90% Perspiration: Engineering Static Analysis Techniques for Industrial Applications

Paul Anderson
GrammaTech, Inc.
317 N. Aurora St.
Ithaca, NY 14850
paul@grammatech.com

Abstract

This article describes some of the engineering approaches that were taken during the development of GrammaTech's static-analysis technology that have taken it from a prototype system with poor performance and scalability and with very limited applicability, to a much-more general-purpose industrial-strength analysis infrastructure capable of operating on millions of lines of code. A wide variety of code bases are found in industry, and many extremes of usage exist, from code size through use of unusual, or non-standard features and dialects. Some of the problems associated with handling these code-bases are described, and the solutions that were used to address them, including some that were ultimately unsuccessful, are discussed.

1. Introduction

GrammaTech has been in the software tools business for almost twenty years. In 1999 *CodeSurfer* was introduced, originally conceived as a slicing tool for ANSI C programs. This was based on a prototype developed as a research vehicle at the University of Wisconsin, Madison. At that time, the limit of its applicability was approximately 30,000 lines of ANSI C code.

Since that time, CodeSurfer has evolved into a general-purpose static analysis platform for several languages, now capable of analyzing tens of millions of lines of code, and used directly and indirectly by many software development organizations. This paper describes the path that this evolution took, including time that was wasted on dead ends.

The remainder of this paper is structured as follows. Section 2 describes CodeSurfer as it exists today, including a brief description of *CodeSonar*, a tool that uses the CodeSurfer infrastructure to find programming errors in source code. Section 3 discusses how CodeSurfer was engineered to allow it to operate on very large programs. Section 4 describes key decisions that were taken to expand the reach of the tool beyond its original conception. Section 5 discusses generalizations that were made to support multiple languages. Section 6 describes the compromises to principles that had to be made to be successful. Section 7 briefly describes anticipated improvements and new directions. Finally, Section 8 presents some conclusions.

2. CodeSurfer

CodeSurfer is a whole-program static analysis infrastructure. Given the source code of a program, it analyzes the program and produces a set of representations, on which static analysis applications can operate.

These intermediate representations (IRs) comprise the following:

- Low level lexical information about tokens and their locations in the source code files.
- Information about how the preprocessor was used to transform the input to the parser.
- The abstract syntax tree (AST), including the symbol table, as generated by the parser for each compilation unit.
- The control-flow graph (CFG), where each node represents a program point such as an expression, a parameter, a call site, etc. An option for static single assignment form is given.
- Each CFG node is associated with a normalized AST fragment representing the computation at that point.
- The whole-program call graph.
- A whole-program points-to graph.

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.11

IEEE computer society

- Variable definition and use information for every program point.
- The set of non-local variables used and potentially modified by each procedure (GMOD).
- The system dependence graph (SDG), with both data-dependence and control-dependence edges, as described in [5].
- Summary edges, which capture the transitive dependences of a call to a procedure at a call site [6].

CodeSurfer has a graphical user interface that allows a user to navigate and query the program in terms of these representations. A user can do forward and backward slicing, or perform regular or truncated chops. Predecessor and successor operations yield immediate neighbors in the dependence graph. A set calculator can be used to combine sets of program points using set-theoretic operations.

Figure 1 shows a simplification of the dependence graph for a small program comprising two procedures. Only data and control-dependence edges are shown.

An API provided in both Scheme and C provides access to all of these representations.

Figure 2 shows a schematic of the architecture of CodeSurfer. This illustrates how the IR can be generated from many different source languages, and how various applications can be built on the API.

2.1. Path Inspector

The *Path Inspector* is a CodeSurfer add-on that allows users to reason about paths through the code. It uses model-checking techniques, where the model is the interprocedural control-flow graph. Users specify properties that should hold, and the model checker attempts to prove that they do. If a property does not hold, then the model checker delivers a counter-example where the property is violated.

The properties are formulae in a temporal logic, but users specify them as state machines, where the states are sets of program points and transitions imply paths.

To the user, the queries look like templates over paths. For example, a query might be *"There is no path from A to C that does not go through B"*, where *A*, *B* and *C* are arbitrary sets of program points.

