Bugs errors and mistakes in software
source code analysis with procedure summaries

Volleberg, G.T.G.

Award date:
1999

BUGS ERRORS AND MISTAKES IN SOFTWARE

SOURCE CODE ANALYSIS WITH PROCEDURE SUMMARIES

Name: Guido Volleberg

Mentor: Dr. D. Brand
Supervisors: Ir. G.L.J.M. Janssen
Prof. Dr.-ing J.A.G. Jess
Dr. Ir. L. Stok

Location: IBM T.J. Watson Research Center
Yorktown Heights, NY, USA

Eindhoven University of Technology
Dept. Electrical Engineering, group ICS
Eindhoven, the Netherlands

Period: June 1998 -Jan 1999

Preface

This report describes the results of my master thesis performed at the IBM T.J. Watson Research Center in Yorktown, NY, USA. During this six-month period (mid June 1998 – mid December 1998) I have been working as a co-op student from the Eindhoven University of Technology, the Netherlands. The month January 1999 has been used for gathering/grouping the results, writing this report and preparing the presentation held on January 29, 1999.

The emphasis of my study of Information Technology was very interesting. In this respect my graduation project, dealing with finding bugs, errors and mistakes in software was a very challenging one. Having written many programs one knows that the software development process is not always as easy. Working on this project cleared my view on this point and showed me of what is possible with today's source code analysis methods.

Hereby I would like to thank my mentor at IBM, Dr. Dan Brand, for his daily guidance and coaching and my manager, Dr. Ir. Leon Stok, who gave me the opportunity to work at T.J. Watson. From the Eindhoven University of Technology I would like to thank my mentor Ir. Geert Janssen for suggesting IBM for my master thesis and for his support.

Furthermore I would like to thank the members of the Logic Synthesis group for useful suggestions and discussions. And for all the new people I have met many thanks for making my stay in the USA a very pleasant one!

Finally I would like to thank my family and my parents in particular for their endless support all these years working towards this and for having to miss me for this half year.

Abstract

BEAM is a tool for source code analysis of C++ programs. It tries to uncover various errors, either generic (e.g., bad memory access), or application specific (e.g., bad database access). Within BEAM both procedures and loops are selectively expanded inline, which means that the expanded control and data-flow graph tends to become large. Since not all procedures can be expanded, the side effects of unexpanded procedures need to be summarized. This project concerns the building of these summaries and their propagation through the CALL-graph. By doing so the aim is to report more real errors and fewer bogus errors.

Summary

This thesis deals with source code analysis, i.e., finding symptoms of possible errors in the C++ programming language. Programs are becoming larger and quality (fewer errors) needs to be improved. Software developers are having problems archiving those goals in their programs. Static source code analysis tools can help in reducing the number of errors. BEAM (Bugs, Errors And Mistakes) is such a tool. BEAM tries to find symptoms of errors in source code by statically analyzing it and performing data-flow analysis. This in contrast to dynamic analysis which needs to execute the code. Each procedure that is analyzed may depend on other procedures. A Summary that describes the behavior of a procedure would be helpful during analysis. Selective expansion of procedures will then be possible, based on the available Procedure Summary. With Procedure Summaries, BEAM will be able to report more real errors and fewer bogus errors because of the detailed information available.

Prior to the actual implementation of the Procedure Summaries, the kind of analysis that BEAM performs is explained in detail. Since there are other static source code analysis tools available on the market a comparison is done to describe the advantages of BEAM. Most tools available check only for stylistic errors, especially the "++" part of C++, e.g., class structures. BEAM checks only very few stylistic errors, and focuses more on the problem of finding executable paths leading to a real error. Besides looking for general symptoms, BEAM is also used for finding application dependent error symptoms. Furthermore, measurements on symptoms found showed that the application dependent part is far more powerful (order of a magnitude) compared to finding general symptoms.

Before implementing the several Procedure Summary methods a propagation algorithm was needed which decides (re-) computation for the procedures. This algorithm makes its decisions based on the information available in the CALL-graph. First a post-order list is built to determine the order in which procedures need to be (re-) computed. After a Procedure Summary is computed, the dependencies on other procedures are checked and based on that information the algorithm determines and initiates the (re-) computation of procedures involved.

A flexible framework for the Procedure Summaries was desirable. Only few changes in BEAM were needed for BEAM to be able to use the Summaries. Different Procedure Summary algorithms have to work independently and it should be easy adding new algorithms. Also the propagation algorithm was not allowed to depend on the Procedure Summaries. A framework conforming to these restrictions has been implemented for flexible and extendable Procedure Summaries in BEAM.

Several Procedure Summary algorithms have been implemented, starting with an algorithm providing merely statistics. Conclusions are drawn based on the results of computations with this algorithm concerning demand driven analysis. The second implementation is pointer dereferencing. At the moment this is the only algorithm actually used by BEAM in its analysis with interesting and useful results. More detailed analysis was needed for global variables. Control-flow analysis to determine the order of assignments and to find MAY/MUST information was used. The last implementation is a pointer aliasing Procedure Summary. This also involved control-flow analysis similar to the one for global variables. The data-flow analysis for figuring out pointers was more complicated. For now only the control-flow aliasing part is implemented and the data-flow part is described here in this report.

The core of the Procedure Summary environment for BEAM has been set with the propagation algorithm and several Procedure Summary algorithms. All implementations behave according to expectations and deviations from desired behavior are discussed with their possible solutions. The use of the pointer dereferencing Procedure Summary by BEAM showed impressive results by finding twelve new symptoms. Along with the flexible framework, the basis for the Procedure Summary environment has been realized.

Table of Contents

List of code fragments & figures

Chapter 1. Introduction

Static source code analysis is a useful method for finding errors in source code. Already it has been proven to be a very useful method in software projects to increase the quality by decreasing the number of errors. The programming language C++ is full of hazards and it takes time for programmers to learn to work with/around them. However, it is easy to lose track in programs that consists of millions of lines or more. This is where static source code analysis with data-flow analysis on procedure graphs can help.

Unfortunately source code analysis is not straightforward. Procedures depending on other procedures and recursive procedures make it hard to perform solid and stable analysis. When a procedure changes its behavior in any way all the depending procedures need to be recomputed. Sometimes procedures are expanded inline. This means that when it comes to analyzing a procedure, all procedures that provide information to this procedure are loaded and analyzed. These inline expansions need to be limited in order to handle recursive procedures and to limit the size of the graphs.

A static analysis tool named BEAM (Bugs, Errors And Mistakes) is implemented as a research project at the IBM T.J. Watson Research Center. One of its goals is to improve the accuracy of current static analysis. An option is to develop a method for better handling of the (recursive) procedures. Graph algorithms are under investigation to improve the analysis concerning procedure expansions.

The goal of this project is to implement accurate Procedure Summary algorithms. These algorithms are to be implemented in separate modules. Graph algorithms need to be written to perform the extensive control- and data-flow analysis. A propagation algorithm is needed to decide which procedures need recomputation because of its dependencies.

BEAM is still in its development phase. Other tools are available on the market and a comparison would be useful to determine the strengths of BEAM. Especially the ratio in finding application dependent/independent symptoms and C versus C++ like symptoms is interesting.

In this final report, BEAM's source code analysis environment is discussed; types, classifications and comparisons of analysis and, of course, possibilities. Comparisons of the competitor tools with BEAM are described in Chapter 2. This is followed in Chapter 3 with the problem outline and goals of this project; also the graph structure is described here. The first implementation is the propagation algorithm, which is discussed in Chapter 4. Chapter 5 is devoted to the framework in which the Procedure Summaries are implemented. The algorithms for the actual Procedure Summary computation start in Chapter 6. Statistics, Pointer Dereferencing and Assignments to Global Variables are discussed in this chapter. Chapter 7 is devoted to the last Procedure Summary namely pointer aliasing. The report finishes with conclusions and future work.

Chapter 2. The BEAM project

The demands for software-projects are getting higher and higher. Developers have to develop more features, develop and maintain newer versions on different platforms. Everything in a decreasing development period, without making concessions to the quality of the software. This can only be archived if the software development process is very efficient. These processes include a thorough test phase for improving the quality of the software. Source code analysis of software can add quality to ensure a more effective development process including higher product quality.

2.1. BEAM & Source code analysis

BEAM is a tool, developed at IBM Research, which performs source code analysis on C++ code. The name BEAM stands for Bugs, Errors And Mistakes. It tries to find symptoms possibly leading to these bugs, errors and mistakes (later on referred to as errors) with static source code analysis methods.

Starting with the unmodified source code it first parses the code with a compiler. This first compiler phase is done with an existing compiler that has an open structure. This structure allows tools like BEAM to tap off the parsing phase. The given parse tree is then transformed into graphs, which is the second phase. Each procedure and loop gets its own graph with all information needed to execute that piece of code. At this point all structures have been set up to start with the analysis (one type of analysis has already been performed as is explained later). Before starting with the actual bug finding, BEAM builds a Procedure Summary with information about what happens when that procedure is called. Finally the actual analysis starts where BEAM tries to find the bugs, errors and mistakes in the source code.

Summarizing the phases:

1. Parsing of the source code
2. Graph generation
3. Building the Procedure Summaries
4. Performing the data-flow analysis

To give an idea how subtle the difference between a good and bad program is, a short example is given. In the following procedure, with one input parameter **A**, is either **X** returned or the value "**0**". Which one depends on two if statements. In this case **X** might be uninitialized when returning **X** in the bad program.

```
1: int P( int A )
2: {
3:    int X, I = 0;
4:
5:    if ( A ) { X = 5; I = 1; }
6:
7:    if ( I ) return X;
8:    else     return 0;
9: }
```

Code 1: The good program.

```
1: int P( int A )
2: {
3:   int X, I = 0;
4:
5:   if ( A ) { X = 5; I = 1; }
6:
7:   if (!I ) return X;
8:   else     return 0;
9: }
```

Code 2: The bad program.

With only inverting the if condition at line 7 this program is transformed into a bad program. Following the path (line 1,3,5,7) in the bad program (**A** being **FALSE)** one can see that variable **X** is being read but has never been written. This is an example of an uninitialized variable, so if the program is executed the call of this procedure results in garbage.

Since the difference between a good and bad program can be so small it is hard for programmers to find such errors. This is where source code analysis tools like BEAM can give a helping hand. BEAM can analyze code from realistic programs with millions of lines of code. It has thus been found very useful in finding errors in such programs.

2.2. BEAM's approach

2.2.1. Use of BEAM

Different source code analysis tools have different goals, some of them are discussed in section 2.4. At the moment BEAM is a prototype but the intended use of BEAM in the future is as follows. BEAM focuses on different specific applications, thus it is useful to split the analysis into two parts. One application independent part that should understand the programming language, and the algorithms for finding general errors. The other part consists of separate application specific modules. These modules could be dynamic loadable, to make it easy to add specific error analysis.

Since source code analysis consumes a lot of time (much more than when a compiler compiles its code), BEAM maintains a database of code which has been previously computed. When analyzing large projects only the code that has been updated during the day has to be reanalyzed. This gives enough time during the evening/night to do thorough analysis. Of course code relying on the updated code also needs to be reanalyzed.

As projects evolve new knowledge becomes available about the application specific errors. This knowledge needs to be added to BEAM for being able to find these errors.

Knowing the intended use of BEAM now the goals are explained. BEAM has to be able to handle realistic programs, in practice this means being able to handle millions of lines of code. In contrast with other tools BEAM does not need user information in the code. It takes the code "as is". No adding or change in the source code is wanted nor needed. This keeps the code clean and no other knowledge from the programmer is needed for being able to use BEAM. One older source code analysis tool, lint, was known for reporting a lot of bogus errors. BEAM rather misses a real error than reporting bogus errors. When however in the error report from BEAM the user decides that an error is a bogus one, the error can be turned "off" so that in the future this error will not appear again.

Speed is of minor importance to BEAM, it has no intention of running at compiler speed. A compiler does not need thorough code analysis for being able to build a running program. Finding errors is more expensive in time thus it is allowed to take this time and with the updated code recognition BEAM is well able to handle large pieces of code without the need of running at high compiler speed.

The following goals are of less importance to BEAM. Reporting all errors, handling arbitrary programming languages e.g., Java, Pascal or perhaps even VHDL and handling arbitrary applications. Section 2.3 describes the strength of BEAM. It is better in finding application specific errors than finding general C/C++ errors. There are other tools on the market available (section 2.4), which handle those kinds of errors better than BEAM.

2.2.2. Analysis classification

In the testing phase of large software projects, developers have a wide variety of choices available for detecting errors in their implementation. The first way of finding errors is through dynamic analysis. This type of analysis provides possible inputs and then checks the output for correctness. Programmers are able to use this type of methods because they know how the program should behave and thus know the input/output relation. If one could cover all the possible inputs a program can be thoroughly tested, however due to the size of today's software projects it is almost impossible to cover all possible inputs. For dynamic analysis to work, it needs to actually execute the program in order to find the input/output relation ship. This is a disadvantage because a target machine is needed.

Another possibility is static analysis, this can be divided in symbolic simulation, data-flow analysis and state space exploration. Without explaining the methods in full, some characteristics are given. Symbolic simulation searches the procedure graphs in DFS order for all possible paths. Data-flow analysis does the same but then in BFS order. State space exploration tries to find all reachable states.

The last possibility is test case generators. These are a combination of both dynamic and static analysis. The principle is to find interesting input patterns for dynamic analysis. Ideally they use static analysis to determine what these interesting input patterns are.

BEAM is classified as a static source code analysis tool. More specific, it uses data-flow analysis to find the symptoms leading to a bug, error or mistakes.

2.2.3. Classification of data-flow analysis

Different types of data-flow analysis are available from low accuracy but fast to high accuracy but slow in time. Flow insensitive analysis is very fast but does not have such a high accuracy as state sensitive analysis. In the following paragraphs it is explained, with examples, what the exact difference is including what types of analysis BEAM uses. During this example the same program is used as in section 2.1.

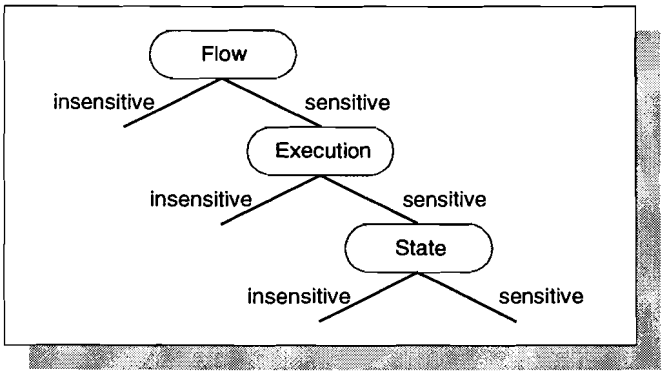


Figure 1: Data-flow classification.

Flow insensitive analysis looks at the statements as an unordered set. In the example below we see the statements concerning **X** in **bold** font. Since the order is unknown, this type of analysis can not distinguish the good from the bad program given in section 2.1. Flow insensitive does only work for detecting uninitialized variables if that variable is never assigned anywhere in the program.

```
1: int P( int A )
2: {
3:   int X, I = 0;
4:
5:   if ( A ) { X = 5; I = 1; }
6:
7:   if ( I ) return X;
8:   else     return 0;
9: }
```

Code 3: Flow insensitive.

Flow sensitive & execution insensitive is more detailed in its analysis. It looks only at the variable **X** but in contrast with flow insensitive it now knows the order in which the statements are executed. Plus it sees that some statements are executed conditionally but because it treats executable and non-executable paths the same it still can not distinguish between the good and bad program.

```
1: int P( int A )
2: {
3:   int X, I = 0;
4:
5:   if ( A ) { X = 5; I = 1; }
6:
7:   if ( I ) return X;
8:   else     return 0;
9: }
```

Code 4: Flow sensitive and execution insensitive.

