example, without any constraint on `x`, `ZoneSlice(z, x, 4)` aliases both `ZoneSlice(z, 0, 4)` and `ZoneSlice(z, 4, 4)`, but the latter two are clearly not aliases.

6.1.9 The Fixed Value Policy

`FixedValuePolicy` is implemented in `fixedvalue.py`. The goal of this policy is to allow solvers to keep information about relations between abstract variables at a given point in the program. In the case of `PointerSolver`, the offset of a pointer in a zone is represented by an abstract variable. The problem is that if this abstract variable changes later, the pointer is not affected. Without `FixedValuePolicy`, the relation would no longer represent reality and it would have to be discarded by the solver. This loss of precision is often unacceptable.

`FixedValuePolicy` helps this situation by associating a `FixedValue` (which is another kind of `AbstractVariable`) to each `AbstractVariable` at each edge. An association is created only when the method `getFixedValueFor()` is called for a specific abstract variable so the overhead is minimal. This policy does not update any abstract state; it is up to the solver that requests a `FixedValue` to make sure that this value exists in the right abstract state.

This policy also defines two other methods that are used for debugging. They are `getAbstractVariableFor()` and `getEdgeFor()` which return, respectively, the abstract variable and the edge (or "instruction") at which a `FixedValue` was created.

Note that the other solvers described for this analysis do not need this policy. In particular, `LinearSolver` uses `SolverSymbol` internally, `ModuloSolver` does not keep any relation between abstract variables, and `ComparisonSolver` discards its state at every call to `setValue()`, thus it does not suffer from the problem described above for `PointerSolver`. The main interpreter, however, uses it, see Section 6.1.11.

6.1.10 The Error Log Policy

This policy, defined in `error.py`, provides a method that allows any part of the analysis to log it when an error is detected in the code being analyzed. This method is `logError()` and it takes an instance of a class derived from `DetectedError` as parameter. These classes correspond to different kinds of errors. Currently, they are `MemoryAccessError` and `FunctionCallError`.

The main goal of the analysis described here is to detect memory access errors, and `SafeMemoryVariablePolicy` logs it when a memory access cannot be determined to be safe in the current state. All function calls are also logged as errors because they are ignored by the analysis currently, and thus, there is no way to know if a function call would produce a memory access error or not. Both the main interpreter and `AbstractVariablePolicy` log it when they find function calls.

This policy also provides a method `clear()` to reset the list of errors and a few methods to query the list of errors.

6.1.11 Special Cases to Consider

Many details were left out of the description of the analysis up to now to make it easier to understand. This section discusses the most important of them.

The Interpreter and Imprecise Memory Variables

The main interpreter must be careful when the conversion from an RTL expression to an abstract variable expression returns an `ImpreciseMemoryVariable` because such a value represents many memory words at once, not the particular memory word used (or modified) by the current instruction. Because of that, the interpreter uses `FixedValuePolicy` and it replaces each `imv` ∈ `ImpreciseMemoryVariable` by its corresponding `fv` ∈ `FixedValue` in the expressions returned by `AbstractVariablePolicy`. It also calls `setValue(fv, imv)` for each of them on `currentState` and `resultState` before the main call to `setValue()` or `addConstr()` is executed for the current instruction.

The Comparison Solver and Extended Liveness Analysis

`ComparisonSolver` saves an abstract variable expression when it recognizes a comparison that is likely to be used in a conditional jump. Then, when the jump is taken (or not), the variable expression is transformed into a new expression that is more likely to be understood by the other solvers.

The problem is that, as an optimization, the interpreter tries to remove from the state any information about registers that are dead. If a register is kept by `ComparisonSolver` at a given instruction and it becomes dead at this point, the new constraint generated at the following instruction will be useless.

To fix this problem, a new definition of liveness was used. The only difference is that registers that are used in an instruction that precedes a conditional jump are also considered live in this conditional jump. This is implemented in an analysis called `ExtendedLivenessAnalysis`, which is a slight modification to the liveness analysis described Section 5.2.1.

Widening of Linear Constraints

`LinearSolver` implements widening, but it is incomplete. Currently, it only detects that there is a need for widening on absolute upper and lower bounds of abstract variables. In this case, it simply removes the bound that could prevent convergence. It does not detect if there is a need of widening for linear constraints. This should be fixed in a future version of this analysis.

`LinearSolver` also employs two techniques to decrease the number of widening operations, which affect the precision of the analysis. The first case where no widening is applied is when the result of the abstract interpretation of a new instruction is available at the current instruction for the first time. This tends to decrease the precision of intermediate result states anyway, and we would prefer if things stabilized by themselves instead of applying widening. Note that this case can happen only a finite number of times since there is a finite number of instructions to analyze.

To detect whether the result of a new instruction is available, `LinearSolver` uses two very simple modules. `AccountedEdgeInterpreter` indicates to the solvers which instruction is being interpreted, and `AccountedEdgeSolver` saves this information in the abstract state.

The second case where no widening is applied is when there has been widening between creation of `old` (the last time the instruction was interpreted) and that of `self` (the result of the current interpretation). The idea is that after a widening operation, we want to let the result propagate before resorting to one more widening operation that could be costly in precision.

6.1.12 The Memory Access Analysis

The main analysis class is implemented in `memoryanalysis.py`. It uses the interpreters, solvers and policies discussed in the last few sections. The extremal value of this analysis represents the input that the function will receive when it is called. It is different for each function because it must describe the pointers and the memory zones that are used. Among other things, it must describe the pointer and the zone for local variables of the function and the pointer and the zone for the parameters of the function.

A large part of this information could be extracted from the source code of the program at the same time as the RTL code, but currently it is not. Instead, the size of parameters and local variables must be provided in the file `datadb.py`. It is also possible to describe the possible values of a parameter or a value in memory, and even some complex data structures and relations between zones and pointers.

For each instruction (or edge) that is analyzed, the abstract interpretation is done on the result of the `join()` operation between all its predecessors. Then, widening and narrowing are applied using the result of the last iteration. The successors of the current instruction are added to the queue of instructions to process, except if the last and the new abstract states are equal. Since the analysis stops only when all abstract states have stabilized, it means that the solvers must satisfy the *descending chain condition* in addition to the ascending chain condition or they must implement narrowing.

At the end of the analysis, the list of errors of `ErrorLogPolicy` is cleared and each instruction is interpreted one last time so that errors are reported correctly. This is important because during the main loop, some errors can be reported because of the lack of precision that results from widening, but often, enough precision is recovered after and "errors" disappear. The list of errors is the result that is produced by this analysis.

An important question is whether this analysis is correct. In this case, the correctness property is:

> Given a function that is analyzed, for any execution of the function that has an input state described by the extremal value of the analysis, if the analysis does not report any error then the execution will not have any memory access error.

Note that the function that is analyzed does not necessarily terminate. However, the analysis should always terminate. In its current state, we are not sure that this analysis always terminates because the implementation of widening for `LinearSolver` is incomplete. Thus, we cannot claim that this analysis always terminates. Still, we have not observed any case where it does not.

There are other issues that could affect whether the analysis terminates or not. For most solvers, the argumentation of whether or not they satisfy the ascending chain condition and whether not satisfying it can cause problems is informal. In particular, it is not obvious whether `PointerSolver` would need widening if its precision was improved. Concerning narrowing, we have not yet encountered any situation where it was necessary, but `LinearSolver` and `ModuloSolver` do not satisfy the descending chain condition and they do not implement narrowing; this could affect termination. Also, it is not clear whether the transfer functions implemented by the interpreter are monotone, because they use policies that have states which can change over the course of the analysis.

If we assume that the analysis terminates, there is the more important question of whether its result can be trusted; that is to say, is the function necessarily memory-safe when the analysis does not return any error? The main problem with the analysis is that integers are assumed to be of an arbitrary precision, which is certainly not the case in the low-level RTL code that is analyzed. If there is an integer overflow during execution, it is not modeled correctly by the analysis, something that could lead to a memory access error left undetected by the analysis. Therefore, we cannot claim that this analysis is correct and it must be improved before its result can be fully trusted.

If the problem of integer overflow is fixed, it does not mean that the analysis will be automatically correct. There is currently no formal definition of the semantic of RTL available and our description of the different solvers is also highly informal. Thus, much work is needed on formalization before the correctness of this analysis can be established. The fact that, in addition to the interpreter, some solvers also modify the abstract state could complicate their formalization.

There is another aspect of this analysis that does not affect its correctness, but could affect its precision. It is the order in which the instructions are analyzed. When there is widening, an instruction should not be analyzed again before all the instructions that can affect it have been analyzed and their result have been propagated to it. The goal is to avoid widening more than necessary. The current queue of instructions to analyze does not try to ensure that. The fact that it does not even guarantee that the instructions are analyzed in the same order every time the analysis is run could theoretically lead to some cases where a function is determined to be safe only one time out of two. This would still mean that the function is safe.

6.2 Results

This section presents some examples of code that can or cannot be determined safe by this analysis. It also shows some examples of code that is correctly identified as unsafe. In all cases, we present C source code, but this code was transformed into RTL by GCC before it could be analyzed.

6.2.1 What Works

First, we present in Table 6.2 a simple function that has an obvious memory access error, which is correctly identified.

This function tries to change the eleventh element of an array that has only 10. This is wrong, and the analysis catches it. To get this result, we only had to indicate in `datadb.py` that the symbol `array` corresponds to a zone of 40 bytes (10 4-bytes integers). The message tells that there is a memory access error in the zone named `Sym_array`, i.e., the zone that corresponds to symbol `array` (4 is an internal ID for the zone). The function tries to access the byte at offsets 40, 41, 42 and 43, but the valid offsets are only between 0 and 39.

Table 6.2: Analysis of a simple function with a memory access error.

The function analyzed:

```
static int array[10] = {1,2,3,4,5,6,7,8,9,10};

void simple_error()
{
    array[10] = 1;
}
```

The diagnostic of the analysis:

```
Analyzing simple_error ...
Error: line 5 : Memory access error in zone Zone(4_Sym_array):
word of length 4 at offset 40 may be outside the bounds [0,40]
```

The next function, in Table 6.3, is also pretty simple and this time it has no memory access error.