State machines can be specified with quantifications over objects too. Thus a user can specify paths for particular objects. This is useful for asking about the state of a particular variable. For example, "*There is no path to points where F is written to after F is closed*".

The path inspector uses weighted pushdown systems as its underlying formalism [8]. As such, it was a more sound model than is used in CodeSonar. Section 6.3 discusses why CodeSonar uses different techniques.

2.2. CodeSonar

CodeSonar is the largest and most sophisticated application built using the CodeSurfer infrastructure. It is designed to find programming flaws in software. It uses symbolic execution techniques to perform a path-sensitive analysis of the subject software. It works bottom up on a version of the call graph where cycles have been removed. Each procedure is analyzed by exploring paths through the CFG while maintaining information about the values of variables and how they relate to each other. As anomalies that indicate potential flaws are encountered, a report is issued. When the analysis of a procedure is completed, summary information is generated and used in the analysis of the clients of that procedure.

CodeSonar was designed to be fast and scalable. As such it could not rely on many of the more sophisticated representations generated by CodeSurfer. As a bug finder (as opposed to a program verifier), the analysis can afford to be both unsound and incomplete.

Many of the changes to the CodeSurfer infrastructure were driven by requirements imposed by the need to make CodeSonar successful.

CodeSonar uses a surprisingly modest subset of the IR that CodeSurfer generates: the call graph, the CFGs, and the normalized ASTs are used. The unnormalized ASTs are discarded, pointer analysis is not used, and neither control or data dependence edges, including summary edges, are generated.

3. Engineering for Scalability

This section describes how CodeSurfer was engineered to improve its scalability.

3.1. Stratification

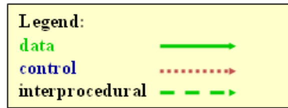
Not all clients of CodeSurfer need access to all of the intermediate representations that it generates. For example, summary edges are unnecessary if the user has no interest in slicing or chopping in linear time. As these are among the most expensive representations to compute and store, it is useful to be able to turn their generation off.

Most of the intermediate representations are optional. Those that are particularly expensive in terms of time and space are the following.