The next step is execution sensitive & state insensitive. This does take the conditions into account but does know nothing about the assignments to the variables. Also it does know nothing about the relationship between **A** and **I.** This type of analysis can still not determine good from bad in the example.

```
1: int P( int A )
2: {
3:   int X, I = 0;
4:
5:   if ( A ) { X = 5; I = 1; }
6:
7:   if ( I ) return X;
8:   else     return 0;
9: }
```

Code 5: Execution sensitive and state insensitive.

State sensitive analysis looks at everything including assignments to other variables. Only with this extended information gathering BEAM is able to distinguish good from bad programs.

```
1: int P( int A )
2: {
3:   int X, I = 0;
4:
5:   if ( A ) { X = 5; I = 1; }
6:
7:   if ( I ) return X;
8:   else     return 0;
9: }
```

Code 6: State sensitive.

BEAM is able to find many errors, some of which are easy to find. It does not use expensive state sensitive analysis to find these errors. Examples of these errors are a mismatch in a printf pattern or dead code. Errors like this can be found for all paths, it is not necessary to search for a specific path leading to these errors. Errors that do need a specific path can be divided into two groups. One without constraints e.g., uninitialized variables and memory leaks and errors with constraints, e.g., dereferencing **NULL** and index out of range.

2.2.4. Comparison of goals

Different tools have different objectives. Knowing the differences in data-flow analysis, let's look at the comparison of goals of other tools using data-flow analysis.

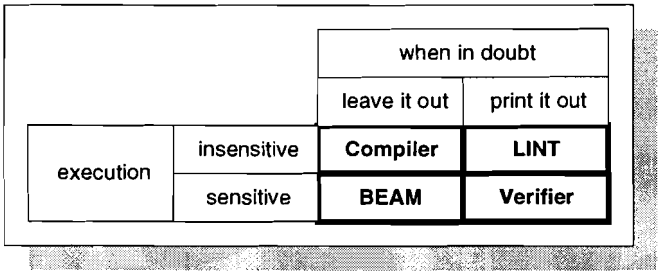


Figure 2: Comparison of goals.

The diagram shows 4 quadrants divided by execution sensitive/insensitive and yes/no error report. Compilers and LINT-like programs are alike in their analysis, both are execution insensitive thus not that detailed. In case they find something LINT will definitely report it, which results in a lot of bogus (not-real) errors. Compilers in contrast to this, just leave it out, nothing is reported unless it really affects the program. Verifiers and BEAM are more precise and use both execution sensitive analysis. But in error-message printing they differ. BEAM wants to prevent bogus errors and thus if it is not sure it reports nothing where a verifier does print an error message.

BEAM uses execution sensitive analysis to make sure that a path leading to an error is actually executable before reporting it. This is explained with the following example.

```
1: int Q( int *p )
2: {
3:   return *p;
4: }
```

Code 7: Path to an error in procedure Q.

The problem in this program, what if **p** = **NULL**? BEAM will not report the error because it can not find enough evidence that **p** might actually be **NULL**. This in contrast to a Verifier who will report the error because **p** might be **NULL**. For BEAM this is solvable if it knows the calling environment of this procedure Q. But the calling environment might not be known due to different reasons; not computed yet or simply not available for analysis.

2.3. *Achievements*

The tool BEAM has been used on various large software projects. Symptoms found by BEAM are often reviewed "by hand" and then reported. In total BEAM is able to detect about 45 symptoms of errors. They can be divided in four different groups ∀ path, ∃ path, generic and application specific.

	∀ path	∃ path
General	• Never used variables • Incorrect number of printf arguments	• Alloc/Write/Read/Dealloc • Index out of range
Application specific	• Database check not done	• Exiting without clean up • Open/.../Close database errors (not implemented)

Table 1: Symptoms types.

Leaving out the application specific symptoms that BEAM is able to find, still an extensive list of general/application independent symptoms can be found.

Application Independent detectable symptoms by BEAM	
• Uninitialized location	• label never referred to
• returning deallocated location	• missing break in a switch
• memory leak	• mismatch in array initialization
• deallocating an already deallocated location	• procedure never called
• accessing a deallocated location	• procedure never called except by itself
• variable assigned, but not used	• bad printf argument type printf("%s", 123)
• variable used, but not assigned	• not enough printf arguments
• variable neither assigned nor used	• too many printf arguments
• parameter never used	• scanf argument is not a pointer
• parameter is a structure rather than a pointer to a structure	• bad printf option printf("%y")
• statement has no effect	• local variable masks a global
• binary op (e.g., &) inside an if condition (instead of &&)	• constant expression
• bit wise operation (e.g., &) between two relations	• assertion
• casting a value into a type not containing the value	• dereferencing NULL
• assignment inside if	• out of range
• unreachable statement	• passing pointer to deallocated memory
• backward goto	• string index out of range
	• duplicate external procedure
	• NULL class pointer

Table 2: Application independent symptoms.

BEAM tries to find symptoms and not errors because for example an unreachable statement does not influence the program execution. Or one might say; "if I initialize all variables with zero, BEAM would never report any uninitialized variables". True, but this is only hiding the symptom of what might become an error. The next chart gives an idea of the number of symptoms categorized for application specific and general symptoms found by BEAM.

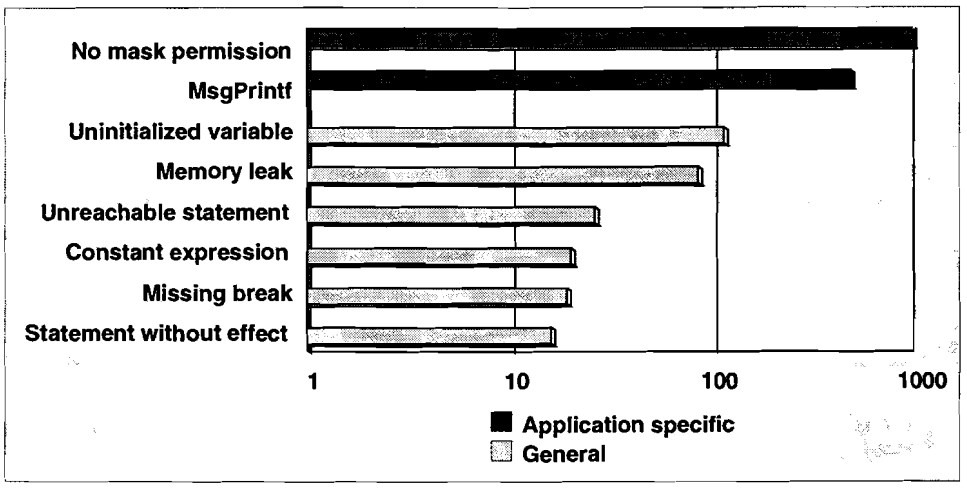


Figure 3: Symptoms found by BEAM.

On the y-axis the symptoms are depicted as possibly becoming errors. On the x-axis the number of occurrences it displayed. It is of minor importance of looking at the actual numbers because they are accumulated over time. Nor is the explanation of the specific symptoms important. What is relevant is the ratio between the between the application specific symptoms at the top and the general C++ symptoms below in the chart.

The difference between these kinds of symptoms is an order of a magnitude. This is one of the strengths of the source code analysis tool BEAM. BEAM has a lot of application specific knowledge built in for specific projects. It thus can find errors that no other tool is able to find unless it has programmable analysis capabilities.

2.4. Competitors

There are, of course, other source code analysis tools available on the market. These tools focus on different goals than BEAM. In section 2.3 it is made clear that BEAM's advantages are mainly in the field of application specific symptoms. These kinds of symptoms are not checkable by the tools discussed here (unless they are programmable). Thus it is not meant to compare BEAM with other tools available, however this is sometimes done to indicate choices made in BEAM. This section intends to give a broader view of available tools with their advantages and disadvantages, where to get more information about them and some interesting Internet links. This section is based on [MEYE97] and [SCZE97].

C++ is a widely used programming language, especially because of its power and flexibility but also because of its hazards. Hazards like not implementing a virtual destructor in the base class often result in incomplete destruction of the derived class objects, when they are deleted through base-class pointers. These very specific C++ hazards are not checked by BEAM. Programmers who are very experienced in the C++ language often learn to avoid these kind of troublesome constructs. But it is a misconception that such experience should be necessary for writing good programs. Also these kinds of errors can often be found through static source code analysis.

2.4.1. Available tools

So what tools are available on the market for statically analyzing C++ source code. Tools performing dynamic analysis are ignored. However these tools may also be found very useful in large software projects but they are not related to BEAM. Dynamic analysis complements static analysis and does not replace it. Also tools concentrating on lexical issues (identifier naming and style) are not part of this research.

The following alphabetical list shows the investigated tools with some important characteristics.

- **The Apex C/C++ Development Environment from Rational Software.** This tool includes, among other capabilities, 22 predefined rules concerning C and C++ programming under UNIX.

- **C++ Expert from CenterLine Software.** Static and dynamic analysis is performed on both C and C++ source code. The static checks are derived from two books, [MEYE92] and [MEYE96]. Furthermore this tool provides links in its reports to Internet links of online versions of these books. Also this tool runs under UNIX.

- **CodeAdvisor from Hewlett-Packard.** HP's SoftBench development environment for UNIX includes this CodeAdvisor analysis tool. Included are 23 predefined rules plus the possibility is given to extend its capabilities by coding new analysis in C++.

- **CodeCheck from Abraxas Software.** This is one of the few tools available for multiple platforms, DOS, Windows and UNIX. Like HP's tool, new rules can be programmed but in a C-like language. Several predefined analysis programs are already provided for metrics and code-portability.

- **CodeWizard from ParaSoft.** A tool only for UNIX with 24 rules selected from the book [MEYE92].

- **FlexeLint/PC-Lint from Gimpel Software.** Again a tool designed to run on multiple platforms, UNIX, DOS, Windows and OS/2. Concerning rules this one exceeds all others, it is able to check over 600 rules in both C and C++ source code. Maybe this one is the closest to the classic lint but in contrast it is able to perform more detailed data-flow analysis.

- **QA/C++ from Programming Research,** another tool for UNIX only. Like BEAM it works in multiple phases. It first parses the C and C++ code and stores its information in a database (graphs in BEAM). Different analysis may then be run on this database. The modularity however does not provide user programmable analysis.

More information and company (Internet) addresses about the tools can be found in Appendix A.

A note about the two books [MEYE92] and [MEYE96], both books contain guidelines and rules for C++ programming. Both books written by S. Meyers, a software-development consultant, are well received in the C++ programming community. A lot of corporate-coding guidelines currently existing in companies have been derived from these books.

Today's compilers are also getting better at performing data-flow analysis. They already did perform this type of analysis to find for example def-use associations. However in contrast of what people might think the extensiveness of analysis performed is still not close to what a static analysis tool like the ones described above and BEAM do.

The GNU G++ compiler is known for its extensiveness in compiler messages (run with options `-ansi -pedantic -Wall -O`). In [MEYE97] five compilers are tested on their warnings about troublesome C++. 36 rule violations were tested of which 3 compilers identified no violations and one compiler identified only one. GNU G++ was able to identify the most, 2 out of 36. This confirms the above that compilers analysis is not as thorough and not usable for the same purpose.

2.4.2. Comparison

The comparison approach for the tools mentioned above, is as follows. First a set of rules on the structure of C++ is developed. The set is tried to be representative of the kinds of errors real programmers would find useful. The rules, see Appendix B, are divided into several categories. Noticeable is that almost all rules are stylistic which implies that not all of them result in errors if not applied. They are merely there to protect the software from becoming hazardous. When rules like these are followed the software becomes more reliable.

CodeCheck and CodeAdvisor were very good at detecting rules from the list. However not directly, for most rules the vendor claims that the user can program the tool to detect violations of the rules. The question which symptoms/rules are checked by BEAM is simple, none! However some are related and have been found useful in directly preventing an error. Below is a list of those symptoms with the reasons why:

- **Symptom 16)** *Use pass-by-ref-to-const instead of pass-by-value when both are valid.* BEAM does print a warning when a structure (or a class) is passed to a function, as opposed to a pointer or reference. The reason is that passing a structure invokes copy constructors and destructors behind the scenes, which is not only inefficient, but sometimes causes problems. But BEAM has no preference for pass-by-ref-to-const as opposed to pass-by-ref.

- **Symptom 18)** *Do not overload on a pointer and an integer.* BEAM checks code on both 32 bit and 64 bit machines. Since compatibility issues are sometimes a (application specific) problem, BEAM checks this.

- **Symptom 23)** *Avoid use of "..." in function parameter lists.* BEAM has nothing against "..." but flags cases where a structure (or a class) is passed to a "..." argument. The reason is that people are normally not aware that the compiler will handle this differently from ordinary arguments.

Most tools focus on handling the "++" part of C++ this in contrast to BEAM, which application independent symptoms are more C like. Most tools also do this but unfortunately no information was available for comparing that part with BEAM's symptoms. Lexical analysis is also a possibility in some tools. In the first phase of BEAM some lexical analysis is performed but not more than a regular compiler would do. If it can parse the code it is satisfied and does not report any errors.

Since BEAM is designed to handle large portions of code it would be interesting to know how these programs would behave in such an environment. The only information available about this is that most tools have "trouble" handling such programs.

Another point of interest is the precision of the tools. State sensitive analysis is needed to find for example uninitialized variables (section 2.2.3). Most tools do not perform that kind of thorough analysis, which only increases the number of bogus errors reported by them. BEAM achieves one of its goals of few bogus errors with taking the time for state sensitive analysis and thus has a clear advantage over the other tools.

Chapter 3. Problem outline & goals

3.1. Procedure expansions

Before addressing the main objectives of this thesis an introduction is given about BEAM's analysis. This is followed with a proposal for improving the current situation with Procedure Summaries.

3.1.1. A bug's life

From hereon only errors (bugs) are discussed happening along a specific path. When handling large programs, in the order of a million lines of code, a path leading to a symptom can be a billion assignments long. Finding paths is even more difficult when conditions and branches are considered. The paths must be checked for being executable. Also all statements where memory is referenced need to be checked.

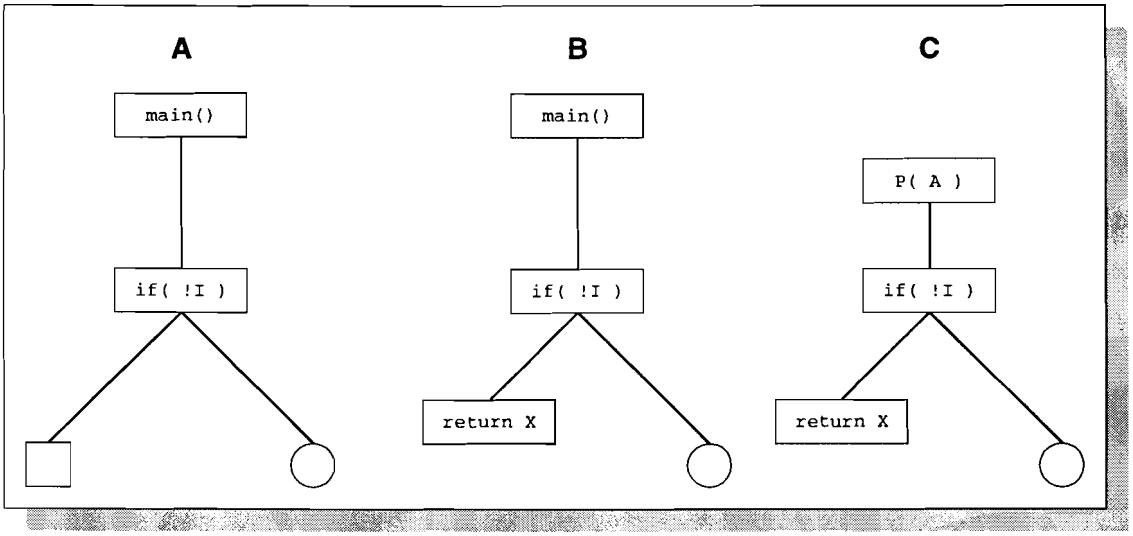


Figure 4: Shortening paths in BEAM.