This function uses one local variable, thus we had to specify in `datadb.py` that it uses 4 bytes of local memory in addition to the zone of the array. This function is interesting because it requires a minimum of precision from the analysis to be analyzed correctly. We observe that there is an instruction that is a memory access error, but it is never executed because the condition in the `if` is always false. However, to detect that, the analysis must do many things. If it lacked any of the following items, it could not tell that the function is safe.

- It must detect that the first use of the pointer `p` is an alias to `array[0]`.
- It must detect that the second use of the pointer `p` is not an alias to `array[0]`.
- It must use the condition in the `if` to create the correct constraint for the true branch.
- It must detect that the condition can never be satisfied.
- It must know that an instruction on a path that is impossible cannot generate a memory access error.

The next function, in Table 6.4, is a little more complicated. To correctly analyze

Table 6.3: Analysis of a safe function with aliases and dead code.

The function analyzed:

```
static int array[10] = {1,2,3,4,5,6,7,8,9,10};

void write_global()
{
    int *p;
    array[0] = 1;
    array[1] = 2;
    p = &array[0];
    *p = 42;
    p++;
    *p = 3;
    if (array[0] < 42)
        array[10] = 1; // Error
}
```

The diagnostic of the analysis:

```
Analyzing write_global ...
this function is safe for the specified preconditions
```

Table 6.4: Analysis of a function with a loop and a nontrivial precondition.

The function analyzed:

```
// The first element is the size
int array_sum(int *a)
{
    int res = 0;
    int i;
    for (i = 1; i<*a; i++)
        res += a[i];
    return res;
}
```

The diagnostic of the analysis:

```
Analyzing array_sum ...
this function is safe for the specified preconditions
```

this function, we must specify in `datadb.py` that it uses 4 bytes of arguments and 8 bytes of local variables, but this is not sufficient. We must also indicate that the first (and only) parameter is a pointer to a zone that we will call `z1`, that this zone must have at least one 4-byte word at offset 0, and that the first element of the array indicates its length, including the first element. With `za` representing the zone of arguments and `z1` representing the zone pointed to by the first argument, the list of constraints is:

- `Equal(ZoneSlice(za, Integer(0), 4), AbstractAddress(z1, Integer(0)))`;
- `LessThanEqual(StartOffset(z1), Integer(0))`;
- `GreaterThanEqual(EndOffset(z1), Integer(4))`;
- `GreaterThanEqual(EndOffset(z1),`
  `Times(ZoneSlice(z1, Integer(0), 4), Integer(4)))`.

These constraints are converted into abstract variable expressions and they are added to the extremal value of the analysis using `addConstr()`.

This function is also interesting to analyze. There must be widening because we have no integer upper bound for `i`. However the widening must not discard the precondition

Table 6.5: Analysis of a function with a circular linked list.

The function analyzed:

```
struct IntList
{
    int i;
    struct IntList *next;
};

int circular_sum(struct IntList *l)
{
    struct IntList *first=l;
    int res = first->i;
    for (l=first->next; l!=first; l=l->next)
        res += l->i;
    return res;
}
```

The diagnostic of the analysis:

```
Analyzing circular_sum ...
this function is safe for the specified preconditions
```

because the analysis needs it together with the condition of the loop to verify that the function is safe. Note that the analysis of this function must be done completely symbolically because of the lack of integer bounds for the size of the array.

The next function, in Table 6.5, uses a circular linked list. Just like for `array_sum()`, we must specify in `datadb.py` that this function uses 4 bytes of arguments and 8 bytes of local variables, but also some constraints to describe the pointer received as parameter and the recursive structure it points to. With `za` representing the zone of arguments and `zc` representing the circular structure, the list of constraints is:

- `Equal(ZoneSlice(za, Integer(0), 4), AbstractAddress(zc, Integer(0)));`
- `LessThanEqual(StartOffset(zc), Integer(0));`
- `GreaterThanEqual(EndOffset(zc), Integer(8));`
- `Equal(ZoneSlice(zc, Integer(4), 4), AbstractAddress(zc, Integer(0))).`

Table 6.6: Analysis of a function with a linked list.

The function analyzed:

```
struct IntList
{
    int i;
    struct IntList *next;
};

int list_sum(struct IntList *l)
{
    int res = 0;
    for (; l; l = l->next)
        res += l->i;
    return res;
}
```

The diagnostic of the analysis:

```
Analyzing list_sum ...
Warning: line 11 : Memory access error in zone Zone(1_Unknown):
word of length 4 at offset 0 may be outside the bounds [Unknown,Unknown]
Warning: line 10 : Memory access error in zone Zone(1_Unknown):i
word of length 4 at offset 0 may be outside the bounds [Unknown,Unknown]
```

With these constraints, the analysis is able to determine that this function is safe. Note that we do not have any constraint on `NumInstances(zc)`, thus the analysis cannot assume that two pointers to `zc` are equal. Each element of the list is another instance of `zc`, even if they are all represented by the same zone.

6.2.2 What Does Not Work

Table 6.6 shows a function that cannot be determined safe by the current analysis. It looks much like `circular_sum`, but here the list ends with a `NULL` pointer instead of being circular.

This function cannot be analyzed correctly because currently, we have no way to express the fact that `next` can be either a pointer to an instance of the structure or

Table 6.7: Analysis of a function with null-terminated strings.

The function analyzed:

```
char *strcpy(char *dest, const char *src)
{
    char *d = dest;
    while (*src != '\0') {
        *d= *src;
        d++;
        src++;
    }
    *d = '\0';
    return dest;
}
```

The diagnostic of the analysis:

```
Analyzing strcpy ...
Warning: line 5 : Memory access error in zone Zone(1_Unknown):
word of length 1 at offset 0 may be outside the bounds [Unknown,Unknown]
Warning: line 4 : Memory access error in zone Zone(1_Unknown):
word of length 1 at offset 0 may be outside the bounds [Unknown,Unknown]
Warning: line 9 : Memory access error in zone Zone(1_Unknown):
word of length 1 at offset 0 may be outside the bounds [Unknown,Unknown]
Warning: line 5 : Memory access error in zone Zone(1_Unknown):
word of length 1 at offset 0 may be outside the bounds [Unknown,Unknown]
```

`NULL`. This is why the analysis reports an error with an access to an unknown zone with unknown bounds. In the message, the offset 0 is arbitrary.

Another function that cannot be determined safe by the current analysis is shown in Table 6.7. This one manipulates null-terminated strings. This function cannot be analyzed correctly because currently, we have no way to express the fact that the offset of the first null character represents the length of the string or that of the zone. Thus, the analysis reports an error for each use of a pointer, that is, four times.

Table 6.8 shows yet another function that cannot be determined safe by the current analysis. This time, it is because function calls are not supported. This is reported by the analysis because we cannot consider that a function is memory-safe if any of its

Table 6.8: Analysis of a function with a function call.

The function analyzed:

```
void call(void)
{
    char buf[10];
    strcpy(buf, "hello");
}
```

The diagnostic of the analysis:

```
Analyzing call ...
Warning: line 49 : Function call to strcpy not supported
```

calls is unsafe.

6.2.3 Performance

For the simple functions that we tested, the time required to perform the analysis varied greatly. It took anywhere from 0.01 second to 0.90 seconds on a 2 GHz AMD Sempron CPU. The factors that affect the most the duration of the analysis seem to be:

- The length of the function to analyze;
- The number of variables used by the function;
- The complexity of the extremal value (which describes the input of the function);
- The presence of loops;
- The use and manipulation of pointers;
- The existence of many variables that are linearly related.

It is easy to understand why most of these factors affect the length of the analysis. When there is a loop, the analysis must often analyze the instructions it contains many times before the abstract state stabilizes. The use of pointers can slow down the analysis because of the way new constraints are handled in `PointerSolver`. For linearly related variables, the methods `join()` and `widen()` of `LinearSolver` can also be slow.

The RTL-Check framework and the memory access analysis were completely implemented in Python. Python is an interpreted language that is usually considered slow. Therefore, to speed up the analysis, it would be possible to optimize the parts that are known to take much time, but it would also be possible to rewrite it in a language that has a better run-time performance. It is important to keep an eye on the performance of the analysis, because as its precision will improve, its performance will most likely decrease.

In the current version, the main optimization is that the interpreter removes the dead registers from the abstract state by calling `setValue()` with unknown values. One function in particular takes 106 seconds, instead of 0.9, when analyzed without this optimization. It has 13 lines including two `if` statements and one `for` statement. It uses two parameters, two local variables and an array of 10 variables.

6.3 Possible Improvements

There are many areas in which the analysis could be improved. Concerning correctness, something must be done to detect integer overflows that could affect memory safety. For ensuring termination, the implementation of widening for `LinearSolver` and that of narrowing for `LinearSolver` and `ModuloSolver` must be completed. From a theoretical point of view, much work is needed to formalize RTL and the different parts of this analysis, and to obtain complete proofs of termination and correctness for the analysis. This was discussed in Section 6.1.12.

There are also some parts of the analysis that are not completely implemented; there are many assertions that force abortion when some case that is not yet handled is encountered. For example, not all RTL expressions are known by the interpreter currently. Improving the analysis to handle these cases would make it possible to analyze more programs.

Concerning precision, many things can be improved. For example, the `join()` method of `LinearSolver` and `PointerSolver` could return a more precise result in some situations. It would be very interesting to support function calls, because, in practice, no useful program can be written without any function call. Also, existing solvers should be improved or new solvers should be created to make it possible to verify that functions dealing with more complex structures are safe. This was discussed in Section 6.2.2.

Finally, concerning performance, Section 6.2.3 hints at some functions that could be optimized. There are probably some other areas of the analysis that would benefit from optimization.

Chapter 7

Metaprogramming and the Visitor Design Pattern in RTL-Check

In this chapter, we discuss how metaprogramming is used in RTL-Check and the benefits it brings. Moreover, we describe how metaprogramming can automate the implementation of an improved visitor design pattern.

7.1 About Metaprogramming

Metaprogramming is [Wik]:

> the writing of programs that write or manipulate other programs (or themselves) as their data or that do part of the work that is otherwise done at runtime during compile time.

There are many kinds of metaprogramming. A compiler is a metaprogramming tool because it translates a source program into a binary one. Thus, by nature, writing a compiler is metaprogramming. Creating a tool that does static analysis of programs is also metaprogramming by nature, however this chapter is not about metaprogramming at such a high level.