- **Unnormalized Abstract Syntax Trees** in native forms can be extremely expensive in terms of space. One reason for this is there is usually a high degree of redundancy between compilation units. Most front ends are not designed to store this information efficiently for whole programs.

```
void main()
{
 int sum, i;
 sum = 0;
 i = 1;
 while (i<11) {
   sum = add(sum,i);
   i = add(i, 1);
  }
 printf("sum=%d\n", sum);
 printf("i=%d\n", i);
}

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



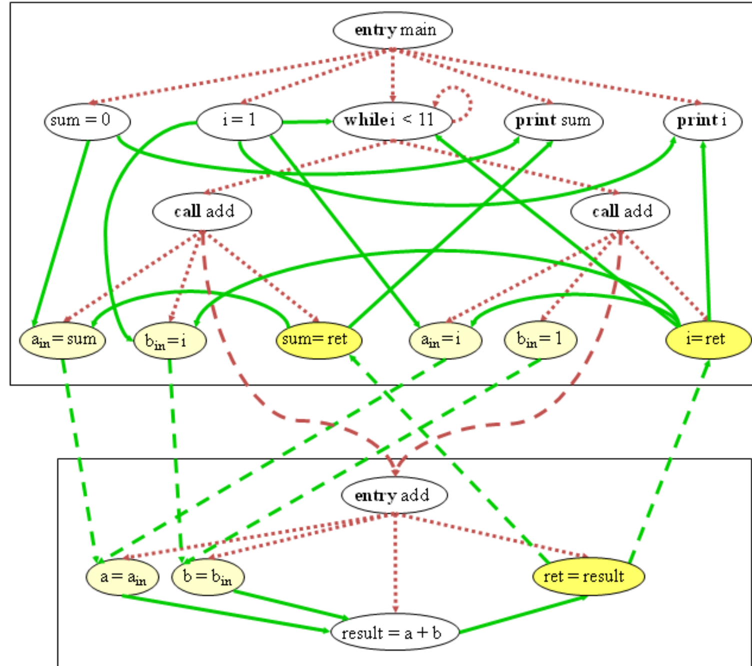


Figure 1. A simplification of the SDG created by CodeSurfer for a small program.

- **Pointer Analysis** is discussed further in Section 6.2 below. It can be turned off entirely.
- **GMOD** is only usually useful if a pointer analysis phase is used to compute aliases. With such aliases it is very time-expensive.
- **Summary Edges** require time $O(n^3)$ in the number of parameters to compute. Although aggressive factoring is used to reduce their space consumption, the cost in both time and space is very high.

These may all be suppressed or scaled back when building the program model.

3.2. Memory-mapped Input/Output

CodeSurfer operates by creating the IR and storing it to disk such that a subsequent process can read it in and perform an analysis. Originally, CodeSurfer did this by storing the information in a set of files in an ASCII format. Each analysis process was required to parse these files to recreate an in-memory representation of the IR. This was hopelessly slow for all but the most trivial of applications. On-demand reconstitution of the IR was not a significant improvement.

The first solution to this was to use memory-mapped IO. Instead of allocating the in-memory objects on the heap using the standard *malloc* and *free*, a new memory allocation library was used that provided a similar interface, but where instead of using *sbrk* to request more memory from the operating system, it used *mmap* instead. The effect of this was that the internal representation of the IR was the same as the external representation. The virtual memory layer of the operating system took care of paging the contents in and out of main memory. This approach resulted in the IO cost being reduced to negligible quantities.

There were a few disadvantages to this approach, and it has since been discarded. One was that the file had to be mapped at the same virtual address every time it was read in. Also, the address space used was required to be contiguous. This meant that the choice of the base address of the file was very important. It had to be chosen to leave enough space to create very large structures, and also to not clash with other addresses in use, including the normal heap.

This was successful enough that it was the main IO for-

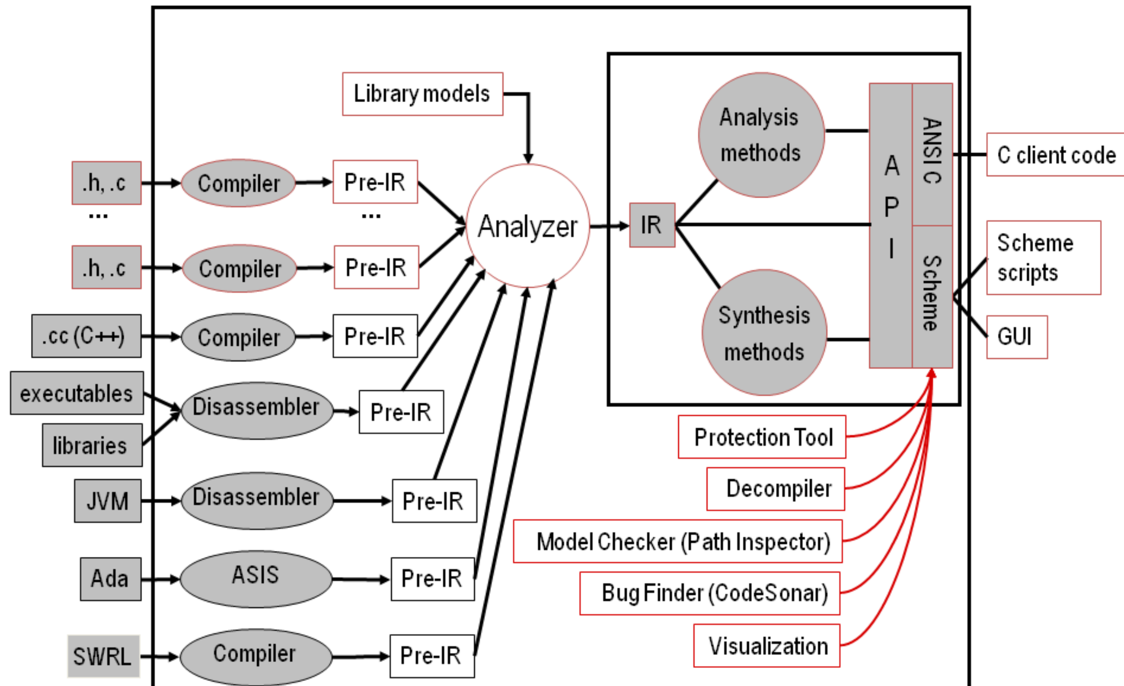


Figure 2. The architecture of CodeSurfer.

mat used by CodeSurfer for many years. However, it was eventually discarded because of a fundamental limitation: address space. On most machines with a 32-bit address space, the maximum amount of addressible memory is 2Gb, and in practice because of the need to avoid clashes, and the requirement for contiguity, the actual amount available is much less. This imposed a hard limit on the amount of space available for CodeSurfer IR storage, which was unacceptable.

3.3. Object Store

The storage mechanism that replaced the memory-mapped IO was designed and implemented entirely in house. Several third-party options were evaluated, but none satisfied all our requirements.