The above diagram shows how to shorten those paths. BEAM is only able to compute the paths if they are reduced in length. In Figure 4, A the normal situation is depicted. Starting with executing `main()` and some billion assignments the paths diverse into the actual and the intended execution. Again after a billion assignments the symptom is executed (a recognizable fault). This may lead to a crash or just different than expected results (a circle instead of a square).

Looking for earlier symptoms can shorten the path. In Figure 4, B an uninitialized variable is detected at the `return X` statement. There is one hazard because what is wrong with passing garbage around? Of course it is only wrong when it is directly effecting the result. Looking for earlier symptoms can be done in various ways. Compilers look for application independent symptoms. Another approach is searching application specific symptoms. BEAM uses both methods. Verifiers tend to use assertion when looking for earlier symptoms.

Starting the path at the entry of procedure P and not at `main()` reduces the path even more (see Figure 4, C). Also this approach has disadvantages; the author of P might rely on a specific state of memory for its procedure P or a specific relationship between parameters and memory. Other tools also use a different starting state however they make different assumptions about them. Compilers tend to say that everything is unknown, verifiers say everything is possible. BEAM takes another approach, which is not discussed here, single- and multi-statement reachability.

Now the path has been significantly reduced but is there more? Procedure calls make the paths to investigate longer. Most of the calls are irrelevant to the execution, which gives possibilities to shorten the path. Important is to know which calls can be ignored so more information about the side effects of the procedures is needed. Verifiers rely on the user to explicitly assert the side effects. Compilers assume anything, i.e., arbitrary side effects of a procedure.

BEAM has three approaches for assuming the side effects of a procedure depending on the information available. If none is available and nothing can be made available it assumes no side effects. Second if a procedures side effects have been computed and been stored in a Procedure Summary it uses that approximate information. Suppose a procedure updates/changes relevant variables but the information supplied by the Procedure Summary is not sufficient then BEAM will expand the procedure inline. This expansion means that BEAM will try to find an executable path through the procedure.

Other burdens on long paths are loops. BEAM treats loops exactly the same as procedures with the disadvantage that only little iteration can be considered. But in most cases symptoms are reproducible in only a couple iterations so it suffices for BEAM analysis. Verifiers try either loop invariance or convergence of states. Compilers also try convergence but use data-flow instead.

Paths under investigation have multiple conditional statements. These conditions increase the complexity of the paths mainly because the number of paths to be considered becomes larger. Conditions are not reduced by a certain method but data-flow takes care of them. This is the situation in BEAM and also for most compilers, verifiers tend to use symbolic simulation.

3.2. Procedure Summaries

Expanding procedures and loops inline means that all graph information needs to be in computer memory for analysis. Consequently it is important for memory size considerations and time that procedures can be selectively expanded. Based on information available in Procedure Summaries this selection can be made.

The information in the Procedure Summaries should represent the procedure's behavior to the program. But the question is, what is representative? Several sorts of assignments influence the program's behavior. The goal for the Procedure Summaries would be to implement all of them however this was not possible due to time. Here are some items listed but the list is not meant to be complete.

3.2.1. Information in Procedure Summaries

- **Global variables**: global variables can be read and written in all procedures. Thus if one procedure writes a global variable this may influence an if-statement in another procedure. These global variables include static variables in classes.

- **Pointers**: pointers may point to objects which are either local or global. Also more than one pointer may point to the same object which is then called an alias. Procedures with assignments to pointers, aliases and dereferenced pointers can influence other procedures and are thus interesting to include in a Procedure Summary.

- **MAY/MUST information**: a program normally consists of multiple if- and switch-statements. These conditions are influencing the control-flow of a program. Assignments to pointers/globals etc. are thus conditionally and consequently MAY happen or MUST happen when executing the procedure. If this information with each assignment to e.g., a global would be included in the Procedure Summary, BEAM could then do more specific analysis. For example knowing from the generated Procedure Summary that a specific global variable MUST change when calling a procedure, there would be no need to analyze the procedure any further. That assigned value can be directly used in the analysis. In contrast with MAY information, when a global MAY has a certain value, BEAM could start a more specific computation by expanding the procedure inline.

- **Values**: values can be included in the Procedure Summaries and thus making them more detailed. Of course it sometimes is too hard to figure out what a certain value should be, e.g., not enough information available, thus values marked as unknown are also valid. Further analysis by BEAM will be necessary to figure these out.

- **Range of variables**: apart from one specific value sometimes ranges of values are in order. Suppose a procedure with a for-loop; the value, which controls this loop, is allowed to have a value within the range from which it starts until it ends. Initially only the values will be used but in the future ranges might also be implemented.

- **Procedure dependence on global state**: in what sense does a procedure depend on, for example the state of the heap. If there are certain restrictions to this, the procedure, when called, might provide different results. This is particularly interesting in procedures that perform hardware memory operations and thus often rely on the state of that memory.

Implementing Procedure Summaries only provides information to the analysis part of BEAM. With this information BEAM can make decisions whether what the procedure does is relevant to the analysis currently performed. The analysis part needs to be renewed in order to use the Procedure Summaries and to be able to make decisions based upon them. One of them can be to take a closer look and to expand the procedure.

3.2.2. Benefits for BEAM

The information provided by the Procedure Summaries gives BEAM more information before starting its analysis. With this information BEAM is able to perform faster analysis; procedure expansions are not always necessary. It is also possible for BEAM to detect more errors and report fewer bogus errors.

An easy to use, expandable Procedure Summary framework would increase flexibility for adding new Summary information and application specific information. Efficient propagation of the computed information to other procedures will also enhance BEAM's Procedure Summary analysis.

3.3. Memory model and graph representation

BEAM uses a specific kind of memory model, which is enhanced for analysis but does not always correctly represents a real computer memory. The graph representation that BEAM uses is also focused on analysis. This section describes these working environments for BEAM.

3.3.1. Memory model

The memory model BEAM uses for its analysis ignores any hierarchy. It is a monolithic memory with an unlimited number of locations of all possible sizes. There is no relation between the locations, which implies that pointer-jumps to other locations are not possible. A NULL pointer is identified with a special location. The result of a malloc will thus be either the actual allocated location or this NULL location. Pointers are always a pair; location and offset. The offset will always be linked with its location what again implies no pointer-jumps possible. If the offset however exceeds the size of the location, it is out of range. BEAM is able to handle this. If memory is deallocated it will never be used again. So, if a program relies on the allocater for supplying certain memory, BEAM will not be able to detect errors violating this.

How does BEAM handle the distinction between a variable, locations and values? First, variables exist only during parsing. If a program is executed only the memory location exists, which is the same in BEAM, during analysis only the locations are visible. As a variable can be a specific value, a location in BEAM can hold a certain value.

3.3.2. Graph representation

BEAM uses the VIM representation for its graphs. VIM is an enhanced graph representation. More precise, VIM is a network representation because there are more entities possible than edges and nodes. But within the BEAM project these networks have always been referred to as graphs. Therefore will this thesis use the name graphs for these networks.

Using VIM enables BEAM to do easy traversal through graphs (back and forth) storing information on nodes and edges. In this section the graph that BEAM uses is explained along with the terminology used.

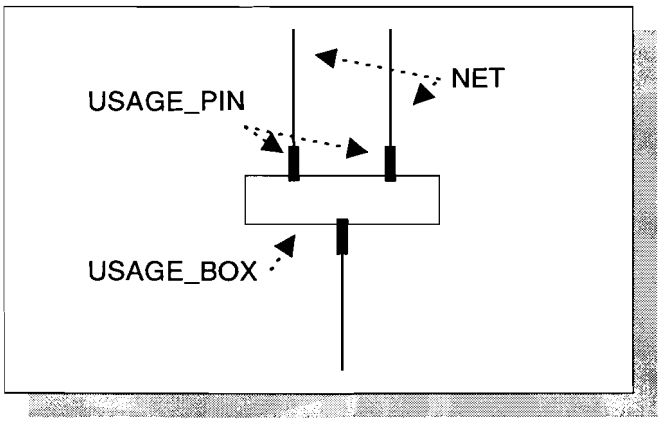


Figure 5: The graph representation.

A node in a VIM graph is called a USAGE_BOX. A USAGE_BOX can have one or more pins (called INPUT_PIN or OUTPUT_PIN). These pins, called a USAGE_PIN for USAGE boxes and NETWORK_PIN for a NET, connect the edges in the graph to the nodes. Edges in a VIM graph are called NET's. One NET may have multiple (more than two) pins connected to it, here they are called SOURCE_PIN or SINK_PIN. To store information in the graph keywords are introduced (called KEYWORD_DEF). Keywords are allowed to be of different kinds. Integers, strings, graphs or pointers are valid types. These keywords can be stored on NETS, USAGE_PINS and USAGE_BOXES in the graph. The graph itself is gathered in one box that is called the PROTO_BOX. This PROTO_BOX provides input and output connections as PROTO_PINS. These pins may consequently have NETS connected to them.

There are several functions available to request values/entities from this graph. Without giving an extensive list a few are given to provide the reader with an idea of what is possible. Suppose the example above where starting with the net below, traversal is wanted through the box to the input nets. (Parameter **C** is a context parameter.)

```
1:    NET          Net;
2:    NETWORK_PIN  Q;
3:    USAGE_PIN    P;
4:    USAGE_BOX    Box;
5:
6:    ...
7:    Q    = f_net_pin(Net, SOURCE_PIN, C);
8:    Box  = get_usage_box_from_usage_pin((USAGE_PIN) Q, C);
9:
10:   for(P = f_usage_pin(Box, INPUT_PIN, C);
11:       P;
12:       P = n_usage_pin(P, C))
13:   {
14:     Net = get_net_from_usage_pin(P, C);
15:     ...
16:   }
```

Code 8: Backward graph traversal.

The code parsed by BEAM needs to be translated into a VIM graph. About 40 types of boxes are possible and vary from locations to assignments and from integers to bit operations. Not all of the boxes are relevant to discuss; the most important ones are described in Appendix C.

A graph in BEAM can represent either a procedure or a loop. BEAM does not make any distinction between these two. Loops are therefore represented as recursive procedures. A call in a procedure might consequently be either a procedure call or a loop call.

As an example a code fragment is used in Appendix D, this fragment is especially concerning how BEAM handles pointers and will be referred to again in Chapter 7. Important to see is the difference between the data-flow and the control-flow. Both are built with the same VIM components, NETS, USAGE_BOXES etc. The difference from a procedure's point of view is as follows. The path, which can be taken in a procedure, is a path that can be taken in the graph. Dividing branches due to if- and switch- statements belong to these paths. Assignment boxes are always on a path since assignments happen at a certain program point. Thus are they related with the flow of a procedure and consequently occur only on the control-flow. Other implementations of source code analysis tools often do not have such an extensive graph as BEAM has. Their analysis is less precise because of this; another graph structure often used is discussed in 7.2.2. BEAM provides the control-flow part in the graph with a data-flow part. This data-flow part computes precise information for assignments done in the control-flow. It computes indexes, values being assigned, pointers being referenced etc.

In the example in Appendix D the difference between control-flow and data-flow is shown with a dashed versus a solid line patterns. The procedure starts with the assignment **p = &x**. The data-flow therefore computes the "**&x**" with a pointer box. Together with an index "**0**" for location "**p**" it is the first assignment on the control-flow. The next assignment does not include data-flow operations and location "**x**" at index "**0**" is simply assigned "**1**". The last assignment is more complex. Location and index are straightforward but the index needs some data-flow computation. First location "**p**" is fetched which means retrieving the content of location "**p**". The result from this operation is "**&x**" because this is what "**p**" has been assigned previously (to figure that out the control-flow input is provided). The extract boxes now get rid of the "**&**" and thus inputs for the next fetch are "**x**" and "**0**". This fetch node searches along the control-flow for an assignment to "**x**". The result is "**1**" so that is used as an index to the last assign for A.

For best analysis a 1-1-graph translation of the source code is needed. Meaning, the graph describes exactly, in full detail the source code. Unfortunately compilers have the habit of optimizing the code, e.g., substituting constants for variables. Most compilers have options to turn this off and so has the parser that BEAM uses. BEAM is thus able to analyze the clean code like the programmer wrote it. However in some cases there are minor optimizations but they were not found being of influence on the analysis.

Goto's have always been a problem in programs. It is considered best to avoid them but sometimes they are unavoidable. When BEAM analyzes the source code it does not complain about forward goto's. The paths going through such goto's can be correctly analyzed. But BEAM is not able to handle backward goto's and complains if it finds one.

BEAM uses backward analysis on its graphs. For knowing which branch you take it is thus relevant to see a INT_MUX box before you analyze a branch from which you do not know where it came from. For this reason BEAM places the INT_MUX node at the bottom of the branches and not, what is most common, at the top.

Chapter 4. Propagation

In this project we start with the implementation of a system for Propagation of the Procedure Summaries throughout the CALL-graph. This is needed because recursion and loops make procedures depend on each other and thus might need recomputation. In this chapter we explain what a CALL-graph is and how the algorithm works for deciding which procedures need recomputation.

4.1. CALL-graph

BEAM uses a CALL-graph to check the dependencies of procedures. Procedures might have recursive behavior thus the information computed now depends on the information computed before. Loops have the same behavior; each time the loop is entered, previously computed information needs to be included in the Procedure Summary. Thus, BEAM needs to know which procedures are recursive. In the second phase of BEAM, the building of the graphs, also a CALL-graph is build. This CALL-graph is different from the regular procedure graphs. Each procedure is represented with one node and is identified with a name. Dependencies are represented with directed edges. If for example procedure A calls procedure B there will be a directed edge from procedure B to procedure A. This represents the information flow, information computed at B is used in A.

Below are two examples of the following small programs X and Y.

```
Program X
1: void A() { B(); }
2: void B() { A(); C(); }
3: void C() { }
4: void D() { B(); }

Program Y
1: void P() { P(); Q(); R(); }
2: void Q() { P(); }
3: void R() { }
4: void S() { P(); R(); }
5: void T() { R(); }
```

Code 9: Example programs for a CALL-graph.

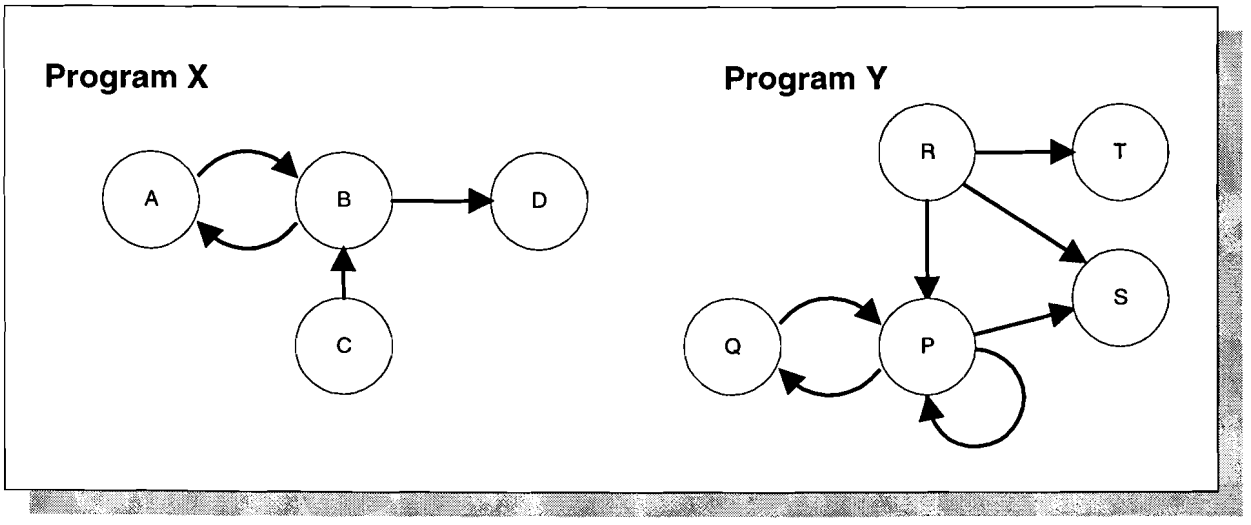


Figure 6: Corresponding CALL-graphs.