Some approaches to metaprogramming at a lower level include reflection, templates in C++, higher level functions in functional languages and macros in LISP and its derivatives. Python is a great language for metaprogramming. It offers a good reflection model, higher level functions and metaclasses, a lesser known feature that allows creating classes at run time. Python is also a very dynamic language in that it makes it possible to modify classes at run time, even after instances have been created.

RTL-Check uses metaprogramming to generate and modify some classes and to automate the implementation of an improved visitor design pattern. In each case, metaprogramming could be replaced by equivalent "non-meta" code. The following sections explain how metaprogramming is used in RTL-Check and the advantages of using it.

7.2 Using Metaprogramming to Generate Classes

The following sections explain the different techniques used in RTL-Check to generate classes dynamically and why this is useful.

7.2.1 The Rtx Class Hierarchy

The `Rtx` class hierarchy is an object oriented model to represent RTL, the low-level intermediate language used by GCC. The C structure of GCC to represent RTL is essentially an integer indicating the RTL code followed by a list of operands. Each RTL code has a fixed number of operands, each with a predefined type. The supported types for operands are:

- Integer;
- String;
- Pointer to another RTL expression;
- Vector of pointers to RTL expressions.

Each class in the `Rtx` hierarchy represents one or more RTL codes that are conceptually related and which have some field in common. Instead of using a position number to access the operands of an RTL expression (as with the C structure) the classes expose them via field names.

Table 7.1: Example of a class in the Rtx class hierarchy (RtxGenBinArith).

```
class RtxGenBinArith(RtxGenArith):
    @property
    def op1(self):
        return self.xexp(0, 'e') # Operand 0 (an RTL expression)
    @property
    def op2(self):
        return self.xexp(1, 'e') # Operand 1 (an RTL expression)
```

Table 7.2: Declaration of classes in the Rtx hierarchy using a function.

```
_createRtxSubclass('GenBinArith', RtxGenArith,
                   (rtldef.PLUS, rtldef.MINUS, rtldef.MULT),
                   (('op1', 0, 'e', 'xexp'),
                    ('op2', 1, 'e', 'xexp')))
_createRtxSubclass('Plus', RtxGenBinArith, rtldef.PLUS)
```

The hierarchy has multiple levels to make it possible to deal with only one specific RTL code as well as many related ones, depending on the situation. For example, the class RtxGenBinArith represents all the RTL codes that are binary arithmetic operations. This class exposes two fields, op1 and op2, both of which are RTL expressions. The class RtxPlus inherits from RtxGenBinArith and represents only one RTL code, the addition. It does not add any new field. At the top-level of the hierarchy there is the Rtx class that represents all RTL codes. Table 7.1 shows what an Rtx class should look like.

All classes in the hierarchy are based on the same pattern, and there are many RTL codes that need there own class. Thus, they are good candidates for metaprogramming. Instead of creating them one by one, it is possible to write a function that creates them. This function knows the main structure of the classes it creates, and it fills the blanks with the parameters it receives. With this function available, creating a new Rtx-derived class is possible without repeating the common code, as shown in Table 7.2.

The function _createRtxSubclass() takes 4 parameters. The first one is the name of the class that must be created, without the 'Rtx' prefix, and the second one is its base class (it can be Rtx). The third parameter is the list of RTL codes represented by the new subclass (more on that later). The fourth parameter describes the list of fields of the new subclass. For each of them, there is a tuple which consists of the name of the field, its position, its type from the format string of the C structure, and its type in RTL-Check.

Using this function to declare classes has many advantages over the traditional way of doing so. It does not only reduce the amount of code, it also improves its legibility because there is less noise surrounding the description of each class. This becomes much more obvious when looking at a long list of class declarations. Another advantage is that it makes it unlikely to have a subtle bug in one of the classes because the pattern was not followed correctly. It also makes it trivial to change the pattern for all classes at the same time.

In addition, the function can do more than just create a new class. In this case, it also associates the class to each RTL code listed in the third parameter. This association is important when an instance of an `Rtx` class must be created from a raw dump file. Given an RTL code, we must be able to create an instance of the `Rtx` class that will be the most meaningful to represent the expression. Thus, this function is more than a simple macro, as can be seen in Table 7.3.

The most important part of the `_createRtxSubclass()` function is the call to `type`. In fact, `type` is not a function. In Python terminology it is a metaclass, i.e., a class which has other classes as its instances. Thus, the call to `type` is technically an instantiation, but for all practical purposes, it can be considered a function call. It returns a new class containing methods that are specified in a dictionary. The name of a method is a key in this dictionary, and its implementation, given as a lambda expression, is the corresponding value.

After a class is created, it is given a name in the global namespace of the current module. The last step is associating the class to the RTL code it represents.

7.2.2 Metaclass for Analyses

In Python, it is possible to change how classes are created with custom metaclasses instead of the default `type` metaclass. This feature has proven very useful in RTL-Check, in particular for classes that implement analyses, i.e., those that inherit from `BaseAnalysis`. This section describes `AnalysisMeta`, the metaclass of `BaseAnalysis` and all the classes that inherit from it. This metaclass implements the core of the RTL-Check framework.

Table 7.3: Function that creates Rtx subclasses.

```
# This function creates a subclass of Rtx
def _createRtxSubclass(suffix, base, crange, methods=()):
    name = 'Rtx' + suffix

    # Create the members dictionary for the class
    memberDict = {}
    for (xname, element, format, xtractor) in methods:
        xget = {'xint': lambda self, el=element, fmt=format:
                            self.xint(el, fmt),
                'xstr': lambda self, el=element, fmt=format:
                            self.xstr(el, fmt),
                'xexp': lambda self, el=element, fmt=format:
                            self.xexp(el, fmt),
                'xvec': lambda self, el=element, fmt=format:
                            self.xvec(el, fmt),
               }[xtractor]
        memberDict[xname] = property(xget)

    # Create a subclass of Rtx
    subclass = type(name, (base,), memberDict)

    # Add it to this module
    globals()[name] = subclass

    # Set the new constructor for all RTL codes in crange
    if isinstance(crange, int):
        _rtxConstructor[crange] = subclass
    elif isinstance(crange, tuple):
        for code in crange:
            _rtxConstructor[code] = subclass
    else:
        raise TypeError("_createRtxSubclass: crange has wrong type")
```

Table 7.4: Constructor of `AnalysisMeta` (beginning).

```
class AnalysisMeta(type):
    def __init__(cls, name, bases, dict):
        type.__init__(cls, name, bases, dict)

        . . .
```

The only time at which the metaclass gets involved is when a class that inherits `BaseAnalysis` is declared (created). At this time, the constructor (`__init__()`) of the metaclass is invoked. Remember that a class is an instance of its metaclass. The constructor receives as parameter the object that will represent the new class, its name, the list of its base classes and a dictionary containing its attributes (mostly methods). The argument that the constructor receives usually comes from the class declaration, i.e., the `class` instruction.

The first thing the metaclass does is delegating a large part of the work required for the creation of the new class to `type`, its base class. This is shown in Table 7.4.

The next step in the creation of the new analysis is sorting the dependencies between analyses. In Section 5.1.5 we explained how analyses, policies, interpreters and solvers can declare that they depend on other analyses with their static member `_requiredAnalysis`. Table 7.5 shows how the metaclass sorts the dependencies between everything.

It produces a tuple (which can be seen as a list) of 2-tuples, each containing an analysis class and its name (modified so that the first letter is lower case). The analyses are sorted so that they appear after their dependencies. In `_iterAllAnalyses`, the class `LIFOQueue` implements a LIFO queue (last in, first out) which is special in that it keeps only one copy (the last one) of elements that are added many times to it. Thus, after the call to `_sortRec`, the queue contains all the analyses in the dependency tree, and an analysis without dependency will be popped first.

A similar process is used to sort the dependencies between policies, solvers and interpreters. However, in these cases, the boundary of analyses is not crossed because these modules are local to the analysis that needs them. Another not so different process is used to collect the list of interfaces that the abstract state class must implement.

Table 7.5: Constructor of `AnalysisMeta` (iteration over analyses).

```
# Returns an iterator over analyses, starting with those that
# have no dependency. All analyses are searched recursively
def _iterAllAnalyses(cls):
    def _sortRec(cls, q):
        for interp in getattr(cls, '_requiredInterpreters', ()):
            _sortRec(interp, q)
        for policy in getattr(cls, '_requiredPolicies', ()):
            _sortRec(policy, q)
        for solver in getattr(cls, '_requiredSolvers', ()):
            _sortRec(solver, q)
        for base in cls.__bases__:
            _sortRec(base, q)
        for analysis in getattr(cls, '_requiredAnalysis', ()):
            q.add(analysis)
            _sortRec(analysis, q)
    q = LIFOQueue()
    _sortRec(cls, q)
    return (q.pop() for i in xrange(len(q)))

class AnalysisMeta(type):
    def __init__(cls, name, bases, dict):
        . . .

        # Tuple of (Analysis, name)
        _sortedAllAnalyses = tuple((x, x.__name__[0].lower()
                                        + x.__name__[1:])
                                    for x in _iterAllAnalyses(cls))

        . . .
```

Table 7.6: Constructor of `AnalysisMeta` (creation of `_Interpreters`).

```
class AnalysisMeta(type):
    def __init__(cls, name, bases, dict):
        . . .

        # We create an Interpreters class for cls
        class _Interpreters(object):
            def __init__(self, analysis):
                for interp,name in _sortedInterpreters:
                    setattr(self, '_'+name, interp(analysis))
            def interpret(self, insn):
                for interp,name in _sortedInterpreters:
                    getattr(self, '_'+name).interpret(insn)
        # Add a property get for each interpreter
        for _,name in _sortedInterpreters:
            setattr(_Interpreters, name,
                    property(lambda self, name=name:
                            getattr(self, '_'+name)))
        setattr(cls, '_Interpreters', _Interpreters)

        . . .
```

Once all information about the dependencies is collected, the metaclass starts adding new attributes to the new class. Many of them are needed by `BaseAnalysis`, which cannot provide an implementation for them because they are specific to a given analysis. For example, an attribute named `_Interpreters` is added to the new class. It is a class that represents all the interpreters required by the analysis. Its constructor creates an instance of each interpreter required by the analysis. It also provides a method named `interpret()` which calls that of each interpreter. Moreover, it provides one get-property to access each interpreter. `BaseAnalysis` cannot implement such a class. Table 7.6 shows how the metaclass implements all this.