This object store allows a user to create fine-grained objects in memory, and then have these automatically paged in and out of memory to backing store as necessary. Object-store "addresses" are a full 64 bits, so address space is no longer an issue. Multiple object stores can be used simultaneously, so embedded in each of these is an identifier that specifies the specific store it refers to.

Cache analysis tools were built and used to analyze and optimize the object store for its use in CodeSonar.

4. Engineering for Usability

This section describes strategies taken to improve the usability and general applicability of the infrastructure.

4.1. C/C++ Language Usage

There are a large number of compilers in use in industry, especially for embedded systems development, and they all behave differently in key ways:

- The location of default or system header files.
- The set of preprocessor or other symbols that are implicitly defined.
- The set of command-line arguments accepted and how they are interpreted.

- The way in which they mangle exported names.
- The definition of the language they are prepared to accept as input.

The last point is particularly important. No C compiler is purely ANSI C, and even if the program being compiled is pure ANSI C, the standard header files never are. Most compilers implement non-standard extensions. New keywords are most common. This is especially true for compilers for embedded systems, as vendors like to provide linguistic means for accessing low-level hardware features.

Some examples of this are the following:

- There are many different syntaxes for specifying inline assembly.
- The Keil compiler allows the declaration of a variable such as _x, and occurrences of `x` are then considered uses of that variable.
- A large network hardware company uses a custom version of *gcc* based on a long-obsolete public version. This version accepts string literals that span multiple lines without closing quotes.
- The Microsoft C/C++ compiler implements a `#import` preprocessor directive. This incorporates symbolic information from a proprietary type library.
- Although non-standard keywords can often be ignored by using preprocessor definitions, some compiler vendors have insisted on introducing new syntax that cannot be handled in this way. The IAR compiler allows declarations of the form `int x @ 16;`, indicating that `x` should be at address 16. Multiple variables can be aliased in this way.

CodeSurfer must also be able to recognize all of these compilers, and be able to interpret their command-line arguments and adjust accordingly. CodeSurfer uses the EDG front end for C and C++ parsing [3], extended with modes that allow it to correctly parse these different dialects. *Compiler models* are used to determine which mode to use. These are also responsible for translating command line flags for particular compilers into flags that can be interpreted by the modified EDG front end.

Many compilers document their extensions very poorly, and often accept as legal a superset of what the documentation specifies. It is difficult to anticipate these in advance, so the EDG parser has been modified so that if it encounters something unexpected, it will issue an error message, recover at a suitable restarting point, and keep parsing.

In addition to this, extreme usage of language features is commonplace. Some examples we have encountered are the following:

- The Boost library [1] uses template metaprogramming techniques heavily. A Boost component implementing a parser has an identifier name whose templated expansion is over 12,000 characters long. When mangled without compression, this yields an identifier over 60,000 characters in length. All such mangled identifiers require in excess of 8Gb to store.
- Automatically-generated static initializers of extreme lengths, sometimes with hundreds of thousands of elements on the same line.
- A source file, which when expanded by the preprocessor was over four million lines long.

These extremes are also difficult to anticipate, so the generation of the intermediate representations has been engineered to be somewhat fault tolerant.

Finally, there are unusual usage circumstances. Some users wish to analyze code but cannot even compile the code because they do not have access to the original compiler or its header files. This is common in organizations that provide verification and validation services for clients, or for those who are doing investigations of software after it has failed in the field, as described in [7].

4.2. Identification of Code

With any code analysis tool, it is a challenge to determine what source code must be analyzed, as well as how it should be parsed. In the original CodeSurfer, the user did this by simply identifying the files by name, and by specifying preprocessor flags on the command line. In practice, this interface was a barrier to usage for all but the smallest and most simple of programs.

The best approach to this problem is to observe the build system to identify the code to analyze. Unfortunately build systems are notoriously ad-hoc. Some are controlled by IDEs, some use *make*, or similar systems, and some are just shell scripts. Some build systems create a single executable, and others create several executables. Executables are typically constructed from object files, and these may have been moved, renamed, or collated into archives, so the association with the original source file may be obscure. Often some of the code is generated by the build system as it proceeds. Finally, the precise treatment of a particular source file depends strongly on the compiler used, as different compilers specify different preprocessor symbols and built-ins.

The traditional approach to this problem is to replace the actual compiler with the front end used by the analysis, then invoke the build system. When the build system is complete, the files that have been seen by the front end are those that should somehow be incorporated into the analysis.

The replacement of the compiler allows CodeSurfer to:

- Interpret the remainder of the command line to determine what file to parse, and what preprocessor flags to use to parse it.
- Parse the file and create temporary files containing the transitional IR.
- Optionally determine which object files are produced, and to record the association between those object files and the source files.
- Optionally invoke the real compiler. This may be necessary in cases where a program is compiled, then executed to create more code.

There are two approaches to this:

Wrapping the Compiler. The first approach involves changing the build system to change the specification of the compiler name. CodeSurfer simply requires that the string *csurf* be prepended to the command. For example, if the original build commands were:

```
gcc -c foo.c
gcc -o foo foo.o
```

then changing this to

```
csurf gcc -c foo.c
csurf gcc -o foo foo.o
```

is sufficient to create a CodeSurfer project named `foo` comprising a single compilation unit `foo.c`. When CodeSurfer is then invoked, it sees the exact command line, including the name of the compiler that was invoked, which allows it to select the appropriate compiler model.

While this approach is usually feasible, it is often a barrier to acceptance because it requires that the build system be changed, something which users are very reluctant to do. Many build systems are highly fragile and are not amenable to this kind of change. For example, we have seen build systems where different C compilers are used for different components, and where there are dozens of different locations where the name of the compiler is specified.

Process Interception. The second approach avoids the need to change the build system in most cases. With this approach, the user invokes the build system as normal, but in a context where CodeSurfer can observe the process in action. If the process invokes a compiler, then CodeSurfer then intercepts that invocation and steps in and replaces it. On the Windows platforms, this is done by a device driver. On other platforms it is done by manipulating how the process finds its dynamically loaded libraries.

4.3. API

The CodeSurfer internals are all written in C, but the original CodeSurfer API was exclusively Scheme. Scheme was chosen because it was considered to be easy to learn and to program, and because we had experience with *STk*, a robust open-source implementation with a windowing toolkit [2]. The entire user interface is written in Scheme using this API.

The Scheme implementation was extended with primitive types to represent the IR, such as PDG vertices and edges. Each of these new Scheme primitive types corresponds to an underlying type in C, so creating a Scheme value involves wrapping the underlying C value with Scheme cell information.

There were technical disadvantages to Scheme, of which performance was the most significant. In terms of speed, there were two expensive costs. The first is that every time a value is exposed in Scheme, memory must be allocated for the Scheme wrapper. As the IR consists of very fine-grained objects, this expense is non-trivial.

The second cost is with garbage collection. The implementation of Scheme used does a mark-sweep garbage collection of the heap. When the heap is full, and a new cell is needed, then a garbage collection is initiated, which scans the entire heap. This means that as the heap grows monotonically there is an inherent quadratic cost to scan it.

The combination of these costs was a performance bottleneck for CodeSonar, and hence a barrier to usage, so a C API was introduced. This is potentially more difficult to use, and garbage collection is the responsibility of the client, but it avoids all the performance problems.

A second reason for the introduction of the C API was the surprising resistance by users to the use of Scheme as a scripting language.

5. Engineering for Generality

CodeSurfer is best known as an infrastructure for analyzing C and C++ programs. However, it has been used to analyze other languages. Languages that have been supported include Jimple (a three-address form of Java bytecodes), Ada 95, SWRL (Semantic Web Rules Language), and x86 machine code. (The only other language currently fully supported is x86 machine code.) For each of these languages, the infrastructure was capable of generating most or all of the intermediate representations, including the system dependence graphs.

At the SDG level, the representation is mostly independent of the source language, with the exception of the grammar of the abstract-syntax trees, so it is possible to create an SDG that correctly models a program built from components in different languages.

This generality is enabled through the use of separate parsers for each language, and *language modules*, which are responsible for providing language-specific services to the component that builds the SDG.

The responsibility of the parser is to create the abstract syntax tree for the file being parsed, and a set of preliminary intermediate representations, mainly comprising a control-flow graph annotated with variable usage information.

When the dependence-graph builder creates the whole-program model, it then relies on the language module to provide the following services:

- The grammar of the abstract syntax trees.
- Accessor functions that allow retrieval and deconstruction of abstract syntax tree nodes.
- A map between abstract syntax tree nodes and the source code positions of the constructs that gave rise to them.
- The creation of a database of ASTs that represent types used in the program.

A separate module provides an interface for the demangling of names.

Because the grammar and the management of abstract syntax trees is the entire responsibility of the front ends and their associated language modules, it is possible to use an existing language front ends that have their own native storage mechanisms. CodeSurfer/C originally used the EDG front end, employing the native file format for storage, and CodeSurfer/Ada used the ASIS interface for abstract syntax trees.

6. Compromising on Principles

In the research community, often a great deal of emphasis is put on correctness of implementation. In industry, we have found that for both CodeSurfer and CodeSonar, correctness is much less important to users than usefulness. The cost of correctness is usually poor performance or poor scalability, both of which are detrimental to utility. As a result we have had to retreat from correctness on several fronts. This section describes some of these.

6.1. Soundness in General