Both programs have independent and dependent procedures. Normally each program will have only one main procedure and thus the CALL-graph will have only one sink node. Since BEAM is not always supplied with the complete source code of a project (typically only one directory) it might be possible that there are more sink nodes in the CALL-graph (program Y).

4.2. Post-order list

In one run all procedures in the CALL-graph need to be computed. The question is with which procedure to start/end and why? The algorithm that is implemented starts with procedures with no dependencies. From a graph point of view these are the source nodes. Procedures computed next are those depending on the "source" procedures etc. This happens recursively until only the sink nodes are left which consequently are computed last.

The order of computation described is equal to a post-order traversal of a graph. Such a list starts with source nodes and ends with the sink nodes. The list can be computed in different ways but the algorithm implemented computes the post-order list as follows. It takes an arbitrary node from the CALL-graph to start with. The algorithm knows nothing about the graph so this is the best (and only) thing to do. It then starts a DFS (Depth First Search) on this node but in reverse order, meaning traversing against the arrows. If the DFS hits a source node it adds this node to the post-order list. Then traversing recursively from the DFS back to the starting node all nodes visited are added. At the same time all visited nodes are marked visited. It is then very well possible that not all nodes in the CALL-graph have not been visited. The algorithm continues at the starting point with picking another arbitrary node but now one that has not been visited yet. Followed with performing another DFS on this node and adding the visited nodes to the post order list. This is being continued until all nodes in the CALL-graph have been visited.

Some notes on post-order lists, first the beginning node in the post-order list does not always need to be a source node. Imagine two procedures calling each other, performing a DFS on either one results in a traversal without hitting a source node. This however is no reason for the post-order list to be in-valid or not usable. Another issue is the arbitrary choice of a node from the CALL-graph. Since this is allowed multiple post-order lists may be possible for one CALL-graph. But on the arbitrary chosen node a DFS is performed which means that in the post-order list to be build still the source nodes will be first and sink node will be last.

In the table below are some post-order lists given. They are valid lists for the example programs in the previous section.

Possible post-order lists	
Program X	Program Y
• [A,C,B,D] • [C,A,B,D] • [C,B,A,D]	• [R,T,Q,P,S] • [R,Q,P,S,T] • etc.

Table 3: Post-order lists for program X and Y.

4.3. Loops in CALL-graph

The post-order list is a sufficient order to compute when the CALL-graph does not have loops. Computing the Procedure Summaries for each (depending) procedure, from left to right in the list, will result in stable Procedure Summaries. With stable Procedure Summaries it is meant that they will not change if they are computed again. But each interesting program has at least some loops and/or recursive structures. The post-order list computed is still useable but now an index pointer is needed to indicate which procedure will be computed next. Normally the index pointer will be pointing to the next procedure in the list. When the Procedure Summary has been computed and noted to be different than before a check is performed if there are procedures relying on information just computed. If this is the case a second check is performed if the procedure (may be more than one) is before or after the just computed procedure in the post-order list. With just looking at the index of a procedure stored with a keyword on the procedure node, and the index pointer itself this check can be easily performed.

In practice this means starting with the first node in the post-order list, each node is computed and when dependencies are encountered the index pointer will jump back to recompute the depending procedure. From that point on the procedures to the right of the list will be recomputed and possibly jump back again. This continues until the newly computed Procedure Summary does not change from the previously computed one and thus no new dependencies need to be recomputed. The following example with program X will illustrate this.

The chosen post-order list in this example has been chosen to be [CABD]. First some clarifications on the used coloring. The dashed fill pattern means that this node/procedure is currently under investigation i.e., being computed. The gray fill style indicates that there is a Procedure Summary available (not known if stable).

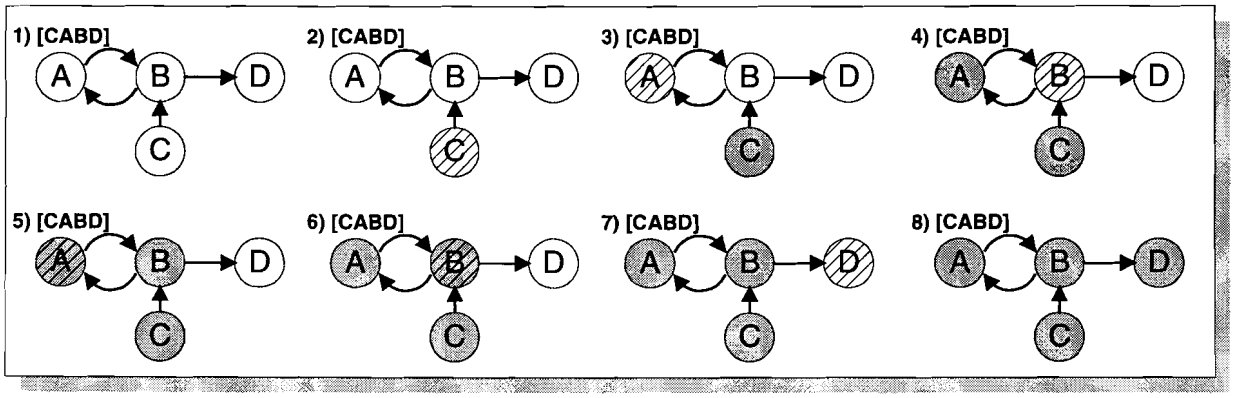


Figure 7: Propagation through the CALL-graph.

The first picture is the start situation, no Summaries available, nothing being computed. Continuing to the first element in the list, picture 2, the Procedure Summary for C is computed. In picture 3 the Procedure Summary for C is available for other procedures, depicted as gray. Presently computed is procedure A, when finished, the Procedure Summary is stored in A and dependencies are checked. In this case A has dependencies to B which means that B needs to be recomputed. B has a larger index in the post-order list and thus will it automatically be (re-) computed, the index pointer does not need to be shifted backwards. Picture 4 shows that B is computed and both A and C have Procedure Summaries available which can be used in B's computation. When finished computing, again a check is done for dependencies. This time B has been renewed and thus A needs to be recomputed. The index pointer is set back to procedure A in the post-order list. Picture 5 depicts the recomputation of A, when finished the computed Summary is compared with the old one. The one that has the most accurate information is stored, for more information about this see section 4.6. Going from left to right in the post-order list B is next for recomputation, again new versus old Procedure Summary is checked plus the dependencies. Suppose B did not change and thus A does not need recomputation, and the algorithm proceeds to situation 7. If B did change, A needs to be recomputed and thus situation 5 is in order again. Hopping back and forth between situation 5 and 6 is done until there is a stable situation exists, no changes in Procedure Summaries. In picture 7 the last element in the list is computed and picture 8 shows that all procedures have a Procedure Summary available.

4.4. Time-Stamp

Imagine in the example that the post-order list was not [CABD] but also a valid one [ACBD]. With the algorithm described so far this means that if procedure A needs to be recomputed, procedure C is also recomputed. But this is not necessary since C is a source node and recomputation will not change the Procedure Summary for C. To solve this problem Time-Stamps are used. Each node has connections to the edges with a pin. It is possible to store a keyword e.g., an integer or long, on these pins. The algorithm puts the Time-Stamp integer that is global, i.e., static, to the Propagation class on both input and output pins. The Time-Stamp is only allowed to increment. By doing so the pins are marked to be newer than the rest with lower Time-Stamp values.

Initially all pins have this integer Time-Stamp set to zero. Just before actual computation of the Procedure Summary the Time-Stamp on the input pin is updated meaning that the global Time Stamp integer is incremented followed by copying this value to the keyword on the input pin. After computation the Time-Stamp on the output pin is updated but only when there is a change in the Procedure Summary, i.e., the newly computed differs from the old one.

The algorithm now works as follows; having picked a procedure for possible computation the question is asked is the procedure is out of date. Out of date means if the Time-Stamp on the source pin of the net connecting to the input pin has a larger value than the value on the input pin. If this is the case, only then will the procedure need recomputation. With this implemented in the algorithm, post-order list [ACBD] will not take any more time to compute than list [CABD] since these Time-Stamps prevent unnecessary recomputation.

4.5. *Deletion from List*

During the first tryouts of this algorithm on large programs, memory problems were encountered, more specifically, memory exhaustion. The post-order list can be very large since there can be many procedures (and loops) in the CALL-graph. The graph for each procedure is already in memory or is loaded when needed for computation. Deletion does not happen until Propagation of all procedures finishes. In the example described, (post-order list [CABD]) C is no longer needed and thus its graph can be deleted from memory right after computation. Note that the computed Procedure Summary is saved as a keyword on the node in the CALL-graph and will always be available for depending procedures.

A second pointer in the list is introduced in the algorithm, the delete index pointer. This pointer indicates that all procedures to the left of this pointer are deleted from memory because they are not needed for recomputation. At the end of the algorithm when the Procedure Summary has been computed and the Time-Stamps have been updated a check is performed to see which procedures can be deleted. If a procedure has no dependencies or the procedure is depending on procedures standing to the right in the post-order list, the procedure can be deleted from memory. This addition to the algorithm was sufficient to solve the memory problem described above.

4.6. *Comparing the Procedure Summaries*

To determine if the Time Stamp needs to be updated a comparison needs to be done with the old Procedure Summary and the new one. This is not implemented as part of the Propagation. Each Procedure Summary decides for itself if it changed. Possible decisions are the size of a Procedure Summary. They are only allowed to grow and if the new one is smaller than the old the old one is saved and the new one discarded. Consequently the procedure will not be recomputed. When the sizes are the same a check is performed to check changes in the Procedure Summary.

```
1:    list = BuildPostOrderList();
2:    {
3:      for (list)                              /* for all elements in list */
4:      {
5:        if (OutOfDate(list[index]))           /* OutTime > InTime  */
6:        {
7:          PutTimeStamp(inputpin);
8:          if (CalcSideEffects(list[index]))   /* a change?  */
9:          {
10:           PutTimeStamp(outputpin);
11:           index = CheckIfDependencies();    /* jump back? */
12:         }
13:       }
14:       DeleteBoxesPossible();
15:     }
16:   }
```

Code 10: Algorithm for Propagation.

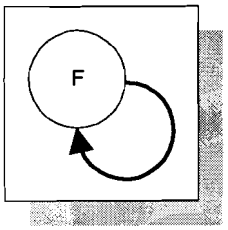
4.7. Algorithm

The algorithm for Propagation is summarized in Code 10.

4.8. Troublesome procedures

There are some special cases for which the algorithm fails to work. The first problem is a procedure for which the Procedure Summary keeps on changing with each iteration.

```
1: void F()
2: {
3:   N = 0;
4:   F();
5:   N = !N;
6: }
```

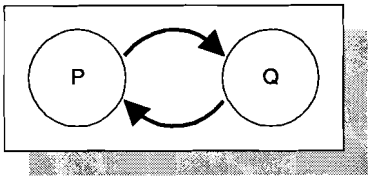


Code 11: "Bouncing" Procedure Summary with corresponding CALL-graph.

The Procedure Summary will be repeatedly computed because of the recursive structure and a changing Summary. With the first computation the Procedure Summary will be "**N = 0**" followed by "**N = 1**" then again "**N = 0**", "**N = 1**" etc. When taking a closer look at the procedure one can see that it is pretty hopeless. A programmer would never have meant to write a procedure like this because there is no path leading out of the recursion. BEAM can solve this problem by using path analysis to find "a way out". However this is too complicated for Procedure Summaries and the only restriction that should be used is to restrict the number of Procedure Summary (re-) computations of a procedure.

The second problem concerns the choice of the starting procedure for computation. The resulting Procedure Summary differs depending on the choice the starting procedure.

```
1:    void P()
2:    {
3:      N = 0;
4:      Q();
5:    }
6:
7:    void Q()
8:    {
9:      N = 1;
10:     P();
11:   }
```



Code 12: Procedure Summary depending on start procedure with corresponding CALL-graph.

The Procedure Summary when starting with procedure P is P: "**N = 0**" and for Q: "**N = 0**". The other way around when first computing Q will result in P: "**N = 1**" and Q: "**N = 1**". Also these procedures on itself are useless because there is no path which leads out of the recursion. Again path analysis needs to be done to solve this problem.

Chapter 5. The Framework

This chapter describes the Procedure Summary framework. It does not talk about each line of implemented code but does give a description of the class structure, also each class is discussed briefly of what it exactly does.

5.1. The Data-structure and motivation

Propagation is the main part of the Procedure Summaries. However this is not chosen to be the class which BEAM accesses. For flexibility reasons it was desired that the Procedure Summaries could be independently used by BEAM. The following framework was the result that is easy to use by BEAM, only few extra lines of code are needed. More computations for Procedure Summaries can be added by putting them along with the other calculations. In addition it provides adjustable (without recompiling) detailed messaging for both Propagation and Calculation. The picture below gives an idea of how the classes interconnect.

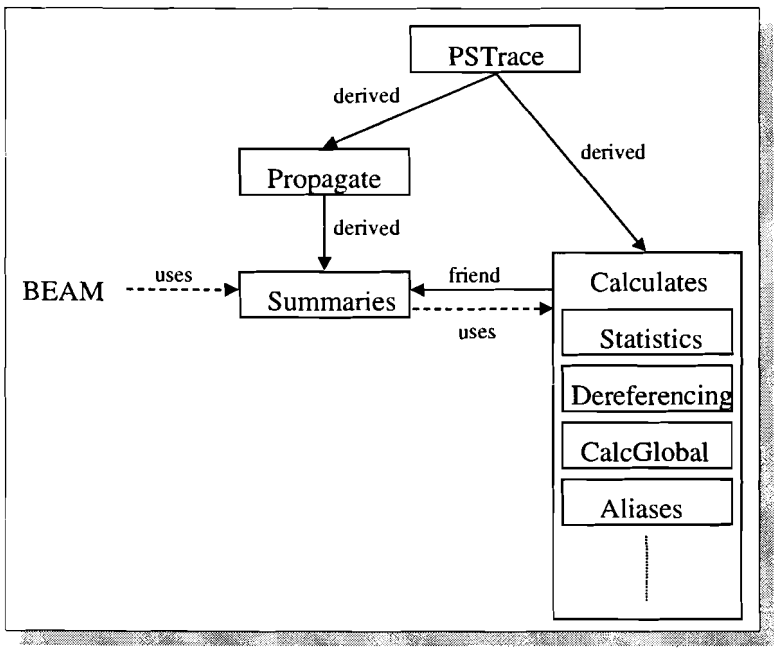


Figure 8: The implemented Procedure Summary framework.

PSTrace: PSTrace is the root class, but does not provide any virtual functions so that Propagate or Calculate can derive them. PSTrace merely takes care of the initialization of the variables that control the message printing used in Propagate and Calculate. All variables are static because only one instance of PSTrace is required. When BEAM calls upon the Procedure Summaries the first time an initialize is called for PSTrace which reads the values of the variable directly from a file. This is useful because no recompilation is needed in contrast to the present situation in other parts of BEAM.

Summaries: this main class stands in direct connection with BEAM, Propagation and the Calculates. First it supplies one header file, enough for BEAM to use the Summaries. Also 2 accessible member functions are provided. These two functions give BEAM the choice to calculate a Procedure Summary for one procedure (initial calculate) or for all the procedures in the CALL-graph through Propagate. When Propagation is used, Propagation on itself calls a (virtual) function in Summaries, which decides the calculations to perform. Several are possible and this is controlled also in a file so no recompilation is necessary. Functions for Procedure Summary handling (save/store) are provided and functions for handling information stored on edges in the graphs during computation.

Propagate: as described in the previous chapter, propagate starts selectively the (re-) computation of the Procedure Summaries and is controlled by Summaries.

Calculates: (Statistics, Dereferencing, CalcGlobals and Aliases) all the Calculates are uniform and have at least one public function CalcSideEffects. This function is being called from Summaries, which initiates the particular calculation. All the Calculates are a friend of Summaries. The functions for storing and retrieving the Procedure Summaries from the nets and boxes are in Summaries.