The metaclass uses similar code to add a class that represents the abstract state (`_State`, composed of solvers) and another representing policies (`_Policies`). It also adds get-properties that make it possible to access the different policies and interpreters directly from the analysis, and a method named `_initAllAnalyses()` that is called by the constructor of `BaseAnalysis` for top-level analyses.

If metaclasses were not available, it would still be possible to implement most of the RTL-Check framework in the constructor of `BaseAnalysis` because Python makes it possible to add attributes to classes and instances, even after an instance is created. However, there would be the following drawbacks.

- The constructor runs once for each instance of analysis class that is created instead of once for each kind of analysis. This could become a performance problem if it was not handled properly.
- Mixing normal code of the constructor with code that modifies the class of which an instance is being created can be confusing. It would make the class more difficult to understand, and the framework would be less maintainable.

If we used a language that does not provide a replacement for Python's metaclasses and which is not as dynamic as Python, we would have to resort to different approaches to implement the framework. For example, we could use a method that takes the name of the interpreter that the caller wants as a parameter instead of providing a get-property named after the interpreter that returns it directly. If everything else failed, we would have to implement a framework doing less work, and thus delegating more work to the programmer that uses it to implement a new analysis.

7.3 Automatic Implementation of the Visitor Design Pattern

Another interesting use of metaprogramming in RTL-Check is for implementing the visitor design pattern automatically. RTL-Check uses this pattern extensively for its `Rtx` class hierarchy, its abstract variables and abstract variable constraints. For example, the core of an interpreter is often implemented as a visitor, which can itself use other visitors to complete its work. The different solvers also use visitors to implement the transformation rules that were described in Section 6.1 for linear derivations, congruences and pointers.

7.3.1 Description of the Visitor Design Pattern

The visitor design pattern [GHJV94] is useful when there are many operations that must be performed on a complex data structure. Let us use the `Rtx` class hierarchy as an example. Currently it contains more than 40 classes that represent as many different kinds of RTL expressions. This is in addition to the classes allowing them to be grouped logically, which we will ignore for now. RTL expressions are composed and linked together to form a structure that represents a program.

Suppose that we must implement two operations on it: `print` and `interpret`. Each kind of RTL expression will have to be printed or interpreted differently, so it is not possible to rely only on methods in the `Rtx` base class to do anything useful. One obvious solution is to write two methods (`print()` and `interpret()`) in each class of the `Rtx` hierarchy. This approach has a few problems. The main ones are that the code implementing a single operation is scattered among many classes, which makes it more difficult to understand when looking at the code. Also, if the operation needs to maintain some state, there is no central place for it; it must be passed as parameter between methods. Moreover, if a new operation had to be written, it would require modifications to all of the 40 classes that represent RTL expressions.

The visitor design pattern solves all of these problems. Figure 7.1 shows the UML model for this pattern applied to `Rtx` classes. The idea is that each operation is implemented in one class, called a visitor. This class implements an interface (`IRtxVisitor`) containing one method (`visit()`) overloaded for each class it must work with, i.e., those that implement `IRtxElement`. This interface contains a method `accept()` that calls the right method `visit()` on the visitor passed as parameter. It makes it possible to invoke an operation on any element without knowing the exact type of element we are dealing with. Ultimately, the method invoked depends on both the kind of element and the kind of visitor.

7.3.2 Automatic Implementation

The visitor design pattern is good, but it is not perfect. One of its main disadvantages is that it requires much code just to forward calls from `accept()` to `visit()`. Table 7.7 shows the C++ code that implements `accept()`. This is always the same code, but it cannot be shared in a base class because for each class we want the compiler to generate a call to a different `visit()` method, the one with a parameter that matches the real type of `this`.

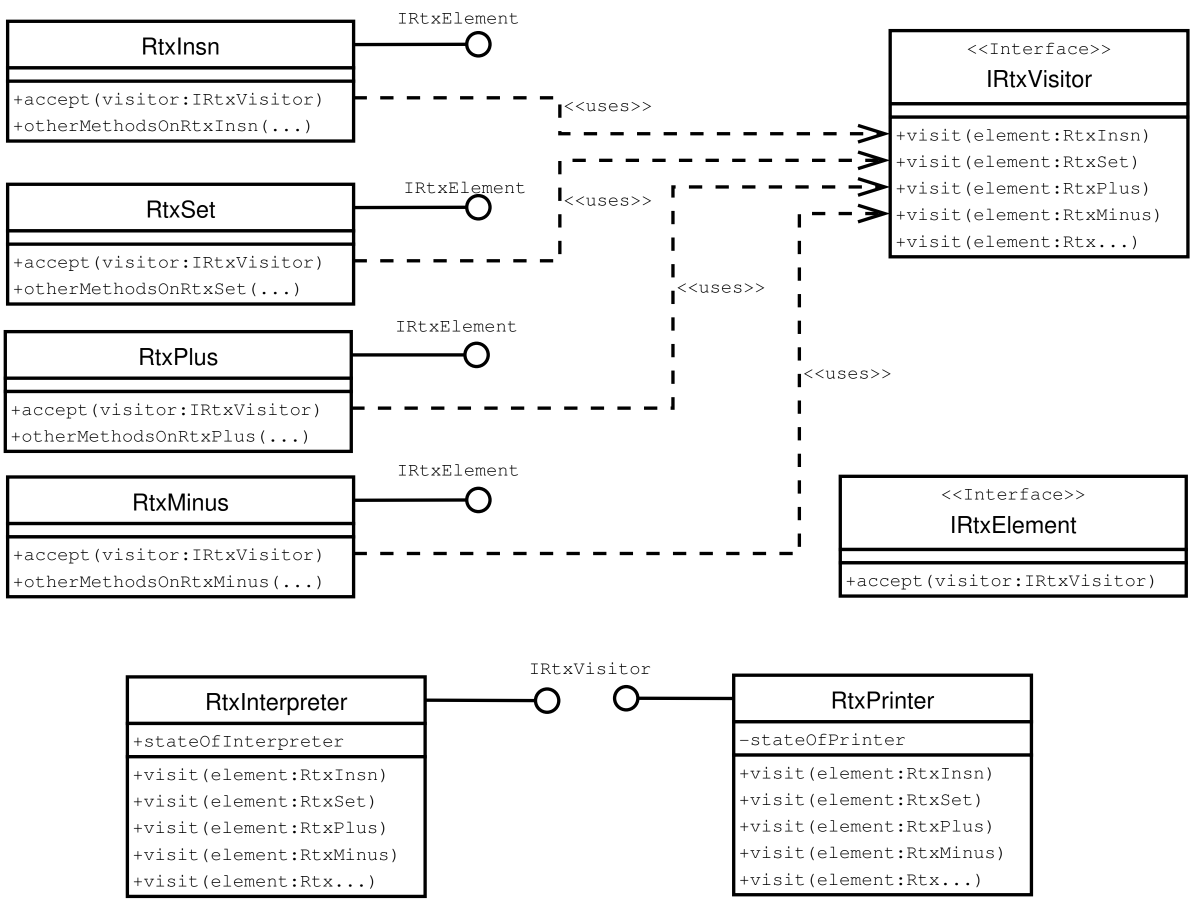

Figure 7.1: The classic visitor design pattern applied to `Rtx` classes.

Table 7.7: Implementation of `accept()` in C++.

```
void accept(IRtxVisitor visitor)
{
    visitor.visit(this)
}
```

Table 7.8: Automatic implementation of the visitor design pattern.

```
# This function implements the visitor design pattern
# for a given list of classes
def _createVisitor(classes, visitorName):
    visitorMethods = {}
    for cls in classes:
        # Add an accept method to the existing class
        visitName = 'visit' + cls.__name__
        def accept(self, visitor, visitName = visitName):
            visitMethod = getattr(visitor, visitName)
            visitMethod(self)
        setattr(cls, 'accept', accept)

        # Create a visit method for the interface
        def visit(self, obj, visitName = visitName):
            raise RuntimeError(visitName + " not implemented in "
                               + str(type(self).__name__))
        visitorMethods[visitName] = visit

    # Create and return the interface
    return type(visitorName, (object,), visitorMethods)
```

Metaprogramming is useful to save the programmer from writing this boring code. It is also faster and less error-prone. For example, many frameworks use source code generation to automate the creation of `accept()` methods in C++.

In Python, it is not possible to overload methods, so the pattern must be adapted slightly. Each `visit()` method is suffixed with the type of parameter it receives. For example, the method that visits classes of type `RtxPlus` is named `visitRtxPlus()`. Figure 7.2 shows the new UML model.

Since Python is a dynamic language, we can generate much of the code needed for the pattern at run time. Table 7.8 shows one possible implementation.

With this implementation, all the classes that must be visited (e.g. `RtxInsn`, `RtxPlus`, etc.) can be created without thinking about the visitor design pattern. The function `_createVisitor()`, receives a list of elements (classes) that must be made visitable.