An analysis that is sound will never have a false negative result. With CodeSurfer, a false negative might mean a missing dependence edge where one ought to be, and for CodeSonar it may mean failure to find a real flaw. It is of course easy to achieve perfect soundness, but only at the cost of extremely poor precision.

In some domains, such as compiler optimization, soundness of analysis is a requirement, because without it there is a risk of introducing a bug. For this reason, sound analyses are usually restricted to a fairly limited scope. Many such analyses are intraprocedural for example.

Both CodeSurfer and CodeSonar are much more ambitious in scope: they attempt whole-program interprocedural analyses. We believe that soundness of this kind of analysis is a mirage.

The single most prominent reason for this is there is too much that is unknown.

When an analysis encounters a call to a procedure whose body is not available to the analysis, then it must make an assumption about what that procedure does. The problem is that a fully conservative assumption would be that the procedure can read or write any variable that is in scope, or reachable by pointer indirection; or that it could call any other procedure in scope, including those of the operating system. It may even throw an exception or abort the program. The truly conservative thing to do must be to assume it could do all of these things. Unfortunately this is unlikely to be useful as it would give rise to far too many false positives.

Both CodeSurfer and CodeSonar make some simple assumptions about undefined procedures: CodeSurfer assumes only that the output parameters depend on the values of the input parameters. CodeSonar assumes even less: that the procedure does nothing with the input parameters and the output parameters' values are unknowable.

6.2. Pointer Analysis

CodeSurfer uses a version of the Andersen flow-insensitive pointer analysis [4] that has been extended to handle structure fields and to have a measure of context-sensitivity through inlining. Pointer analysis is another of these techniques which, if soundness is attempted, ends up being so imprecise that it is of limited use.

To address this, the pointer analysis algorithm has been extended with options that reduce its soundness, but which eliminate sources of imprecision. For example, one option allows the analysis to assume that a pointer value cannot be stored in a type whose size is smaller than a pointer. This can be done by chopping a pointer value up into bytes and storing each byte in a character-typed variable. However allowing the pointer analysis algorithm to eliminate this possibility does eliminate a good deal of imprecision.

The disadvantage of these options is that it takes experience and skill to find the settings that are best for a particular program. However, effort on this is richly rewarded as this has the most profound effect on the quality of the analysis. Imprecision in the pointer analysis has a ripple effect on the rest of the analysis. The more imprecise a pointer analysis is, the larger are the points-to sets, which means that the results of subsequent phases are also less precise, and take more time. Consequently, an imprecise pointer analysis may terminate quickly, but may cause subsequent phases to take much longer.

Despite these options, pointer analysis remains the biggest bottleneck and barrier to large-scale analyses. At the time of writing, with the most precise options, it is most often incapable of completing on programs in excess of 100,000 lines of code.

6.3. The Path Inspector

The path inspector was originally conceived as the underlying infrastructure that would allow us to build a general-purpose defect detection tool. Much effort was expended on this, and several queries were formulated, including one to find null pointer dereferences. However, this effort ultimately failed because we could not make it scale to large programs, and because the rate of false positives was unacceptably high. The approach now used in CodeSonar was developed as the alternative.

6.4. Library Models

As mentioned above, CodeSurfer makes a simple assumption about procedures for which source code is not available. For most standard library functions, these assumptions yield a weak model of the program. For example, for the function *strlen*, this assumption means that the analysis would miss the fact that the function dereferences its parameter.

To avoid this, a large number of library models have been developed. These are stubs of the functions that are written to model the real semantics of the function. In some cases, the models are in fact full implementations of the library functions, as is the case with several of the string manipulation functions. However, with functions that eventually call the operating system, it is not feasible to do this completely. These are also sources of unsoundness and imprecision.

For example, a slightly simplified version of the model for the `open` function is shown in Figure 3.

The identifiers beginning with `CSM` are macros that provide a simple abstraction layer helpful for writing models. `CSM_DEPENDENCES3` returns an undetermined value that is computed from the values of the three parameters. The net effect of the `if` statement is to model the fact that `path` variable is dereferenced, and that the result is control dependent on that value and the values of the remaining parameters.

This model does not capture the fact that a call to `open` may change the state of the file system. Most functions that manipulate the file system may change its state. To be sound, models for these functions would have to take into account a potential flow of information between all such functions. In some cases this is acceptable, but it is very rarely what users really want, as the cost in imprecision is too high.