BEAM: for BEAM are the Procedure Summaries easy to use, after including the header file from Summaries it can initiate the Procedure Summary calculations from which the result is stored at each procedure node in the CALL-graph.

5.2. Other possibilities with advantages and disadvantages

A disadvantage of the implementation is when Propagation decides to recompute a certain Summary. All selected Summaries are recomputed although for example only aliasing was needed. The choice was between either run Propagation over all calculates (present) or over each separate Calculate. If the last situation was chosen each procedure graph needs to be loaded into memory repeatedly for each run. Disk access is time consuming so the choice was made at the cost of recomputation to run Propagation only once.

Chapter 6. Procedure Summaries

The Procedure Summaries consist of four different types; statistics, dereferencing, globals and aliases. Statistics compute the number of boxes of each type in the procedure graph. Dereferencing gives a list of locations being dereferenced in a procedure and Globals computes a list of all globals being assigned in a procedure. Aliases are discussed in the next chapter.

A Procedure Summary information node is a quadruple of four fields. Each Procedure Summary algorithm holds its own Summary, they are not mixed or joined. The Summary class maintains the fields (read/store/combine etc.). The fields are divided into the following groups:

- **Name:** the name of the location (not the variable name)
- **Index:** index in the location to where value is assigned
- **Size:** the length in bytes of the assignment
- **Value:** the actual value assigned to name at index of length size.

These fields are part of a structure that can be used in a linked list. A Procedure Summary will generally be a linked list on which operations such as searching, joining etc. can be performed.

6.1. Statistics

This is the simplest of all Procedure Summary calculations; it does not involve any control- or data-flow traversing. A procedure BEAM graph can have up to 41 different nodes each representing a different data part or operation. To get some insight in the complexity of a procedure it is useful to know how many types of a certain box are used. Note that BEAM does not use these statistics in its computation and for now is not intended to. Papers have been written about demand driven analysis including for example alias analysis. Several papers, [DUES95], [DUES97], [JOHN93] and [ZHAN98], talk about combined analysis (for pointer aliasing). They use different types of analysis (from fast to slow but more thorough) under specific conditions. In general BEAM wants to do more thorough analysis prior to "be fast". But these statistics measurements could provide BEAM with, for example, the number of pointers in a procedure. This could then be used to do more or less sophisticated alias analysis on a procedure.

The algorithm is straightforward; iterating (arbitrary) over all the nodes in the procedure graph it looks at the type of the box. In an integer array are the node-type values stored and updated. This information can be printed on screen and/or stored in a file on disk, which is especially useful when comparing different projects.

The statistics measurements have been used to build charts. These charts indicate the complexity of the selected programs. Since there are 41 possible nodes a selection has been made. We focus on the following groups:

Statistic measurement groups	
Operations	Boxes/nodes in graph
• Integer	INT_ADD, INT_CONST, MULT_INT, MOD, DIVIDE_INT
• Floating-point	ADD_FLOAT, MULT_FLOAT, DIVIDE_FLOAT
• Bit wise	BITWISE_AND, BITWISE_OR, BITWISE_XOR, BITWISE_NOT, SHIFT
• Pointer	EXTRACT_LOCATION, EXTRACT_OFFSET, EXTRACT_SIZE, FETCH, POINTER
• Calls	CALL

Table 4: Groups used for statistics testing.

Each group consists of a different amount of boxes so comparing the amount of floating-point versus integer will give an unfair ratio. What is interesting is to compare the ratio of integers over different sort of programs. For this 6 programs or program parts have been selected. The first

two, tryout and examples have been used for getting to know BEAM. Little programs with errors on purpose, just to see how BEAM handles them. The next four are parts of BEAM's source code from which the first is the new Procedure Summary (PS) source code. Secondly the graph handling part (GR) of BEAM followed by BB which includes the **main()** and another CO in which the conditions are programmed.

The following table shows the result of the statistics in actual number of boxes. Some programs, e.g., GR, are relatively large compared to others. The sections and charts that follow only use percentages.

Programs	Integers	Floating	Bit wise	Pointers	Calls
Tryout	21	0	0	132	119
Examples	39	0	0	242	160
PS	659	0	0	1872	828
GR	3388	13	152	31094	18070
BB	690	0	0	2134	1058
CO	1399	0	145	2930	1186

Table 5: Total number of boxes per operation type.

6.1.1. Integers

BEAM performs its analysis with mostly only integers. The charts can consist of more types but as is shown in the following charts, integers are the largest number in overall types (integers/ floats/ bits). But Figure 9 shows that there is also a large distinction in percentage of integers between the programs. Tryout and Examples are relatively low since these small programs most of the time only have one variable on which a certain bug is tested. PS, BB and CO are more realistic programs so they include more integer computations. This in contrast to GR that does little integer computation, the next section explains why.

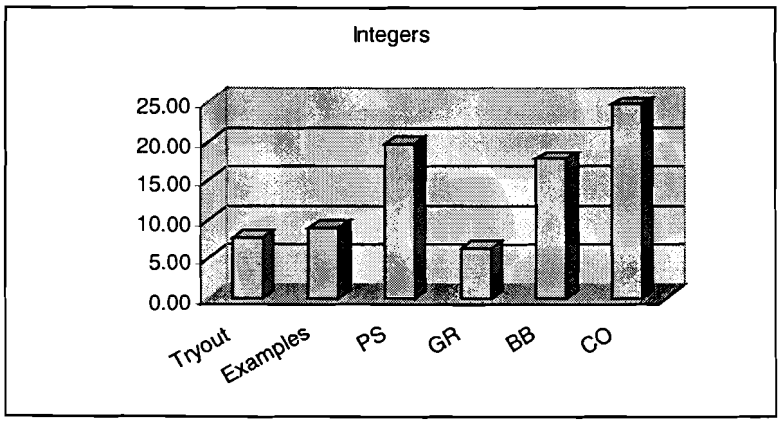


Figure 9: Percentage of integer usage.

6.1.2. Floating-point

None of the programs did any floating-point computations except for GR. However the usage of floating-point operations was only 0.02%. When computing the statistics of other programs the charts will probably look different. Unfortunately no other statistics measurements could be performed. Most likely will programs on for example power estimation in chips have more floating-point operations than the programs discussed here. BEAM will report relative few errors in programs with a lot of floating-point operations. BEAM does not focus on floating-point analysis.

6.1.3. Bit wise

BEAM's highest precision was byte size. A location in the size of one bit was not possible. Currently BEAM's precision has been improved to bit size locations, which again confirms the

high precision goal of BEAM. In the period that these algorithms were written the size was still bytes and therefore everything mentioned in this report uses byte size locations.

Of course in the graph bit operations were possible and thus represented with corresponding operations. BEAM sees these operations as a change in the integer value in that location. Bit operations can be common in programs especially when there is a power of 2 involved. Either the user already optimizes his/her code by using bit shift operations or the compiler performs this type of optimization. In the programs are no bit operations used except for CO (conditions). CO is a piece of code within BEAM, which checks the conditions of branches during data-flow analysis. The percentage of bit operations was very low, 3 percent and thus not usable.

6.1.4. Pointer

Pointers are important for alias analysis described in the next chapter. As mentioned before some papers mention scalable or demand driven alias analysis e.g., [ZHAN98]. It is thus important to know what the ratio of pointers used is in a program. Looking at the following chart there is almost no difference in the programs checked. Almost all programs are written in C++ including classes. This also implies more pointers (e.g., ***this**) than in a normal C program. A note has to be made about scalable analysis. Most papers that use that type of analysis are written from a compiler point of view. For compilers time/speed is crucial and thus there is a lot to gain if they get the same results with faster analysis. For BEAM precision is crucial and from the graph can be concluded that all programs have more or less the same pointer "complexity". It will thus be of no gain for BEAM to use simpler analysis based on this type of complexity.

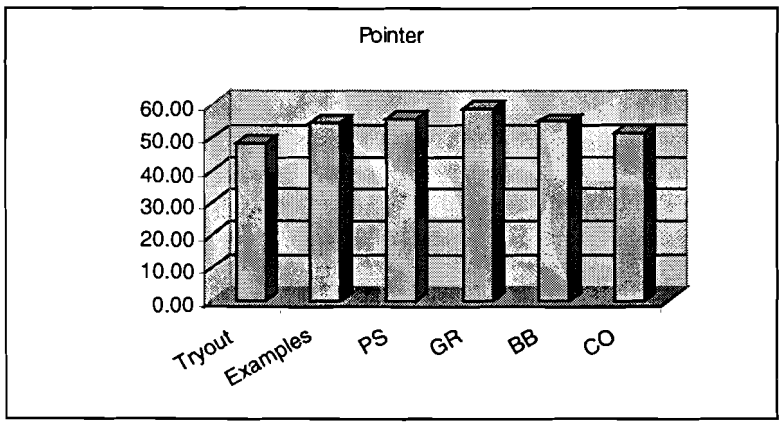


Figure 10: Percentage of pointer usage.

6.1.5. Calls

Another type of statistics measurements was done concerning procedure calls. Unfortunately it is for the statistics not possible to see if the calls are recursive. This would be particularly interesting concerning the Propagation. If there are a lot of procedure calls, more recomputation is needed which consequently takes more time. But presently no estimations on this can be made.

In the next chart it is depicted that especially tryout involves a lot of calls. The reason for this is that a lot of small programs have been written to test the Propagation class. The real programs PS, BB and CO involve about the same number of procedure calls with the exception of GR. GR most likely involves a lot of loops which directly implies more calls since they are treated the same by BEAM.

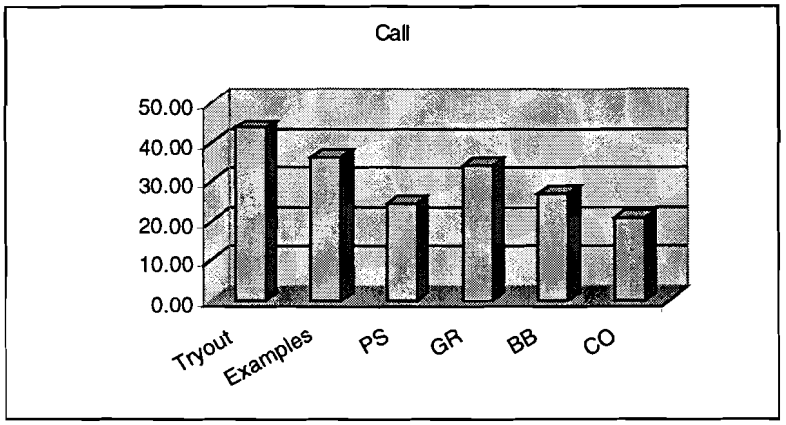


Figure 11: Percentage of Call usage.

6.1.6. Conclusions

The statistics have been written to see if demand driven analysis would be useful in the sense of more or less detailed analysis. The above charts show that realistic programs have more or less the same complexity with the exception for floating-point and bit operations. But those operations can not be checked so there is no time cost involved. We can conclude that the information presently in the statistics Procedure Summary is not sufficient to base demand driven analysis on. These statistics are useful to give the programmers of BEAM an indication of the complexity of procedures.

6.2. Dereferencing of procedure parameters

Dereferencing a pointer happens everywhere in the source code where pointers are involved. But dereferencing is dangerous if the pointer is a NULL pointer, what will be the result of the dereferencing? A programmer normally never intends to dereference a NULL pointer so for BEAM's analysis it is useful to know if a procedure is called which pointers will be dereferenced. BEAM may then perform more thorough analysis if a pointer can actually point to this NULL location. Therefore this Procedure Summary computation is written to summarize all dereferenced pointers.

The following program shows a fragment of source code with a dereferenced pointer location as being a procedure parameter. Consider the lines "..." as being of no influence to **p** or ***p**. The program is perfectly OK if the program is never called like this P(NULL).

```
1: void P(int *p)
2: {
3:   ...
4:   x = *p;
5:   ...
6: }
```

Code 13: Dereferencing.

The implementation for finding the dereferenced parameters traverses the graph along the control-flow. It starts at the bottom and traverses each control-flow node bottom up by asking for its dominator. The dominator node is the node you MUST visit to be able to get to the starting point. This traversing through the procedure graph implies that no branches (i.e., if and switch) statements are considered, at least not in the first version. If the algorithm finds an assign node it then looks if at this assign node a pointer parameter is being dereferenced by doing data-flow analysis. On the right hand side of the assignment is a search started (also bottom up) to find the EXTRACT_LOCATION box. If this box is found only then something is dereferenced and this box must then be connected to an input parameter of the procedure. Only if this can be found the specific parameter is added to the Procedure Summary.

The first implementation has some limitations, no branches, no call boxes plus one level referencing only (**p is considered as only *p). In a second version these disadvantages have been repaired. Presently this is the only Procedure Summary, which BEAM uses in its analysis. The first tryouts on real programs have been very useful. The number of errors found by BEAM using this has increased with 12. These are errors would not have been found without the dereferencing Procedure Summary.

6.3. Globals

A global variable can be accessed in every part of a program. It can be read and written anywhere so assignments done to these variables can influence any procedure in which this variable is used. Three different versions of Global Procedure Summaries have been written, the only really useful one is the last one. The first version is a simple one, it does not consider the control-flow. The second version does include control-flow but no MAY/MUST information and the last one does it all.

The algorithms are described in a pseudo C++ language. Function names will not always represent corresponding function names in the actual source code.

6.3.1. CalcGlobalEffects

This was the first implementation of computing all the assignments done to a global variable and stores them in a Procedure Summary. The algorithm is straightforward:

```
1:    CalcGlobalEffects()
2:    {
3:      ∀boxes in graph
4:      {
5:        if ( is( box, LOCATION ) )
6:        {
7:          if( InLarger Scope(box) )
8:            PS = AddToPS( Info(box) );
9:        }
10:
11:       else if ( is( box, CALL )
12:       {
13:         CALLNode = SystemNodeFromProcName(box);
14:
15:         CALLPS = ReadPSFromBox(CALLNode);
16:         while(CALLPS)
17:         {
18:           if( InLarger Scope(box) )
19:             PS = AddToPS( box );
20:
21:           CALLPS = CALLPS->next;
22:         }
23:       }
24:     }
25:   }
```

Code 14: First algorithm for global calculation.

It iterates over all the boxes in arbitrary order. The order does not matter for this computation because the only thing that is recorded in this Procedure Summary is if a global variable is assigned something. Values are ignored, only "Assigned" is stored in the Summary. The first check in the loop tests if the box is a location box. If this is true it then checks if this location is global. When also this is true the information from the box is extracted and added to the Procedure Summary. The computation so far is sufficient if there are no calls to other procedures or loops. The algorithm above however is able to handle loops and therefore the

else clause is implemented. If a box is a call node then is starts with extracting the node in the CALL-graph and reading the Procedure Summary from this node. It is not sufficient to join the Summary with the one that is presently computed as will be shown in the next example. Each element in the Summary needs to be checked if global and only then can be added to the new Procedure Summary.

This version of the Global Procedure Summary put is only information in the first location field of a Procedure Summary entry, index, size and value are left unknown.

The following segment of source code shows that variables can be global for one procedure or loop but not for another procedure.

```
1: int x;
2:
3: P()
4: {
5:   for (int i=0; i<10; i++)
6:     x = i;
7: }
```

Code 15: Global type mix-up.

The above example consists of two pieces, one procedure "P" and a "for" loop. Two variables exist, **x** and **i**. Variable **x** is global to both procedure and the loop. Analyzing procedure "P" for globals results in getting the Procedure Summary from the "for" loop. The computation from the "for" loop resulted in the following Summary:

```
x[Unknown] (size Unknown) Unknown
i[Unknown] (size Unknown) Unknown
```

To procedure P variable **i** is definitely NOT global and copying this Summary directly into P's Summary would give incorrect results. Consequently it is needed for each element in the Procedure Summary retrieved from a call statement to check for its global-ness. Please note that loops and procedures are the same for BEAM. It is not able to see or detect the difference.