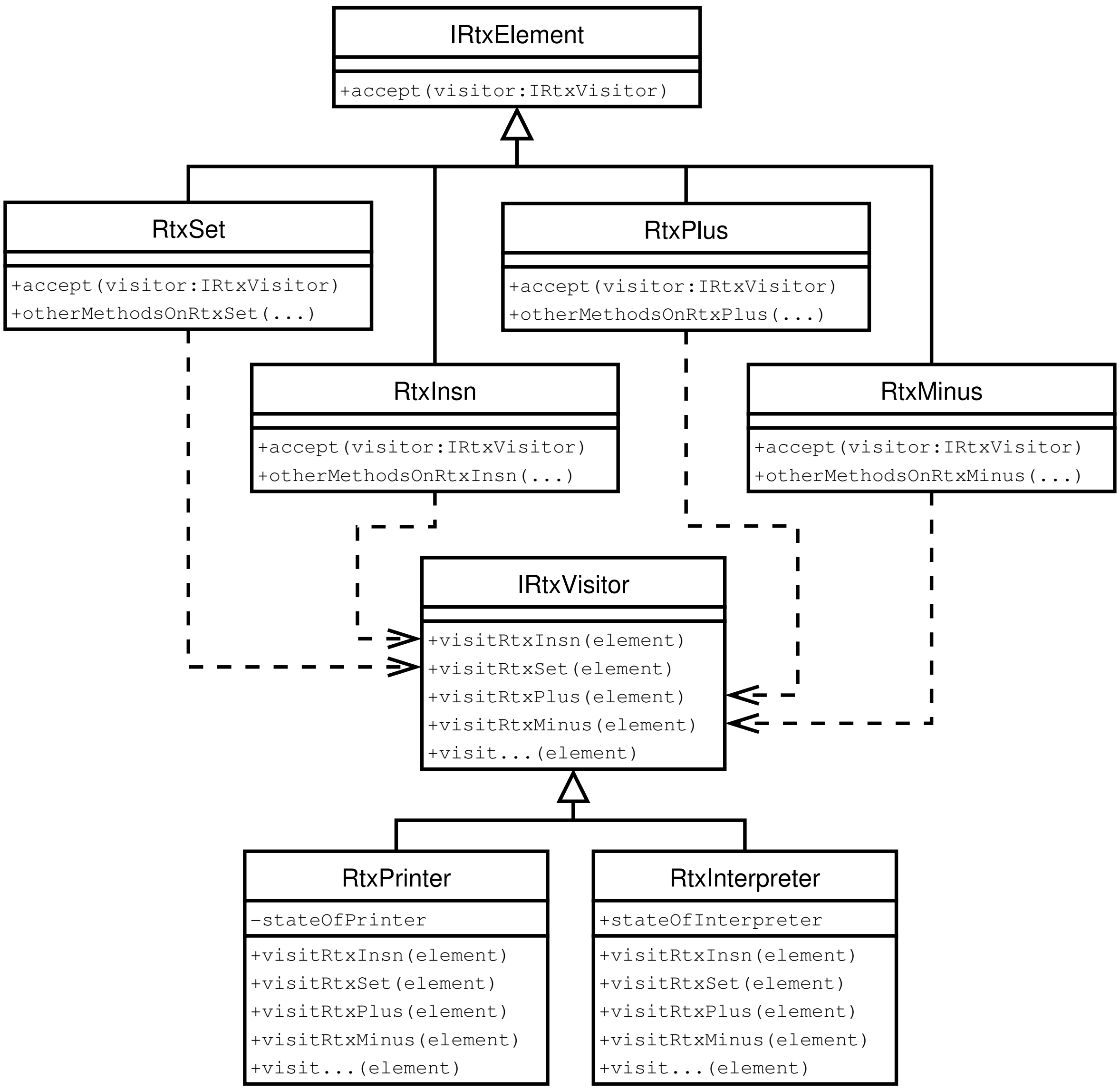

Figure 7.2: The visitor design pattern applied to `Rtx` classes in Python.

For each of them, it adds an `accept()` method on the element and a `visit()` method on the class that represents the interface of the visitor. This method should never be called because a visitor should handle all cases, but if it is, it aborts with a message that indicates which method is not implemented.

RTL-Check does not use this implementation. Instead, an improved version, described in Section 7.4.2, is used.

7.4 Improved Visitor Design Pattern

Automatic code generation alone does not solve all the weaknesses associated with the visitor design pattern. One of them is that if the `visit()` method for many classes is the same, it must still be repeated in the visitor for each of them.

Another weakness is that if a new visitable class is added, every visitor must be modified. It is possible to provide an `accept()` method automatically for the new class, but it is not possible to automatically add support for this class to every existing visitor.

Yet another weakness is that `visit()` methods cannot return information directly. They must use some member variable of the visitor to save information that will be needed later. For example, it is possible to use something like `self.returned_value` instead of returning a value directly, but it is not the best solution when this value has a meaning only for the caller of `visit()`, not for the visitor as a whole.

7.4.1 Returning Information Directly

Allowing `visit()` methods to return a value is easy and it is something we see often in an implementation of this pattern. We just have to ensure that `accept()` returns whatever `visit()` returns. In Python this does not cause any problem because the language is dynamically typed and the interpreter will always be happy to return any value.

The `visit()` method is always free to not return anything if it does not need to. Obviously, when it does return a value, the caller must have some way to know what the value means and how it can be used. Usually, all the methods of a given visitor return a value of the same type.

Table 7.9 shows a part of the class `AbstractAddressEvaluator`, which implements the rules presented in 6.1.5. The goal of this visitor is to transform an arbitrary abstract variable expression into another one that is an `AbstractAddress`. Thus, it is natural to write every `visit()` method such that it returns an abstract variable expression. When the transformation fails, it returns `None`.

In a language such as Java or C#, where every type inherits from something like `Object`, it is possible to define every `accept()` and `visit()` method to return a value of this type. This has the disadvantage of requiring an explicit cast before using the returned value, but it is better than no return value at all. With these languages, it would be difficult to declare a more specific return type because `accept()` must be independent of the visitors and the signature of two methods having the same name cannot differ only by their return value. In some cases, when using code generation for a visitor returning a value of a known type, it would be possible to generate an additional `accept()` method that takes as parameter this specific visitor and that returns an object of this specific type. It would require some changes to the elements that can be visited every time there is a new visitor, but since this code is generated automatically, it is a lesser problem.

In a language such as C++, there is no type that is common to all values, so we could not have a single `accept()` method that works for any kind of return value. However, we could design all visitors so that they return a class that inherits from a common base class. We could also use code generation as discussed above for Java or C#. But C++ has a strong template mechanism that could be used to instruct the compiler to automatically generate the correctly typed `accept()` method at compile time in some cases.

7.4.2 Handling Class Hierarchies

In this section, we describe a small modification to the visitor design pattern that substantially improves it when the elements visited form a multi-level class hierarchy. Essentially, it allows us to benefit from *method inheritance* (i.e., automatically using the method of (or for) the base class when it is not implemented for a specific class) when implementing an operation using a visitor. This possibility was lost when we switched from implementing an operation inside a class to the classic visitor design pattern.

The main change is in the `accept()` method of each kind of element. It tries to call

Table 7.9: Example of a visitor that returns a value directly.

```
class AbstractAddressEvaluator(ValueVisitor):
    def visitInteger(self, value):
        return value

    def visitAbstractAddress(self, aa):
        return aa

    def visitAbstractVariable(self, av):
        # If possible, we prefer returning an AbstractAddress
        ptr = self._analysis.currentState.pointer.getAddress(av)
        if isinstance(ptr, AbstractAddress):
            return ptr
        # Otherwise, maybe an integer
        integer = self._analysis.currentState.linear.getValueConst(av)
        if isinstance(integer, Integer):
            return integer
        # Otherwise the variable itself
        return av

    def visitPlus(self, value):
        val1 = value.val1.accept(self)
        val2 = value.val2.accept(self)
        if val1 is None or val2 is None:
            return None
        if isinstance(val1, AbstractAddress):
            return AbstractAddress(val1.zone, Plus(val1.offset, val2))
        if isinstance(val2, AbstractAddress):
            return AbstractAddress(val2.zone, Plus(val2.offset, val1))
        return Plus(val1, val2)

    def visitTimes(self, value):
        val1 = value.val1.accept(self)
        val2 = value.val2.accept(self)
        return Times(val1, val2)
```

Table 7.10: Automatic implementation of the improved visitor design pattern.

```
# This function prepares a class hierarchy for the improved
# visitor design pattern (for class hierarchies)
def _makeVisitable(classes, visitorName, visitableBase):
   for clsName in classes:
      visitName = 'visit' + clsName
      def accept(self, visitor, clsName=clsName, visitName=visitName):
         try:
            visitMethod = getattr(visitor, visitName)
         except AttributeError:
            return super(globals()[clsName], self).accept(visitor)
         return visitMethod(self)
      setattr(globals()[clsName], 'accept', accept)

   def visitBase(self, obj):
      raise RuntimeError("visit" + visitableBase
                         + " not implemented in " + str(type(self))
                         + " for " + str(type(obj)))
   globals()[visitorName] = type(visitorName, (object,),
                                 {'visit'+visitableBase: visitBase})
```

visit() for its own type, as usual, but if it is not implemented, it calls the accept() method of the base class, which implements the same algorithm. This implies that there is no more an interface that must be implemented completely by a visitor.

In RTL-Check, we use metaprogramming to generate accept() methods automatically. Table 7.10 shows how this is done. Notice that compared to the implementation of Table 7.8, this one creates only one visit() method in the base visitor class. It also puts this class directly in the global namespace (instead of returning it). We use this implementation for abstract variables and abstract variable constraints, which are both multi-level class hierarchies. For the Rtx class hierarchy, it is a modified _createRtxSubclass() (see Table 7.3) that does the work.

This improved visitor design pattern brings many advantages over classic visitors. Since it provides method inheritance, it is often possible to implement a visitor with less code. For example, the LivenessInterpreter collects all uses of and modifications to registers (gen and kill respectively). In the Rtx class hierarchy, RtxGenBinArith is

Table 7.11: Visitor that uses method inheritance (`LivenessInterpreter`).

```
class LivenessInterpreter(VisitorInterpreter):
    def visitRtxSet(self, rtx):
        if isinstance(rtx.dst, RtxReg):
            self.kill.add(Register(rtx.dst.num))
            rtx.src.accept(self)
        elif isinstance(rtx.dst, RtxMem):
            rtx.src.accept(self)
            rtx.dst.accept(self)

    def visitRtxGenBinArith(self, rtx):
        rtx.op1.accept(self)
        rtx.op2.accept(self)

    def visitRtxMem(self, rtx):
        rtx.addr.accept(self)

    def visitRtxReg(self, reg):
        self.gen.add(Register(reg.num))
```

the base class of all binary arithmetic operations (`RtxPlus`, `RtxMinus`, `RtxMult`, etc.). Table 7.11 shows a part the `LivenessInterpreter`. Notice that this visitor does exactly the same thing for each kind of binary operator.

Another more subtle advantage of the improved visitor is that sometimes, it allows us to add new elements that can be visited without having to change all the visitors. For example, the current `Rtx` class hierarchy does not yet include the `Rotate` binary operator. When we will add it, the `LivenessInterpreter` will not have to be modified.

Every visitor has a default `visit()` method that aborts with a significant error message when no other method can handle a certain type visited at run time. This is very useful to implement a visitor progressively on something like the `Rtx` class hierarchy, where there are over 150 different RTL codes but most programs use less than 50.

7.4.3 Our Implementation of the Visitor Design Pattern Compared to Others

There have been many proposals to automate the implementation of the visitor design pattern and to improve it. This section covers the different approaches that try to solve the problems of the classic visitor pattern affecting RTL-Check. We discuss the main differences between them and our class hierarchy variation on the visitor design pattern.