```
int open(const char *path,
         int oflag, ...) {
  int rv;
  va_list ap;
  mode_t dot_dot_dot;

  va_start (ap, oflag);
  dot_dot_dot = va_arg (ap, mode_t);

  if (CSM_DEPENDENCES3(*path, oflag,
                 (int) dot_dot_dot))
  {
      CSM_SETS_ERRNO_TO_NONZERO();
      return -1;
  }
  CSM_ALLOCATE_FILEDESCRIPTOR( rv );
  return rv;
}
```

Figure 3. The CodeSurfer model for the `open` **library function.**

CodeSurfer does provide an option to model the file system as a monolithic entity, but this is not the default, and is rarely if ever used.

Library models have other problems too. Because they are approximations, they may not be good for all potential analyses. A model that is good for data dependences may be wrong for control dependences.

6.5. CodeSonar Limitations

CodeSonar was designed from the ground up to be scalable, have a low false positive rate, and have good performance. As such, compromises on the scope of the analysis were made. It was never envisaged as a program verifier, but as a bug "hunter". As such, retaining soundness was not a requirement. In order to keep the false positive rate low, it was decided to only report warnings if the analysis had a high confidence that warnings are indeed true positives. Users expect analysis tools to be approximately linear in the size of the program being analyzed, and to complete in a small multiple of the time it would take to compile the program, so reasonable close-to-linear performance was a requirement too.

The most fundamental design decision was to make the analysis be bottom up in the call graph. The analysis starts with the procedures that are leaves, then explores paths in those. When the analysis for a procedure completes, summary information for that procedure is computed, and that information is used by effectively inlining it at call sites.

Once summaries are computed for a procedure, the body of that procedure is never examined again, so it can be discarded. This limits the amount of memory in use at any time.

This approach means that the analysis cannot handle recursion, so cycles in the call graph are broken at arbitrary points.

CodeSonar is a path-sensitive analysis, so it explores paths through each procedure. However, as the number of potential paths is unbounded, this had to be limited. First, loops are explored for zero iterations, one iteration, and more than one iteration. This reduced the problem space, but the number of paths remained exponential in the number of branches. To reduce this, an upper limit is placed on the number of paths explored and the time spent exploring them. Once this limit is reached, no further paths are explored for that procedure. Warnings that require universal quantification over paths are then optionally suppressed if they depend on procedures for which this limit was reached.

Finally, CodeSonar does not explore paths that arise because of asynchronous control-flow transfer. Such paths may occur because of interrupt handlers and context-switches between threads. Because such transfers may occur at arbitrary places in the code, a sound analysis would have to consider all paths that these might imply. This is intractable for non-trivial programs.

7. Future Work

This section describes planned developments for the CodeSurfer infrastructure.

7.1. Mixed Language Analysis

GrammaTech has long been working on static analysis of machine code. Future work will allow simultaneous analysis of both source and machine code.

7.2. Rewriting

CodeSurfer will soon be capable of limited forms of rewriting. The scripting language will allow the CFG and normalized ASTs to be rewritten, and for a new source-code representation to be prettyprinted from those representations.

7.3. Incrementality

Currently, the intermediate representations created by CodeSurfer are not reusable, with the exception of those produced by the front end. Thus if a user makes a small change to a source file, the only way to recompute the IR is to throw away the existing representations and start again. This is a barrier to acceptance, as users expect results to appear in time approximately proportional to the size of the change. Future work will address this issue by allowing incremental rebuilds of the IR.

7.4. Web-based User Interface

The CodeSurfer user interface does not scale well to large projects. Coupled with this is the fact that CodeSonar has a web-based user interface that has only very limited program understanding features. Future work will involve migrating more of the CodeSurfer program understanding features to the CodeSonar web-based interface. Scalability to tens of millions of lines of code is a requirement.

8. Conclusion

Many program analysis and manipulation systems have at their core innovative approaches to solving important problems. While these may work well in a controlled environment, there are many stumbling blocks on the way to making them work on the wide variety of programs, environments, and practices encountered in industry. If it is at all possible to have extreme usages of programming language features, it is highly likely that somewhere there is code that has them. This paper discussed how some of these issues were addressed and solved for the CodeSurfer static analysis infrastructure, and the CodeSonar defect detection tool. It is hoped that some of the lessons learned transitioning a static analysis infrastructure from a research vehicle to an industrial strength analysis tool will be helpful to other researchers.

9. Acknowledgments

The success of CodeSurfer and CodeSonar would not have been possible were it not for the contributions of many highly-talented engineers both at the University of Wisconsin and at GrammaTech. Lack of space precludes listing them all, but the following individuals deserve special mention as many of the innovations described here are due mostly or entirely to them: Chi-Hua Chen, David Vitek, Radu Gruian, and David Melski.

References

[1] Boost C++ Libraries. http://www.boost.org.
[2] STk Home Page. http://kaolin.unice.fr/STk/.
[3] The C++ Front End. http://www.edg.com.
[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, January 1990.

[6] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, New York, NY, Dec. 1994. ACM Press.

[7] R. P. Jetley, P. L. Jones, and P. Anderson. Static Analysis of Medical Device Software using Codesonar. In *Static Analysis Workshop*. NIST, ACM Press, June 12 2008.

[8] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.