6.3.2. CalcGlobalCFEffects

This second computation of global variables was the first attempt to use the control-flow information available in the graph. For example assignment **x = 5**; after **x = 3**; along the control-flow results in only recording **x = 5**. Backward analysis is suited for this because then assignments to the same locations can be ignored if previous ones have already been recorded.

The algorithm starts at the top and traverses the graph with a Depth First Search. The DFS is implemented as a recursive procedure so this allows us to do backward analysis with the DFS going forward. No nodes are visited more than once. Information computed is stored on the nets and thus available for other branches. Traversal for the example code fragment is given in the following figure.

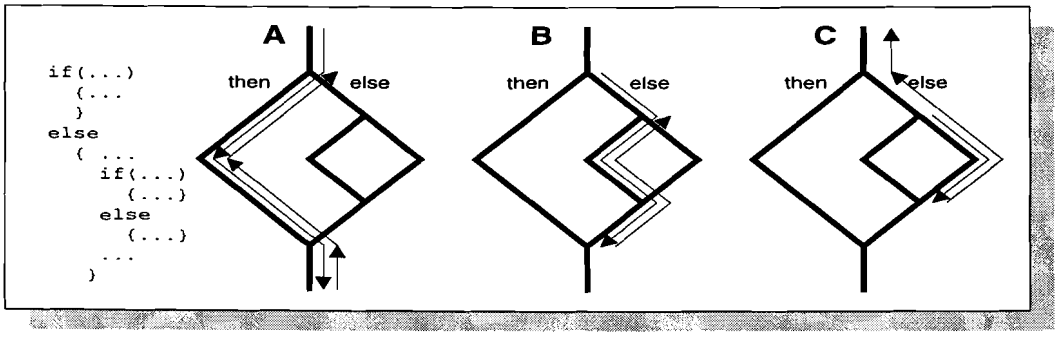


Figure 12: DFS Traversal through the procedure-graph.

Information needs to be gathered to build the globals Procedure Summary. When going up in the graph each node is checked for being either an assign node or a call node. When finding an assign node the algorithm tries to find out what was being assigned. Starting with checking if the location is global, then figuring out the index, size and value of the assignment. If something is too hard to figure out "Unknown" is recorded for that specific field. Finally to decide if this information needs to be recorded in the Procedure Summary a check is performed to see if the same location at the same index and of the same size has not been assigned before. If this is the case the information is added to the Procedure Summary.

CALL nodes are handled the same way as before in globals without the control-flow. However now also the check is performed for each information entry if the location name, index and size are dissimilar. Only then the entry is copied to the new Procedure Summary. The following algorithm summarizes the CalcGlobalCFEffects.

```
1:    CalcGlobalCFEffects()
2:    {
3:      StartNet = GetNetFromInputPin();
4:      DFSControlFlow(StartNet);
5:    }
6:
7:    DFSControlFlow(Net)
8:    {
9:      if( !IsControlFlow(Net) ||
10:           BottomNet(Net) )
11:       continue;
12:
13:     ∀sink_boxes on net
14:     {
15:       box = GetBoxFromUsage(Net);
16:
17:       if( NotVisited(box) )
18:       {
19:         DFSControlFlow(GetNetFromBox(box));
20:
21:         if ( is( box, ASSIGN ) )
22:           figure out information and add to PS;
23:         else if ( is( box, CALL )
24:           figure out information and add to PS;
25:       }
26:     }
27:   }
```

Code 16: Algorithm for CalcGlobalCFEffects.

The algorithm described does use the control-flow to traverse through the graph and to gather information. However no information concerning branches from if- or switch-statements are recorded. The following procedure clarifies this:

```
1: int x;
2:
3: void P(int c)
4: {
5:   if(c)
6:     x = 3;
7:   else
8:     x = 5;
9: }
```

Code 17: Program showing that CalcGlobalCFEffects is incorrect.

The algorithm will take one (arbitrary) branch first, lets say if the condition is **TRUE**. The procedure assigns **3** to the global variable **x**, which is recorded in the Procedure Summary. When it computes the other branch it figures out the assignment **x = 5** but **x** has already been assigned and thus the result will be only:

```
x[0] (size 4) 3
```

This is of course incorrect because first the assignment is conditional which is not made clear in the Summary. Plus possibly assignment **x = 5** is nowhere recorded. The next implementation of the globals Procedure Summary solves this problem.

6.3.3. CalcGlobalCFMMEffects

This implementation uses both the control-flow and the branches for storing MAY/MUST assign information in the Procedure Summary. It also has some demand driven features to only compute branches when needed. Furthermore it uses both backward and forward graph traversal. Going up via the control-flow when figuring out the assignments by doing data-flow analysis and traversing forward in the graph to build the Summary. Finally this implementation tests if the newly computed Procedure Summary differs from the old one.

Before a DFS is started along the control-flow a list is built of all variables that are assigned somewhere in the graph. This list is stored on the first net into the graph, values are for now marked as "Unassigned". The list is used to determine when a branch exists were a variable is not assigned to still make this visible in the Summary. Take for example the following code fragment:

```
1: void P()
2: {
3:   if(c)
4:     x = 3;
5: }
```

Code 18: Conditional assignments.

The Procedure Summary looks as follows (variable **x** is global):

```
x[0] (size 4) 3
x[0] (size 4) Unassigned
```

BEAM can conclude from seeing both entries in the Procedure Summary that the assignment to **x** is a MAY assignment. Variable **x** is either assigned **3** or not assigned at all. If for a specific variable only one entry exists BEAM knows that that assignment is a MUST. Of course BEAM should check if the assignment is done at the same index and is of the same size.

The global unassigned list is built by iterating in arbitrary order over all boxes. Adding them to the list if the box is an assignment to a global. In the previous code fragment this would have resulted in the following (one element length) list:

```
x[0] (size 4) Unassigned
```

The next phase is starting with the DFS traversing backward in the graph. It traverses up in the control-flow until it hits the top, an already visited net or an if-/switch-statement. In case of an if or switch are separate DFS traversals started. Walking along the control-flow the algorithm stops at each ASSIGN or CALL box to gather information. It stores these possible entries into the Procedure Summary temporary on the node so when in the next phase the algorithm traverses forward it can immediately retrieve this information. In principle ASSIGN and CALL nodes are handled exactly the same. The only difference is that a CALL node consists of more than one assignment.

The next phase traverses the graph from sink to source. At the top net is the unassigned list stored or other information already computed, which is then merged with the information at the connecting box.

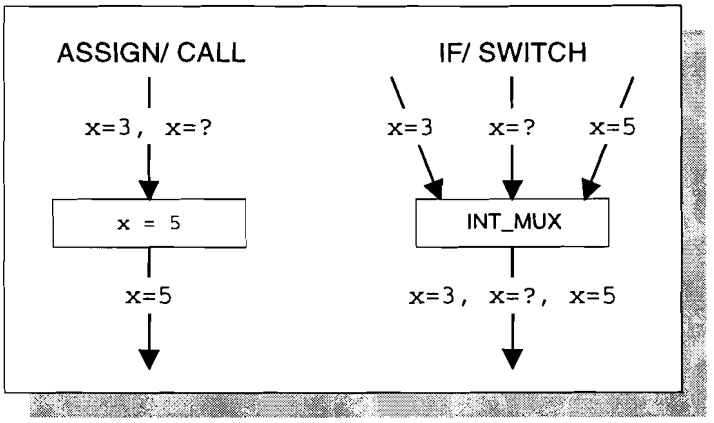


Figure 13: Combine and MuxCombine.

Two different merges exist; one for normal ASSIGN and CALL boxes and another one to handle if- and switch-statements see Figure 13. The merge on the left takes two input parameters, the Procedure Summary on the net above and the one stored on the box. An assignment later in the control overwrites previous assignments to that variable so the result of this combine will be **x=5**. However there may be other variables assigned, these may not be changed. Only the variables with the same name, index and size need to be overwritten.

The MuxCombine gathers the information for the MAY entries. Nothing is being overwritten only everything present at each branch into the if-/switch is joined. If for example on two (or more) branches the same variable is assigned, they all have to be copied into the Procedure Summary representing MAY assignments.

When the forward graph traversal finishes a new Procedure Summary is available. It does not mean that this procedure Summary is more accurate than the former one. Which one is most accurate is hard to figure out. The method used is first to check the size (=number of elements) of the Procedure Summary. It is only allowed for a Procedure Summary to grow thus comparing the new and old one the largest is stored in the CALL-graph. When they are of equal size each element is checked if it changed from the old version. Only the newest is stored relying on the fact that it is most accurate because it is computed with the newest information. The comparison with the old one is needed to return a change (**TRUE** or **FALSE**) value to Propagation. Propagation needs to know this in order to decide recomputation for depending procedures. The algorithm is shown in Code 19.

```
1:    CalcGlobalCFMMEffects()
2:    {
3:      EndNet = GetNetFromOutputPin();
4:      DFSControlFlow(EndNet);
5:    }
6:
7:    DFSControlFlow(Net)
8:    {
9:       if( Computed(Net) )
10:        return;
11:
12:      if( !IsControlFlow(Net) ||
13:          TopNet(Net) )
14:        return;
15:
16:      Box = GetBoxFromUsage(Net);
17:
```

```
18:       if ( is( box, INT_MUX ) )
19:       {
20:         ∀Branches
21:           DFSControlFlow(Branch);
22:
23:         MuxCombine();
24:       }
25:
26:       if ( is( box, ASSIGN ) )
27:         AssignComputation();
28:       if ( is( box, CALL ) )
29:         CALLComputation();
30:
31:       DFSControlFlow(GetNetFromBox(box));
32:
33:       Combine();
34:   }
```

Code 19: Algorithm for CalcGlobalCFMMEfeects.

The implementation has been tested on several cases and works without problems except for the following situation:

```
1:    int x;
2:    char *c;
3:
4:    void P(int cond)
5:    {
6:      if (cond)
7:        {
8:          c = (char*)&x;
9:
10:         x = 1;
11:         c[2] = 2;
12:       }
13:   }
```

Code 20: Problematic procedure for CalcGlobalCFMMEffects.

With the resulting Procedure Summary:

```
x[ 2 ] (size 1) 2
x[ 0 ] (size 4) 1
c[ 0 ] (size 4) Unknown
x[ 0 ] (size 4) Unassigned
x[ 2 ] (size 1) Unassigned
```

The type-casted pointer-array **c** has been found to actually be location **x**, which is perfectly OK. But assignments done to either **x** or a part of **x** are not. It is not noticed that **x**[2] (size 1) overwrites a part of **x**[0] (size 4) which thus changes its value. Considering the MAY/MUST information is everything in order but the precision is not. More information needs to be considered to be able to handle this problem.

Chapter 7. Pointers & Aliases

In programming languages with general-purpose pointer usage like C/C++, pointer aliasing is crucial in source code analysis. Many different pointer aliasing analysis techniques have been presented in the literature [AMME98], [CHOI93], [COOP89], [HAN97], [LAND--], [PAND--] and [ZHAN96]. They vary in cost, precision and produced aliasing information. Most literature talks only about compilers in which speed is crucial. Consequently the techniques are scaled to low precision in tradeoff for speed. However in the case of BEAM precision is important and speed is of lower priority. There are papers that have done research about speed/precision tradeoff [ZHAN98], so called combined analysis. Apart from speed, they also use a different kind of graph representation, which makes porting those algorithms into BEAM more difficult.

This chapter starts with a definition about aliases and alias analysis. The next section discusses the way it is done in most papers with their advantages and disadvantages. The last section starts with discussing the source language for BEAM's alias analysis. Followed with the limitations which BEAM will have using this "restricted" language. Then the algorithms are explained and methods are defined followed by a final discussion of the implementation.

7.1. Problem representation

To acquire a better accuracy, handling of aliases has to be incorporated in BEAM. With aliases more specific and especially more detailed information can be retained. A general definition of an alias is given as follows:

Definition: *An alias exists at a program point when two or more locations refer to the same location as a result of program execution to that program point. An Alias is represented by an unordered pair of locations, see 7.1.1. The order is unimportant since the alias relation is symmetric.*

Alias information will supply the symptom-analysis part in BEAM with alias information. If a certain location is changed BEAM will be able to detect which other locations also have been changed. The alias information will be stored in the summaries (and thus used for inter-procedural analysis) so BEAM will get more detailed information of locations being changed in different procedures.

7.1.1. Example alias

The following procedure shows an alias. It starts with setting the pointer **p** to **&x**. At this point ***p** and **x** are an alias. The program continues with assigning a value to **x**, this does not have any influence on the alias information. The last line in the procedure assigns **p** to **q**, which means that the address of **x** is copied into **q**.

```
1: void P()
2: {
3:    int *p, *q, x;
4:    p = &x;
5:    x = 1;
6:    q = p;
7: }
```

Code 21: Alias example.

The alias Procedure Summary algorithm should conclude for the assignment on line 4: `<*p, x>`, the first alias. In line 6 it should conclude alias `<*q, *p>`. Sometimes papers use `<q, p>` instead depending on r-value and l-value usage. The aliases are an equivalence relation (using `<*q, *p>`). Consequently the alias algorithm should figure out the `<*q, x>` alias. The last one involves good administration.

7.1.2. NP-Complete

Most data-flow problems have been proven to be NP-hard in the presence of aliasing. A more detailed problem, intra procedural aliasing in the presence of multiple level pointers, has also been proven to be NP-hard. One proof of flow-insensitive MAY-alias analysis being NP-hard is given in [HORW97]. Most other proofs are alike and are a reduction from the 3-SAT problem. The problem that we need to solve is inter procedural aliasing with multiple level pointers and recursive data structures (e.g., linked lists).

7.2. The methods found in papers

Most papers start with a definition of the (limited) source language they use. This is followed by a definition of the commonly used graph representation (different from the one that BEAM uses). Here are those restrictions discussed along with their algorithms and problems.

7.2.1. Their Source Language, limitations

Most alias research restricts itself on a subset of the C/C++ language. Actually not all research is being done on C/C++ only, Fortran is a common language too but mostly in only older papers. Most exclusions from the C language are pointers to functions, casting, union types, exception handling, set jump and long jump. Pointer arithmetic is allowed but arrays are treated as aggregates and assumed pointer arithmetic is only between array bounds. Concerning C++ they have the same restrictions for C analysis, except that most do handle type casting within class hierarchy. Also they cannot distinguish between different elements of arrays (i.e., a[i] is considered to be the same as *a).

7.2.2. Their ICFG

The used graph representation differs from the one used in BEAM. Programs are represented by *inter procedural control-flow graphs*, (ICFGs), e.g., presented in [LAND92]. Were BEAM uses a flow graph (combined control and data-flow) together with a CALL graph to connect the procedures (see section 3.3.2 and 4.1), an ICFG is a combination of both flow graph and CALL graph. The flow graphs they use consist only of control-flow information for each procedure, with calls connected to the procedure they invoke.

A formal description can be given as follows: an ICFG is a triple (N, E, ρ) where N is the set of nodes, E is the set of edges and ρ is the entry node for main. N contains a node for each (simple) statement in the program, an entry and exit node for each function, and a call and return node for each procedure call site. An intra procedural edge into a call node represents the execution flow into an call site, while an intra procedural edge out of a return node represents control-flow from a call site once the called function has returned. There are two extra inter procedural edges for each call site: one from the call node to the entry node of the invoked procedure, and one from the exit node of the procedure to the return node of the call site. For an example of an ICFG see Figure 14.