The variation that most resembles ours is the *default visitor* presented in [NI96]. This paper does not discuss automatic implementation of the pattern, but it explains how to handle class hierarchies. The idea is that every visitor on some class hierarchy inherits from a default visitor. This visitor has one `visit()` method for each kind of class that can be visited and each of their base classes. These methods just call the `visit()` method for their base class. In our approach, it is the `accept()` method that makes sure the call goes through the class hierarchy. The result is the same in both cases, i.e., it allows using method inheritance when implementing a visitor.

The paper about the default visitor indicates that it requires more work to implement than the classic visitor. This is because this variation requires one more class that contains all the default methods. This comment does not apply to our approach, since we use automatic code generation to implement the pattern.

More variations on the visitor design pattern are presented in [GH98], which describes SableCC, a framework for developing compilers. The first variation tries to make it easier to add new classes that can be visited. When a new class is created, instead of adding its `visit()` method to the existing visitor interface, which would force every existing visitors to implement this method, a new visitor interface is created. This way, existing visitors which do not have to work on this new class do not have to be modified. With our approach, no modification is required if existing visitors already implement a `visit()` method for an ancestor of the new class.

SableCC also uses something called *analysis adapters*. They are classes that implement the visitor interface and in which every `visit()` method calls a common method named `defaultCase()`. This method is equivalent to the `visit()` method for the base class of the hierarchy in our approach. Thus, visitors which inherit from an analysis adapter do not have to implement every method of the interface.

Another very interesting concept found in SableCC is that of *depth-first adapters*. They are also classes implementing the visitor interface. In these adapters, the `visit()`

method for a given class calls `accept()` on each visitable member of this class. This way, a complex object structure can be traversed automatically if it does not contain cycles. Some of our visitors would benefit from a depth-first adapter. For example, `LivenessInterpreter` is only interested in registers. In its current implementation, most of the code of this visitor is for traversing the object structure.

It would not be trivial to generate depth-first adapters automatically for an arbitrary object structure. SableCC generates these adapters automatically, but it also generates all the classes that are visited. This ensures that it has enough information about these classes to implement depth-first adapters automatically. However, in many cases, we could probably implement this adapter automatically with the use of reflection. In the case of RTL-Check, this would certainly be possible, but we did not try it.

There are a few variations of the visitor design pattern that remove the need for `accept()` methods. This has the advantage of not requiring any modification to the elements that must be visited; the visitors are completely encapsulated. The *extrinsic visitor* is described in [NI96]. A new `dispatch()` method is added to each visitor. This method must be implemented manually and it uses run-time type information to decide the right `visit()` method to call. It replaces all the `accept()` methods.

Another approach to eliminate `accept()` methods is the `Walkabout` class described in [PJ98]. This class implements a default `visit()` method which uses run-time type information and reflection to automatically dispatch the call to the `visit()` method for the right class. Visitors that inherit from this class do not require any `accept()` method to work properly. In RTL-Check, we did not try to remove these methods. They are not an issue because we did not have to modify the source code of the classes that are visited to accommodate the visitor design pattern; we generate them dynamically.

Another very interesting implementation of the visitor design pattern is presented in [TC98]. It uses the *metaobject protocol* of OpenJava, which can be used to implement some kind of metaclasses. The approach described supports method inheritance and it saves the programmer from writing most of the code required for the visitor design pattern. However, all classes and interfaces that participate in the pattern must have some annotation, including the elements that must be visited. This is more work than the simple call to `_makeVisitable()` in RTL-Check.

Chapter 8

Open Source Development

In Section 3.1 we gave our reasons for making RTL-Check open-source software. In this chapter, we go deeper and discuss everything related to the open-source nature of RTL-Check. Section 8.1 explains the bazaar development process and how we tried to apply it to RTL-Check. Section 8.2 presents the results of this open source experiment and Section 8.3 discusses possible explanations for why no bazaar emerged around our project.

8.1 Bazaar Development

Our decision to create an open-source project was much inspired by the success of Linux, which is analyzed in *The Cathedral and the Bazaar* [Ray00a]. This essay was sparked by the shocking revelation that

> a world-class operating system could coalesce as if by magic out of part-time hacking by several thousand developers scattered all over the planet.

At the time Linux was started, everybody thought such complex software could only be built like a cathedral, with careful a priori planning and conventionally-managed workers. Instead,

> the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches out of which a coherent and stable system could seemingly emerge only by a succession of miracles.

Eric S. Raymond tried to understand how and why the Linux development process, which he calls bazaar-style, is so effective. He tested his theory and he proposes nineteen aphorisms about how to run an open-source project effectively. We decided to use them as guidelines for the development of RTL-Check. Here we present the ones we think are the most relevant to our effort.

2. Good programmers know what to write. Great ones know what to rewrite (and reuse).

We did not try to implement anything to parse and deal with the complexity of C and C++. Instead, we relied on GCC, a compiler that already knows about all this. However, for the core of our analysis, we could not find something good enough as a starting point to reach our goals, so we developed it from scratch.

3. "Plan to throw one away; you will, anyhow." (Fred Brooks, *The Mythical Man-Month*, Chapter 11)

As we briefly told at the beginning of Chapter 5, after our proof of concept our first real attempt at implementing an analysis to find memory access errors did not give the expected results. Its code, which was developed mostly from versions 0.0.6 to 0.0.8 of RTL-Check, was not modular enough and it became more and more difficult to improve. We did not hesitate to throw away much of this code, i.e., almost six months of work.

This is probably the single most important decision that we took. We did not see our effort as a failure or lost time; it was a necessary part of a process to understand the full extent of the problem we are dealing with. The lessons we learned from this attempt helped us to establish the foundations of the framework that is now the core of RTL-Check and which allowed us to implement a much better and much more modular analysis.

7. Release early. Release often. And listen to your customers.

Our first release was still at the stage of proof of concept, so we did release early. Whether we did release often is questionable. For the fifteen months during which the main development of RTL-Check took place, we made thirteen releases. At the beginning, we did four releases in less than forty days, but afterward, the typical release cycle was about two months. Since we were, most of the time, alone working on RTL-Check, we felt it was more important to deliver a new version only when there were user-visible improvements rather than half-finished code that cannot run.

Concerning the customers, if we consider they are the users, very few revealed themselves and they did not make huge requests, so listening was not a big issue.

6. Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.

11. The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is better.

19. Provided the development coordinator has a communications medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.

Since one of our goals was to attract competent people in our project to achieve better results faster, these guidelines were very important to us. The problem is that very few of our users gave us feedback in spite of our request in each announcement and, up to now, only one other person contributed bugfixes and new code to our project. Still, we did not hesitate to apply these guidelines when it was possible.

8.2 Results of Our Open Source Experiment

Here we present the results of our open source experiment in relation to the goals we introduced in Section 3.1.

8.2.1 Attracting Attention

Our first goal was showing to the world the kind of research we are doing. The anonymity of the Internet makes it difficult to know precisely who paid attention to our project and why, but we have statistics to show that many people did.

We used two well-known open source related sites to help the promotion of RTL-Check. The first one is `http://sourceforge.net`. This is the site that officially hosts

Table 8.1: Statistics about RTL-Check as of January 4th 2006.

RTL-Check home page hits[a]	2388
Total number of web hits[b]	11267
Total number of downloads	609
Number of releases	14
Average number of downloads per release	43.5
Freshmeat.net record hits	8096
Freshmeat.net URL hits	1982
Current number of Freshmeat.net subscribers	30
Current number of mailing list subscribers	7

[a]Counted as the number of requests for the SourceForge.net logo.

[b]Includes all hits to the RTL-Check web site (6641) and to the SourceForge.net project page (4626).

our project. It provides web space, download mirror sites around the world, a mailing list and many other related services.

The second site, `http://freshmeat.net`, allows project administrators to register their projects and announce new releases. Its front page always shows the projects with most recent releases and its users can also subscribe to a given project so that they get informed when there is a new release.

Table 8.1 shows some statistics we have about RTL-Check. They cover the period from the start of the project in May 2004 to January 4th 2006, i.e., about 20 months. An interesting fact is that more than 80% of the hits to the project home page came from Freshmeat.net (URL hits). This seems to indicate that Freshmeat.net is an effective way to promote RTL-Check.

Figure 8.1 shows the number of download for each version of RTL-Check and their date of release. The number of downloads varies greatly from version to version and it is difficult to explain why. It may be related to the actual text of the announcement, e.g. the list of new features, the amount of time between releases, the period of the year, the day of the week or the number of other releases announced on Freshmeat.net on the day of our announcement. It is most likely a combination of many factors. It is interesting to note that version 0.0.5 is the one with the fewest downloads; it is the only one that was not announced on the Freshmeat.net front page.

Figure 8.2 shows a detailed view of the number of downloads during summer 2005.

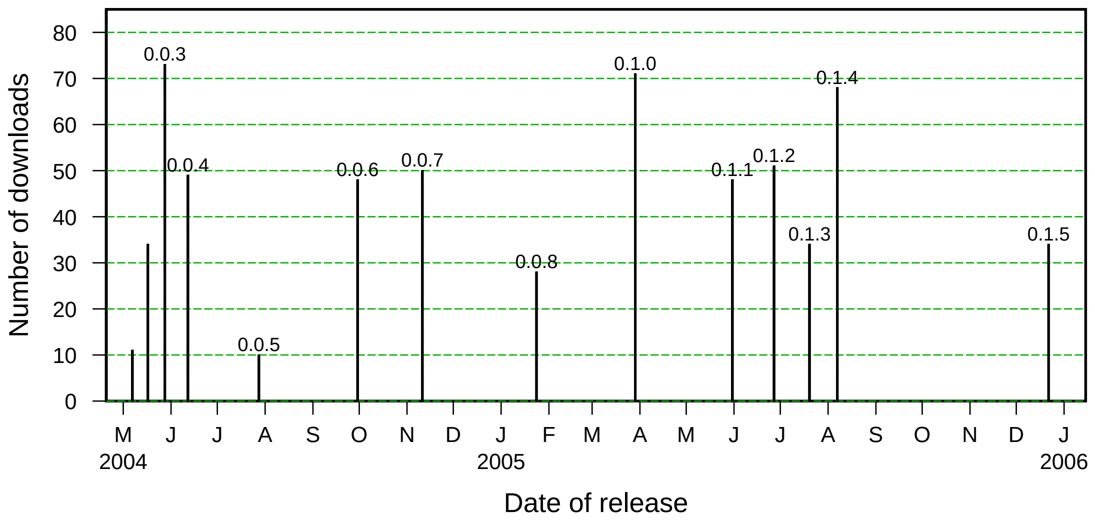

Figure 8.1: Number of downloads of RTL-Check per version.

From this figure, it is clear that most downloads are made during the very first few days after the release. What we do not know is whether our user base is mostly stable or if it is renewed from release to release. The number of subscribers on Freshmeat.net and our mailing list seems to indicate that at least a part of our user base is stable.

Every release of RTL-Check was announced on the project mailing list, but it was mostly one-way communications. However, we were contacted personally by three persons who were all students at different universities working on projects involving static analysis. One of them gave us some hints about GCC; he has also implemented static analysis of RTL.

We also announced the project once on the GCC mailing list. We did that on the day of the second release of RTL-Check. We never received direct feedback from this announcement. The number of downloads increased for this version and the following one, but it is difficult to tell whether it is related.

8.2.2 Attracting Co-Developer

Our second goal was attracting competent people in the project so that we can achieve better results faster. Up to now, only one person has contributed to RTL-Check, Moritz Muehlenhoff. He is a computer science student (Diplomstudiengang Informatik) at University of Bremen in Germany. He chose to work on RTL-Check for his diploma

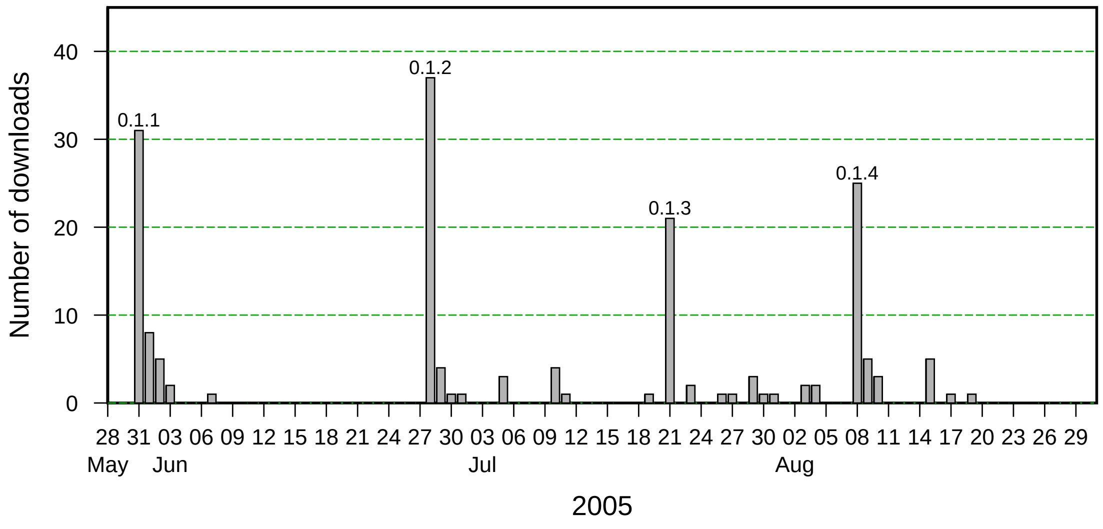

Figure 8.2: Detailed view of downloads during summer 2005.

thesis (Diplomarbeit) because it uses an intermediate compiler representation, it is open-source, and we were helpful when he needed information about how RTL-Check works [Mue06].

He has already contributed code for automatically computing the size of parameters passed to a function instead of having to specify it manually. He plans to do the same with global variables, to add integer overflow and infinite loop detection, to create a shell that helps debugging analyses, to port the modifications of GCC to a newer version, to extract information from debugging symbols and more.

8.2.3 Helping Research

Our third goal was helping research on static analysis, but it is too soon to judge the impact that RTL-Check will have in this area. What is certain is that without RTL-Check, Moritz Muehlenhoff would have to do much more work to achieve the same results. He would have started a new project from scratch because he did not find any other tool that suited his needs [Mue06].

8.3 Why No Bazaar Emerged

RTL-Check has not reached the point where it is developed in a true bazaar style; we did not get enough feedback from our users and only one other person has contributed to our project. We know that RTL-Check has been downloaded many times, but we do not know by whom and what they did with it.

Maybe our expectations were too high. Eric S. Raymond [Ray00a] thinks that

> it would be very hard to *originate* a project in bazaar mode. [...] Your nascent developer community needs to have something runnable and testable to play with.
>
> When you start community-building, what you need to be able to present is a *plausible promise*. [...] What [your program] must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future.

RTL-Check certainly did run on its first release, but it did not do much. Maybe some people just wait for the project to become more mature before creating a real bazaar around it. It still lacks some important features such as interprocedural analysis, which might convince more people of the usefulness of our project. One advantage RTL-Check will have when it will get more users is that, because of its nature, its users are developers, an ingredient of paramount importance for an effective bazaar.

In another related essay [Ray00b], Eric S. Raymond gives another clue that might partly explain why few developers took part in the project.

> Thus, there's an optimum distance from one's neighbors (the most similar competing projects). Too close and one's product will be a “me, too!” of limited value, a poor gift (one would be better off contributing to an existing project). Too far away, and nobody will be able to use, understand, or perceive the relevance of one's effort (again, a poor gift).

Maybe RTL-Check is “too far ahead” among open-source projects. One needs a certain amount of theoretical knowledge to fully understand it. The proportion of developers who know about formal methods and there applications is very low. This is a disadvantage RTL-Check will have for a foreseeable future compared to many other

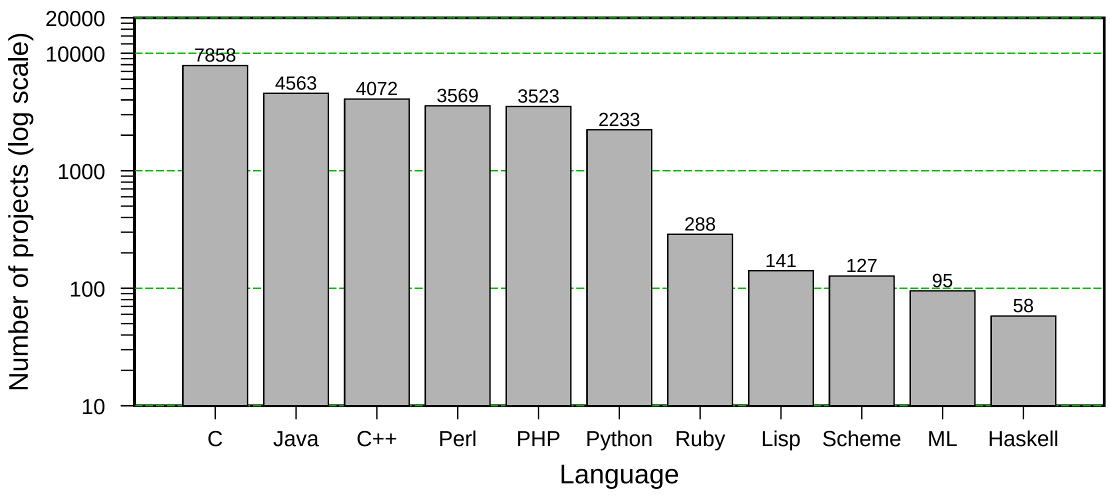

Figure 8.3: Number of projects on Freshmeat.net for selected languages.

open-source projects; there are more people who need an operating system than there are who need to analyze source code.

One can wonder whether Python was the best language to attract developers or whether it is too marginal. Figure 8.3 shows some selected programming languages and the number of projects on Freshmeat.net using them. Python does not seem to be that marginal. On the left side of Python are all the languages that are used for more projects than Python. We did not want to use any of them because of their relatively poor expressivity. The languages on the right may be expressive enough, but they have a user base much smaller than that of Python.

Another aspect that might help explaining why we did not get as many developers as expected is promotion. Announcing our project more often on other mailing lists and web sites could increase the probability of being notified by people willing to contribute. Doing releases more often might also be effective because, in addition to increasing the number of announcements, it also makes the project appear more alive.

Still, we think that one day, when RTL-Check will be a more mature project, a real bazaar might emerge around it. It could become a widely adopted platform for doing research on static analysis and program verification.

Chapter 9

Conclusion and Future Work

In this thesis, we have presented the problem of memory safety in languages such as C and C++ and we have explained its importance in our society where computers and programs are ubiquitous. We have developed a static analysis which aims to prove that a given program is memory-safe. This analysis is not yet complete, but it is extensible and anyone can improve it since it is distributed in RTL-Check, which is open-source software.

We have also created a static analysis framework that allows building analyses from small reusable components. We have demonstrated the effectiveness of this framework by implementing a memory access analysis using it. This framework is not limited to the analysis described in this thesis; it is more general and it could be employed to analyze other, possibly more complex, properties of programs. For this purpose, existing modules from our memory safety analysis could be put to contribution.

Because of its open-source nature, we foresee many improvements to our work. Concerning the framework itself, there could be additions to the list of known RTL codes. Besides, it might be a good idea to support newer versions of GCC and the analysis of GIMPLE, its new intermediate representation which would make it easier to obtain information about program variables.

Also, it would be interesting to see how the framework could help abstracting differences between computer architectures. Up to now, it has been tested only for processors of the Intel 80386 family. Likewise, only C programs were analyzed up to now. We are confident that our framework is fine for other languages, but still, this must be tried out. C++ and Java are two popular languages and many programs written in them could certainly benefit from static analysis.

Much theoretical work has to be performed before we know with certainty if there remain unknown problems in the different parts of our memory safety analysis. The semantics of RTL, that of our concrete and abstract variable expression languages, the solvers and the invariants of the policies must first be formalized. Only then will it be possible to show that an analysis and its interpreters are correct with respect to a given program property.

Concerning existing solvers, many of them could be improved. For example, the linear constraint solver should have a complete implementation of widening and narrowing. It could also be factorized in two or more solvers to make it easier to understand. The modulo solver should have narrowing and the pointer solver should have a complete implementation of `join()` and `isMoreInformative()`. A useful addition to this solver would be knowledge about possibly null pointers.