7.2.3. Advantages and disadvantages of the ICFG

Pro:

- direct propagation of gathered information (may explode)
- knowing in which environment the call is
- no graph explosion, each procedure is only once there

Con:

- relatively little information in the graph (1 node = 1 statement)
- context problem, not knowing which procedure was the calling one
- low precision

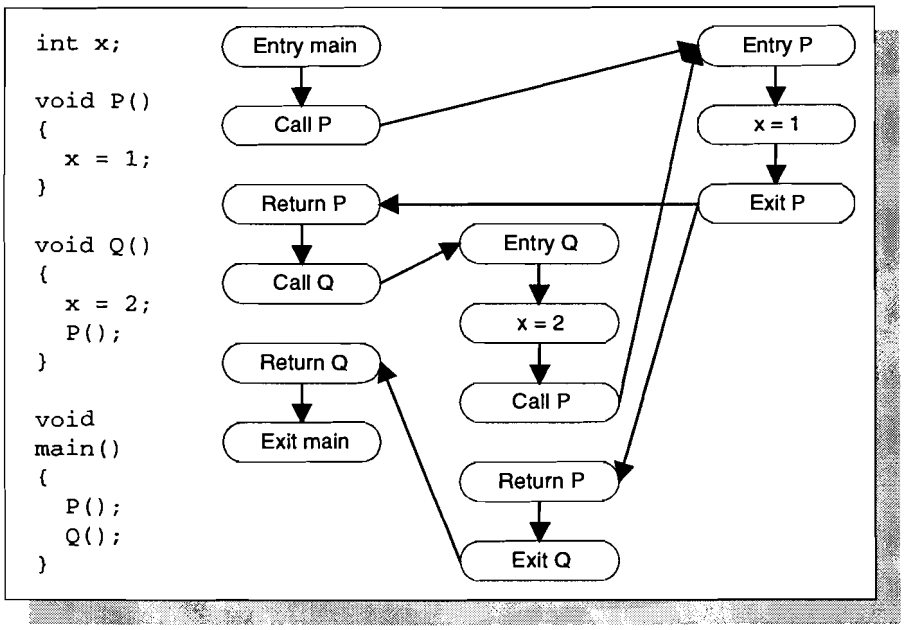


Figure 14: Example of an ICFG.

7.3. The principle for BEAM

7.3.1. The Source Language

The restrictions on the source language that alias analysis for BEAM is able to handle is hard to describe. The graphs used by BEAM do not have a 1-1 relation with the statements, like in an ICFG. The data-flow analysis for aliases will need to figure out each pointer assignment. Sometimes this becomes too complicated. The algorithm will need to stop and decide "Unknown" for those pointers. Only when the algorithm is implemented and tested it will be possible to say more about what the restrictions on the source language are.

Problems with algorithms found in papers will not be a problem for BEAM. For example pointers to functions are visible like any other pointer. Casting will also not be a problem, see Code 20 where the casting is handled correctly. BEAM's precision on handling arrays will provide the necessary information for pointer arithmetic. One problem on classes and virtual functions is present. Currently only the root-virtual function is available for analysis and not the derived ones. Pointers to these kinds of members consequently will not be analyzable.

Precision in case of recursive data structures is a general problem. The papers [JONE82] and [RUGG88] talk specifically about this problem. The solution for BEAM's Procedure Summaries is relative straightforward. When finding these recursive data structures a method is used called k-limiting. k-limiting means to compute recursive structures until depth k. Of course there is a statistically known best value for k. This value is 2 and is determined in the compiler environment depending on speed and precision. But since BEAM's analysis is more thorough some "playing" around will be in order to determine the correct value for k.

7.3.2. The data-flow analysis

Each box in the data-flow needs a rule on what the input/output relation is. The following is explained with the use of example graph in Figure 31. Information is stored/retrieved with the 4 fields in the Procedure Summary information node; name, index, size and value. Only those needed to compute the example graph are explained. "#" means not relevant information and "*" means any value possible.

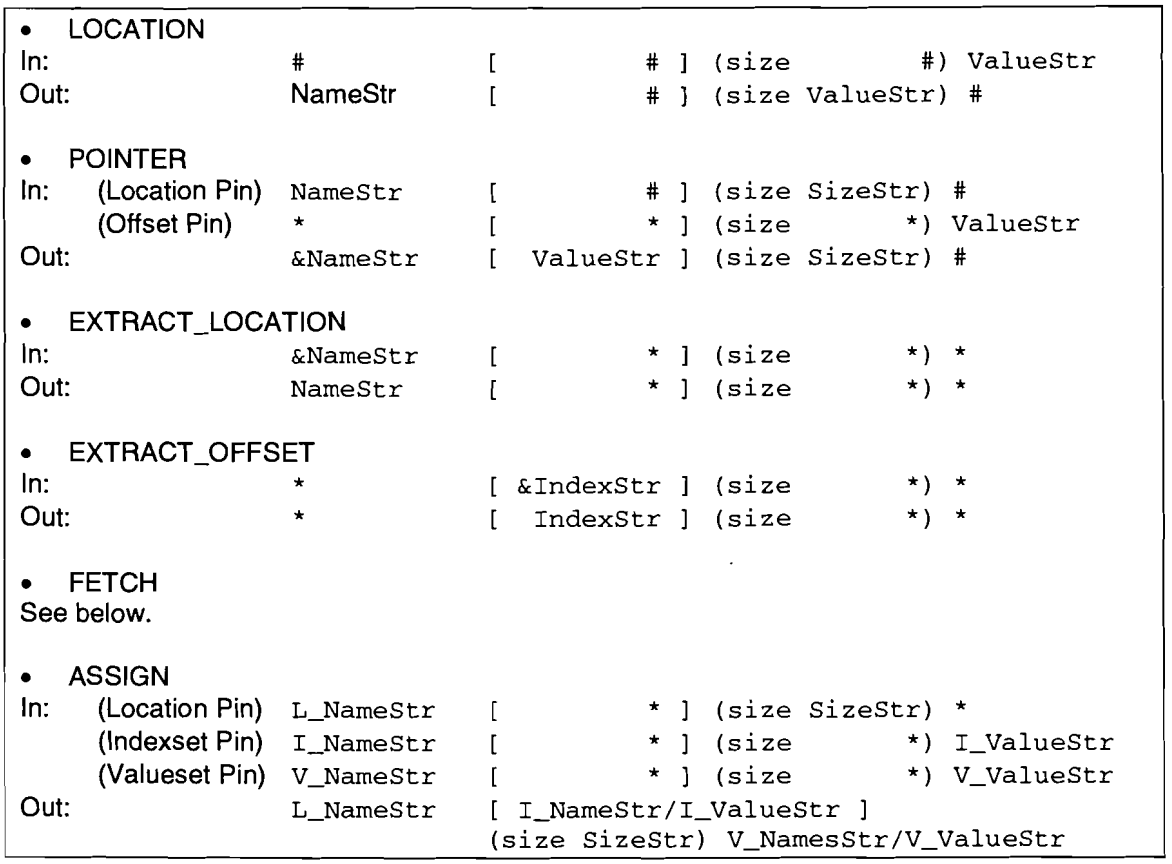


Figure 15: List of rules for alias/pointer data-flow analysis.

The FETCH node is a node that needs special treatment. This node searches backward in the control-flow if it can find an assignment to the given location input at index. This will be either a value or another variable (possibly with "&"). Sometimes FETCH will not be able to find this due to complexity, k-limiting, branches etc. In such cases the value "Unknown" will be used. The example graph with the computed data-flow information is shown in Figure 32.

7.3.3. The aliasing graph traversing algorithm

In general the algorithm traverses forward through the CFG. It starts at the top, walking through the CFG until it sees an assign node. From hereon it tries to figure out the values of the pins on this assign box. The algorithm works on the complete flow graph because the values depend on the data-flow and not only the control-flow. When finding out the value of a pin, the information is retrieved "on demand". A new DFS algorithm is started for each assign box input pin. It traverses the graphs backward and tries to figure out the needed data-flow information.

Paths in a program graph can be very complicated due to if/switch statements and goto's. Double (or more) computation of boxes should be prevented so the information computed the first time is saved in a keyword on the corresponding net. When a follow-up computation needs the same information it does not have to be recomputed because it is already available on the net. This saves computation time plus memory.

The algorithm AliasDFS starts with the root net of the procedure graph. It starts traversing the CFG going forward. Recursively it calls itself and stops until it sees a proto-pin or an INT_MUX node from which not all input nets have been computed. If all input nets from an INT_MUX node have been computed the algorithm continues along the CF. When it hits an ASSIGN node it knows it has to process some information so it calls the function CompAssign. Like with the globals, ASSIGN nodes and CALL nodes are handled the same. For simplicity is chosen to only show the ASSIGN nodes.

```
1:    AliasDFS(NET Net)
2:    {
3:      ∀Sink Pins on Net
4:      {
5:        if (Pin == ProtoPin)
6:          continue;
7:
8:        Box = get_box_from_pin(Pin);
9:        if (Box == INT_MUX)
10:         {
11:           MarkComputed(Box);
12:             if(!QueryComputed(Box));
13:               continue;
14:         }
15:       if (Box == ASSIGN)
16:         CompAssign(Box);
17:
18:       AliasDFS(NetAlongCF(Box));
19:     }
20:   }
```

Code 22: The alias graph traversing algorithm.

The CompAssign procedure gathers information about all the input pins (Location, Offset and Value). If there is no information already available at an input pin it starts a recursive procedure to find out the value(s) for that specific net. This procedure, ComputeAbove, first goes backward into the graph to a sink node and tries to figure out the values on each net. Tries, because it is not always possible to figure out the exact values.

After it checked if it was a valid net, pin and box, it calls (depending on the type of box) a procedure which is able to calculate the properties of the box. As mentioned these properties will be stored on the net for future use.

Crucial is the way these boxes convert their input information to the output information. This has to be specified for each box plus it has to be defined on how to combine this information if two (or more) nets are input to an INT_MUX box.

```
1:    CompAssign(NET Net)
2:    {
3:      if (!Net)
4:        return;
5:
6:      Pin = SourcePin(Net);
7:
8:      if (Pin == ProtoPin)
9:        continue;
10:
11:     Box = get_box_from_pin(Pin);
12:
13:     switch(Box)
14:       {
15:         case: ... break;
16:         case: ... break;
17:         default: break;
18:       }
19:   }
```

Code 23: CompAssign algorithm.

7.3.4. Conclusions

The algorithms for aliasing/pointers have been implemented and tested so far that the example graph can be correctly computed. Problems occur when branches are involved. For example in the following code fragment:

```
1:    int *p, x, y, A[10];
2:
3:    ...
4:
5:    if(cond)
6:      p = &x;
7:    else
8:      p = &y;
9:
10:   A[*p] = 2
```

Figure 16: Problematic branches.

Presently the algorithm computes this Procedure Summary (concerning **A** only):

```
A[ x ] (size 4) Unassigned
A[ x ] (size 4) 2
A[ y ] (size 4) Unassigned
A[ y ] (size 4) 2
```

The real procedure Summary of course needs to be the following:

```
A[ x ] (size 4) 2
A[ y ] (size 4) 2
```

Until now, no straightforward solution has been found to solve this problem. Only extensive administration will most likely provide the correct Procedure Summary.

Not enough information has yet been implemented in the alias algorithms. Not all pointers can be handled, k-limiting is not used and also finding the real aliases is still not possible. The core of the aliasing is implemented but a lot of administration issues still stand.

Chapter 8. Conclusions and future work

8.1. Conclusions

This thesis first explains BEAM's approach to static source code analysis. Several analysis methods are described and an in depth description is given of the precision needed for finding different error symptoms. The achievements when applying BEAM to large software projects are measured and discussed along with the symptoms of errors that BEAM is able to find. Since there are more companies focusing on static analysis tools a comparison is done. Advantages of these tools versus BEAM's are also discussed in the first part of the report.

This is followed with a discussion about path lengths needed for analysis and how to shorten these paths. The solution for this problem is given with Procedure Summaries. They will be used as an information-providing platform for the actual symptom finding analysis performed by BEAM. We discussed possibly useful information in the Procedure Summaries. Before explaining the implementation a description is given about the graph representation used by BEAM. This is a given source on which all the algorithms need to be built.

Prior to explaining the framework in which the Procedure Summaries will be used in BEAM, the propagation algorithm is considered. This algorithm controls the decision of which procedures need to be recomputed. It is fast in building a list in which the dependencies of procedures are shown. This so-called post-order list provides the algorithm with the order in which the Procedure Summaries need to be computed. Loops and recursive procedures are handled correctly. The algorithm uses Time-Stamps to determine necessary recomputation and in addition it deletes graphs that are no longer needed from memory, because this was found to be a problem. There are pathological problematic procedures for which this algorithm fails. This is solved with a restriction on Propagation by limiting the number of recomputations.

BEAM is able to use the Procedure Summaries with only little change in its own code. This is accomplished by building a flexible framework in which different computations can be easily added. Another possibility is given but this involves more graph loading/saving from disk, which takes much more time than the disadvantage of the current implementation in a recomputation every now and then.

The first implementation of the Procedure Summaries discussed is the statistics. These statistics gather information about procedures; based on this more demand driven analysis might be performed. The statistical results of comparing various programs are given. Unfortunately no demand driven analysis will be possible based on only this information. Differences between programs were either minor or irrelevant. The second implementation is the dereferencing of pointer parameters. Analysis on pointer dereferencing is rather straightforward, however has the potential to be of great relevance. Dereferencing resulting in garbage is never intended and since this type of Procedure Summary is actually used by BEAM, its results can be reported. On a logic synthesis system (about half a million lines of code) twelve extra errors have been found which BEAM would not have found otherwise. This clearly indicates the power of the Procedure Summaries and also their significance for BEAM. More of these results are expected when BEAM starts using the other Procedure Summaries. The globals have been implemented next. Starting with a relative easy implementation we progressed to a detailed and more exhaustive analysis for searching assignments to global variables in procedures. The thorough analysis of the control-flow resulted in precise global Summaries. The included MAY/MUST information will be very useful to BEAM when it has to decide if a procedure needs to be expanded inline. The final Procedure Summaries are about aliases and pointers. In a language like C++ pointers are often found to be hazardous. The algorithm described will be able to find these aliases and group them into a Procedure Summary. Precision like MAY/MUST information is archived with thorough control-flow and data-flow analysis. Unfortunately no time was available to implement and test the aliasing to a completely working Procedure Summary.

8.2. *Future work*

The implemented Procedure Summary environment is only a start for BEAM to do more precise analysis. It is an information-providing platform for the analysis phase. It would be very interesting to (re-) implement the algorithms for symptom finding concerning the use of the Procedure Summaries. Currently this has only been done for pointer dereferencing. Since this already resulted in a high number of new symptoms it is also expected for the other Procedure Summaries.

Furthermore BEAM is a tool under construction. It is only able to find symptoms of errors and often it is hard to decide if such a symptom is actually a bug, and if so what the cause is. The messages include a path leading to a symptom but in large programs it is often "hard work". Only testing BEAM very often (also on itself) and getting feedback on the error-reports will improve the correctness of the analysis done by BEAM. Presently the error messages are printed as plain text. A graphical user interface would be useful with a direct link to the actual code. The error messages would then interact with the lines in the source code such that users are directly able to see what happens on each line. A debugger "look" is thought of as being most applicable.

Glossary and abbreviations

This glossary lists and explains the terminology used in this thesis and in BEAM. Most words and terms appear frequently in other papers/books concerning program analysis, data-flow analysis and compiler theory.

Bottom-up Backward graph traversal.

CF(G) Control-flow (Graph), the graph in which executable paths can be found. Such a path on itself is called control-flow i.e., "there where the control-flows".

DF(G) Data-flow (Graph), the graph providing information for the control-flow graph. BEAM uses both graphs combined into one.

ICFG Inter procedural Control-flow Graph explained in section 7.2.2.

Inter procedural Analysis focussing on only one graph/procedure.

Intra procedural Analysis focussing on graphs and procedures including their connection and dependence.

MAY alias This is only used in some papers, e.g., [LAND--]. It does not have a direct relation with MAY/MUST information in the Procedure Summaries described in this report. In their analysis MAY aliasing is faster and easier to compute. They do not use high precision to figure those out. In compiler theory this often provides satisfying results.

Multiple level When performing alias analysis not only single level pointers (`*p`) are of interest but also multiple level pointers (`**p` or derivations).

MUST alias Like MAY aliases but more detailed; takes more time to compute.

PS Procedure Summary, a list of relevant items that might influence program behavior.