Moreover, new solvers could be created to improve the precision of the current analysis. For example, a solver could keep track of null-terminated strings. Another one could be specialized in understanding a given complex data structure. A solver to detect integer overflows would also be important since it is a known weakness of the memory safety analysis.

In order to be able to analyze complete programs, it will be crucial to support function calls and render the analysis interprocedural. Other possible areas of improvement include more parsimonious use of widening and narrowing in the memory safety analysis algorithm, optimizations to the existing modules and a shell to run analyses and help debugging them.

In conclusion, we have created a static analysis framework that is strong enough to support the analysis of a property as complex as memory safety in a language such as C. Much work can be done to improve the memory safety analysis, but the open-source framework and the existing modules distributed in RTL-Check already represent a significant contribution to the field of static program analysis. We hope that our work will help sparking new ideas among researchers around the world and that it will facilitate trying them out.

Bibliography

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 290–301. ACM Press, 1994.

[AIGBRMHT05] Juan-José Amor-Iglesias, Jesús M. González-Barahona, Gregorio Robles-Martínez, and Israel Herráiz-Tabernero. Measuring libre software using debian 3.1 (sarge) as a case study: Preliminary results. *Upgrade*, VI(3):13–16, June 2005. Available from World Wide Web: `http://www.upgrade-cepis.org/`.

[Aik94] Alexander Aiken. Set constraints: Results, applications, and future directions. In *PPCP '94: Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, pages 326–335. Springer-Verlag, 1994.

[Ale96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49):14, November 1996. Available from World Wide Web: `http://www.phrack.org/phrack/49/P49-14`.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM Press, 1993.

[Bal04] Thomas Ball. Formalizing counterexample-driven refinement with weakest preconditions. Technical Report MSR-TR-2004-134, Microsoft Research, December 2004.

[Ban] Banshee home page. `http://banshee.sourceforge.net/`.

[BCDR04] Thomas Ball, Byron Cooky, Satyaki Das, and Sriram K. Rajamaniy. Refining approximations in software predicate abstraction. In *Proc. of Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, 2004.

[CBJW03] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium.* USENIX Association, August 2003.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92, Leuven, Belgium*, pages 13–17. Springer-Verlag, Berlin, Germany, 1992.

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181. ACM Press, New York, NY, June 1995.

[CCF$^+$05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, 2005.

[CFR$^+$91] 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.

[CH01] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *21st International Conference on Distributed Computing*, page 409. IEEE Computer Society, April 2001.

[Cou97] Patrick Cousot. Types as abstract interpretations. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331. ACM Press, 1997.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78. USENIX Association, January 1998.

[CQu] CQual home page. `http://www.cs.umd.edu/~jfoster/cqual/`.

[dR03] Theo de Raadt. i386 W^X. Posted on OpenBSD mailing list, Apr., 2003.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167. ACM Press, 2003.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan/Feb 2002.

[Ele] Electric Fence home page. `http://perens.com/FreeSoftware/`.

[EY00] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks, June 2000. Available from World Wide Web: `http://www.trl.ibm.com/projects/security/ssp/main.html`.

[Fla] FlawFinder home page. `http://www.dwheeler.com/flawfinder/`.

[Fos02] Jeffrey Scott Foster. *Type qualifiers: lightweight specifications to improve software quality.* PhD thesis, University of California, Berkeley, 2002.

[FS01] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *10th USENIX Security Symposium*, pages 55–66. USENIX Association, August 2001.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2002.

[GCC03] Gcc internals, 2003. Available from World Wide Web: `http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gccint/`.

[GH98] Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 140. IEEE Computer Society, 1998.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

[GJC$^+$03] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 345–354. ACM Press, 2003.

[HB03] Eric Haugh and Matthew Bishop. Testing C programs for buffer overflow vulnerabilities. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*. Internet Society, February 2003.

[HJ91a] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–136. USENIX Association, 1991.

[HJ91b] Nevin Heintze and Joxan Jaffar. Set-based program analysis (extended abstract), 1991. Available from World Wide Web: `http://www.cs.cmu.edu/afs/cs/user/nch/ftp/sba.ps.Z`.

[HJ94] Nevin Heintze and Joxan Jaffar. Set constraints and set-based analysis. In *PPCP '94: Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, pages 281–298. Springer-Verlag, 1994.

[HMCCR94] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and René G. Rodríguez. FIAT: A framework for interprocedural analysis and transfomation. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 522–545. Springer-Verlag, 1994.

[JK97] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs, 1997. Department of Computing Imperial College of Science, Technology and Medicine. http://www.doc.ic.ac.uk/~phjk/Publications/.

[Jon95] Richard W. M. Jones. A bounds checking C compiler, May 1995. Department of Computing Imperial College of Science, Technology and Medicine.

[KA05] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS '05: Proceedings of the 12th International Static Analysis Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 218–234. Springer-Verlag, September 2005.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206. USENIX Association, 2002.

[Ket03] Richard Kettlewell. Protecting against some buffer-overrun attacks, April 2003. Available from World Wide Web: http://www.greenend.org.uk/rjk/random-stack.html.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. In *Acta Informatica (Historical Archive)*, volume 7, pages 305–317. Springer-Verlag GmbH, September 1977.

[Lac03] Patrice Lacroix. Les débordements de tampons et les vulnérabilités de chaîne de format. Technical Report DIUL-RR-0304, Département d'informatique, Université Laval, August 2003.

[LC02] Kyung-suk Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–88. USENIX Association, 2002.

[LD05] Patrice Lacroix and Jules Desharnais. Buffer overflow vulnerabilities in C and C++, 2005. Submited to Computing Surveys.

[LE01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*. USENIX Association, August 2001.

[LL03] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 317–326. ACM Press, 2003.

[Mau04] Laurent Mauborgne. ASTRÉE: Verification of absence of run-time error. In *Building the Information Society (18th IFIP World Computer Congress)*, pages 384–392. The International Federation for Information Processing, Kluwer Academic Publishers, Aug 2004.

[MCGW02] J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

[MdR99] Todd C. Miller and Theo de Raadt. Strlcpy and strlcat — consistent, safe, string copy and concatenation. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*. USENIX Association, June 1999.

[Mol03] Ingo Molnar. Exec shield, new Linux security feature. Posted on linux-kernel mailing list, May, 2003.

[Mue06] Moritz Muehlenhoff. Personal communications, January 2006.

[NI96] Martin E. Nordberg III. Variations on the visitor pattern. In *PLoP '96 Writers Workshops*, 1996. Available from World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/workshops.html`.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139. ACM Press, 2002.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[PaX00] PaX Team. Original design & implementation of PAGEEXEC, November 2000.

[PaX02] PaX Team. Why a stack with exec flag? Posted on comp.os.linux.security group, June, 2002.

[Per98] Bruce Perens. The open source definition, 1998. Available from World Wide Web: `http://www.opensource.org/docs/definition.php`.

[PJ98] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, pages 9–15. IEEE Computer Society, 1998.

[PL02] Changwoo Pyo and Gyungho Lee. Encoding function pointers and memory arrangement checking against buffer overflow attack. In *ICICS '02: Proceedings of the 4th International Conference on Information and Communications Security*, pages 25–36. Springer-Verlag, 2002.

[Pot00] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[Pro03] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, August 2003.

[Ray00a] Eric S. Raymond. The cathedral and the bazaar, 2000. Available from World Wide Web: `http://www.catb.org/~esr/writings/cathedral-bazaar/`.

[Ray00b] Eric S. Raymond. Homesteading the noosphere, 2000. Available from World Wide Web: `http://www.catb.org/~esr/writings/homesteading/`.

[SBDB01] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy*, page 144. IEEE Computer Society, 2001.

[Sol97a] Solar Designer. Getting around non-executable stack (and fix). Posted on BugTraq mailing list, Aug., 1997.

[Sol97b] Solar Designer. Non-executable stack – final Linux kernel patch. Posted on linux-kernel mailing list, May, 1997.

[Sot05] Alexander Ivanov Sotirov. Automatic vulnerability detection using static source code analysis. Master's thesis, University of Alabama, 2005.

[SP05] Vincent Simonet and François Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, January 2005.

[Sta] Stack Shield home page. http://www.angelfire.com/sk/stackshield/.

[Sta01] Paul Starzetz. Announcing RSX - non exec stack/heap module. Posted on BugTraq mailing list, June, 2001.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*. USENIX Association, 2001.

[SU99] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the 8th USENIX Security Symposium*, pages 63–78. USENIX Association, 1999.

[SWM00] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381. Springer-Verlag, 2000.

[TC98] Michiaki Tatsubori and Shigeru Chiba. Programming support of design patterns with compile-time reflection. In *OOPSLA 98 Workshop on Reflective Programming in C++ and Java*, 1998. Available from World Wide Web: http://www.csg.is.titech.ac.jp/~chiba/oopsla98/proc/mich.pdf.

[TK02] Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advances in Intrusion Detection: 5th International Symposium*, pages 274–291. Springer-Verlag, October 2002.

[TS01] Timothy Tsai and Navjot Singh. Libsafe: Protecting critical elements of stacks. Technical Report ALR-2001-019, Avaya Labs, Avaya Inc., 233 Mt. Airy Rd., NJ 07920 USA, August 2001.

[VBKM00] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 257. IEEE Computer Society, 2000.

[Wag00] David A. Wagner. *Static analysis and computer security: New techniques for software assurance*. PhD thesis, University of California at Berkeley, 2000.

[Wei03] Georg Weissenbacher. An abstraction/refinement scheme for model checking C. Diplomarbeit in telematik, Institut fur Softwaretechnologie der Technischen Universitat Graz, 2003.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, pages 3–17. Internet Society, February 2000.

[Wik] Wikipedia: Metaprogramming. Available from World Wide Web: `http://en.wikipedia.org/wiki/Metaprogramming`.

[Woj01] Rafal Wojtczuk. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58):0x04, December 2001. Available from World Wide Web: `http://www.phrack.com/phrack/58/p58-0x04`.

[XCE03] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 327–336. ACM Press, 2003.

[XDS04] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *ACM SIGSOFT Software Engineering Notes*, 29(6):117–126, 2004.

[YH03] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 307–316. ACM Press, 2003.

[ZLKR04] K. Zee, P. Lam, V. Kuncak, and M. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, November 2004. Available from World Wide Web: http://www.cag.lcs.mit.edu/~rinard/paper/.