Sink node Nodes in a graph with only incoming edges (root nodes in trees).

Source node Nodes in a graph with only outgoing edges (leaf nodes in trees).

Top-down Forward graph traversal.

References

Throughout the text references like [LAND--] are given. This reference indicates all papers written by author "LAND".

[AMME98] Amme, W. and E. Zehendner
DATA DEPENDENCE ANALYSIS IN PROGRAMS WITH POINTERS.
Parallel Computing, Vol. 24 (1998), Iss. 3-4, p. 505-25.

[CALL86] Callahan D. and K. D. Cooper, K. Kennedy, L. Torczon
INTERPROCEDURAL CONSTANT PROPAGATION.
In: Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, June 1986.

[CALL88] Callahan D.
THE PROGRAM SUMMARY GRAPH AND FLOW-SENSITIVE INTERPROCEDURAL DATA FLOW ANALYSIS.
In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlantam Georgia, June 22-24, 1988.

[CANF98] Canfora G. and A. Cimitile, U. Decarlini, A. Delucia
AN EXTENSIBLE SYSTEM FOR SOURCE CODE ANALYSIS.
IEEE Transactions on Software Engineering, Vol. 24 (1998), Iss. 9, p 721-740.

[CHEN97] Chen, Y. F. R. and E. R. Gansner, E. Koutsofios
A C++ DATA MODEL SUPPORTING REACHABILITY ANALYSIS AND DEAD CODE DETECTION.
Software Engineering Notes, Vol. 22 (1997), Iss. 6, p. 414-31.

[CHOI91] Choi J. D. and R. Cytron, J. Ferrante
AUTOMATIC CONSTRUCTION OF SPARCE DATA FLOW EVALUATION GRAPHS.
Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, January 1991.

[CHOI93] Choi J. D. and M. Burke, P. Carini
EFFICIENT FLOW-SENSITIVE INTERPROCEDURAL COMPUTATION OF POINTER-INDUCED ALIASES AND SIDE EFFECTS.
Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (1993), p. 232-45.

[COOP85] Cooper K. D. and K. Kennedy
EFFICIENT COMPUTATION OF FLOW INSENSITIVE INTERPROCEDURAL SUMMARY INFORMATION.
In: Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 19 (July 1985), No. 6.

[COOP88] Cooper K. D. and K. Kennedy
INTERPROCEDURAL SIDE-EFFECT ANALYSIS IN LINEAR TIME.
In: Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation, Atlanta, GA, June 1988.

[COOP89] Cooper K. D. and K. Kennedy
FAST INTERPROCEDURAL ALIAS ANALYSIS.
Sixteenth ACM Symposium on Principles of Programming Languages, Austin, Texas, January 11-13 1989, p. 49-59.

[DUES95] Duesterwald E. and R. Gupta, M. L. Soffa
DEMAND-DRIVEN COMPUTATION OF INTERPROCEDURAL DATA FLOW.
Conference Record of POPL 1995: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995, p. 37-48.

[DUES97] Duesterwald E. and R. Gupta, M. L. Soffa
A PRACTICAL FRAMEWORK FOR DEMAND-DRIVEN INTERPROCEDURAL DATA-FLOW ANALYSIS.
ACM Transactions on Programming Languages and Systems, Vol. 19 (1997), Iss. 6, p. 992-1030.

[HAN97] Han, D.S. and T. Tsuda
NON-GRAPH BASED APPROACH ON THE ANALYSIS OF POINTERS AND STRUCTURES.
IEICE Transactions on Information and Systems, Vol. E80-D (1997), Iss. 4, p. 480-8.

[HORW97] Horwitz, S.
PRECISE FLOW-INSENSITIVE MAY-ALIAS ANALYSIS IS NP-HARD.
ACM Transactions on Programming Languages and Systems, Vol. 19 (1997), Iss. 1, p. 1-6.

[JONE82] Jones N. D. and S. S. Muchnick
A FLEXIBLE APPROACH TO INTERPROCEDURAL DATA FLOW ANALYSIS AND PROGRAMS WITH RECURSIVE DATA STRUCTURES.
Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, January 1982.

[JOHN93] Johnson R. and K. Pingali
DEPENDENCE-BASED PROGRAM ANALYSIS.
SIGPLAN Notices, Vol. 28 (1993), Iss. 6, p. 78-89.

[LAND92] Landi, W. and B. G. Ryder
A SAFE APPROXIMATION ALGORITHM FOR INTERPROCEDURAL POINTER ALIASING.
In: Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation, June 1992, p. 235-248.

[LAND93] Landi, W. and B. G. Ryder, S. Zhang
INTERPROCEDURAL MODIFICATION SIDE EFFECT ANALYSIS WITH POINTER ALIASING.
In: Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation, Pages 56-67, June 1993.

[LAND98] Landi, W. and B. G. Ryder, P. Stocks, S. Zhang, R. Altucher
A SCHEMA FOR INTERPROCEDURAL SIDE EFFECT ANALYSIS WITH POINTER ALIASING.
Department of Computer Science, Rutgers University, May 1998, Number DCS-TR-336.

[MEYE92] Meyers, S.
EFFECTIVE C++.
MA: Addison-Wesley, 1992.

[MEYE96] Meyers, S.
MORE EFFECTIVE C++.
MA: Addison-Wesley, 1996

[MEYE97] Meyers, S. and M. Klaus
EXAMINING C++ PROGRAM ANALYZERS.
Dr. Dobb's Journal, Vol. 22 (1997), Iss. 2, p. 68,702,74-5,87.

[PAND94] Pande, H. D. and B. G. Ryder
STATIC TYPE DETERMINATION AND ALIASING FOR C++.
Laboratory of Computer Science Research Technical Report, December 1994, Number LCSR-TR-236.

[PAND95] Pande H. and B. G. Ryder
STATIC TYPE DETERMINATION AND ALIASING FOR C++.
Laboratory of Computer Science Research Technical Report, October 1995,
Number LCSR-TR-250-A.

[REPS95] Reps T. and S. Horwitz, M. Sagiv
PRECISE INTERPROCEDURAL DATAFLOW ANALYSIS VIA GRAPH REACHABILITY.
Conference Record of POPL 1995, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995, p. 49-61.

[RUGG88] Ruggieri C. and T. P. Murtagh
LIFETIME ANALYSIS OF DYNAMICALLY ALLOCATED OBJECTS.
Conference Record of Fifteenth ACM Symposium on Principles of Programming Languages, 1988.

[SAGI98] Sagiv M. and N. Francez, M. Rodeh, R. Wilhelm
A LOGIC-BASED APPROACH TO PROGRAM FLOW-ANALYSIS.
Acta Informatica, Vol. 35 (1998), Iss. 6, p. 457-504 (ZU631)

[SCZE97] Sczepansky A.
EFFICIENT SOFTWARE DEVELOPMENT USING STATIC TEST TOOLS. *(German)*
Elektronik, Vol. 46 (1997), Iss. 3, p. 78-81.

[STOC98] Stocks, P.A. and B. G. Ryder, W. A. Landi, S. Zhang
COMPARING FLOW- AND CONTEXT-SENSITIVITY ON THE MODIFICATION-SIDE-EFFECTS PROBLEM.
In: Proceedings of the International Symposium on Software Testing and Analysis, March 1998, p. 21-31.

[TARJ81] Tarjan R. E.
FAST ALGORITHMS FOR SOLVING PATH PROBLEMS.
Journal of the Association for Computing Machinery, Vol. 28(3) (1981), p.594-614.

[ZHAN96] Zhang S. and B. G. Ryder, W. Landi
PROGRAM DECOMPOSITION FOR POINTER ALIASING: A STEP TOWARDS PRACTICAL ANALYSES.
In: Proceedings of the 4th Symposium on the Foundations of Software Engineering, October 1996.

[ZHAN98] Zhang S. and B. G. Ryder, W. A. Landi
EXPERIMENTS WITH COMBINED ANALYSIS FOR POINTER ALIASING.
In: Proceedings of Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), June 1998.

Appendix A. Static Analysis Tools

Listed are contact addresses of tools described in section 2.4. Intrinsa has been added to the list because they also have an interesting tool available. At the end of the list are some interesting Internet links summarized. These links are related with source code analysis and have been found useful in either general sense or helpful in giving more specific detail.

A.1. Companies

Abraxas Software
5530 SW Kelly Avenue
Portland, OR 97201
503-244-5253
http://www.abxsoft.com/

Centerline Software
10 Fawcett Street
Cambridge, MA 02138-1110
617-498-3000
http://www.centerline.com/

Gimpel Software
3207 Hogarth Lane
Collegeville, PA 19426
610-584-4261
http://www.gimpel.com/

Hewlett-Packard
19410 Homestead Road
Cupertino, CA 95014-0604
408-725-8900
http://www.hp.com/sesd/CA/

Intrinsa Corporate Headquarters
444 Castro Street, Suite 130
Mountain View, California 94041
650-390-8600
http://www.intrinsa.com/

ParaSoft Corp.
2031 South Myrtle Avenue
Monrovia, CA 91016
818-305-0041
http://www.parasoft.com/

Productivity Through Software Inc.
555 Bryant, Suite 555
Palo Alto, CA 94301
415-934-3200
http://www.pts.co.uk/

Programming Research Ltd.
1/11 Molesey Road, Hersham
Surry KT12 4RH, UK
+44-1932-88 80 80
http://ww.prqa.co.uk/

Rational Software Corp.
2800 San Tomas Expressway
Santa Clara, CA 95051-0951
408-496-3600
http://www.rational.com/

A.2. Internet links

http://www.accu.org/
Accu is an association of C/C++ and Java users. They include professional programmers, the suppliers of compilers, those just interested in the languages, and anyone seeking to improve their programming skills. Based in the UK, with members in the US, mainland Europe, Russia, Middle East and Australia. This is also an interesting link concerning book reviews on the C/C++ and Java topic.

http://www.prolangs.rutgers.edu/
B. Ryder has been a guest at IBM Research and she presented her research at PROLANGS. The PROLANGS research group was created after a seminar given by B. Ryder in 1988. Since then the group has met weekly as a conference/journal paper reading/discussion group. The topics covered by the group are usually related to Compile-time Analysis, although other subjects in Programming Languages and Compilers are also covered. Most PROLANGS members are, or have been graduate students or are somehow related to the Rutgers Department of Computer Science.

http://www.win.tue.nl/cs/ooti/uk/symposium/
OOTI is a Dutch acronym for Ontwerpersopleiding Technische Informatica which is the Dutch term designating the Software Technology program of the Stan Ackermans Institute at the Eindhoven University of Technology. Studies at the SAI/Software Technology program are aimed at software engineers and architects. Although the theoretical basis is very important, the curriculum focuses on practical methods, techniques and tools.

Appendix B. List of rules

The following is a list of rules on which the programs in section 2.4 are tested. This list should give the reader an idea of the different error categories and more specific the diversity of errors.

General

1. Use const instead of #define for constant at global and file scope
2. Use new-style casts instead of C-style casts
3. Don't treat a pointer to derived[] as a pointer to base[]

Use of new and delete

4. Use the same form for calls to new and delete.(In general, this calls for dynamic analysis which BEAM does not perform, but static analysis can catch some special cases; calls to new in constructors and to delete in destructors, for example).
5. When the result of a new expression in a constructor is stored in a dumb pointer class member, make sure delete is called on that member in the destructor
6. Avoid hiding the default signature for operator new and operator delete.

Constructors/destructors/assignment

7. Declare a copy constructor for each class declaring a pointer data member
8. Declare an assignment operator for each class declaring a pointer data-member.
9. Initialize each class data member via the member initialization list
10. List members in a member initialization list in an order consistent with the order in which they are actually initialized
11. Make destructors virtual in base classes
12. Have the definition of operator= return a reference to *this
13. Assign to every local data member inside operator=
14. Call a base class operator= from a derived class operator=
15. Use the member initialization list to ensure that a base class copy constructor is called from a derived class copy constructor
16. Do not call virtual functions in constructors or destructors

Design

17. Use nonmember functions for binary operations like +-/* when a class has a converting constructor
18. Avoid public data members
19. Use pass-by-ref-to-const instead of pass-by-value when both are valid.
20. Have operators like +-/* return an object, not a reference
21. And make those return values const.
22. Do not overload on a pointer and an integer
23. Make non-leaf classes abstract
24. Avoid gratuitous use of virtual inheritance; that is, make sure there are at least two inheritance paths to each virtual base class

Implementation

25. Do not return pointers/references to internal data structures unless they are pointers/references to const
26. Never define a static variable inside a nonmember inline function unless the function is declared extern.
27. Avoid use of "..." in function parameter lists

Inheritance

28. Do not redefine an inherited non-virtual function
29. Do not redefine an inherited default parameter value

Operators

30. Avoid use of user-defined conversion operators
31. Do not overload &&,||, or, etc.
32. Make sure operations ++ and -- have the correct return type
33. Use prefix ++ and -- when the result of the increment and decrement expression is unused
34. Declare operator= if you declare binary operator(e.g., declare += if you declare + etc.)

Exceptions

35. Prevent exceptions from leaving destructors
36. Catch exceptions by reference

Appendix C. Boxes & Pins

The result of the first computational step in BEAM is a valid CF/DF-Graph. This graph uses boxes (e.g., nodes) which represent certain C++ instructions & operations. In this section the important boxes are described, boxes where something is computed or boxes who are needed for a computation. Each box is described on its input-pin(s) and output-pin(s).

Note: not all the possible boxes in the generated graph are described. Only those needed for the Procedure Summary computation.

C.1. Important definition boxes

C.1.1. Primary boxes

- LITERAL_INT

Inputs: (0) None
Outputs: (1) One output-pin representing the value of the box, for example "*N4*" where "*N*" stands for number and "*4*" the value.

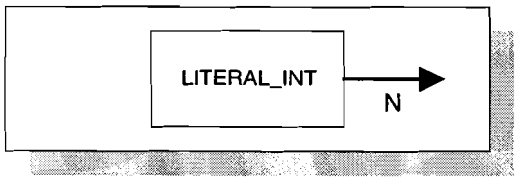


Figure 17: LITERAL_INT box.

- LOCATION

Inputs: (1) One input-pin indicating the size (in bytes) of the location.
Outputs: (1) One output-pin, which shows the location name. The location can for example be local (*L*) or global (*G*).

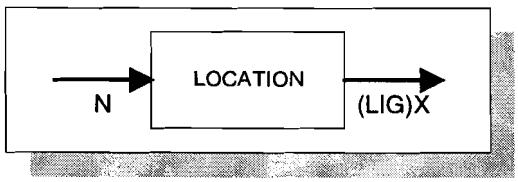


Figure 18: LOCATION box.

- FUNCTION

Inputs: (0) None.
Outputs: (1) Giving the name of the function.

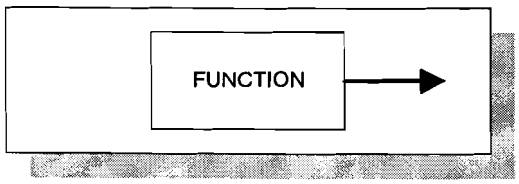


Figure 19: FUNCTION box.

- INT_CONST

Inputs: (1) The input-pin with a net from an integer value.
Outputs: (1) The output-pin represents a expression of type:

$$aX + b$$

Where X is the input, a an integer keyword on the input-pin and b is an integer keyword on the output-pin.

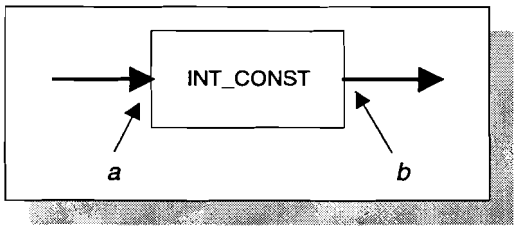


Figure 20: INT_CONST box.

C.1.2. Pointer related boxes

- POINTER

Inputs: (2) First the Location-pin which derives the location to be addressed. Second the Offset-pin to determine the offset of the given location.
Outputs: (1) One output-pin, which gives the address of the given location at given offset.

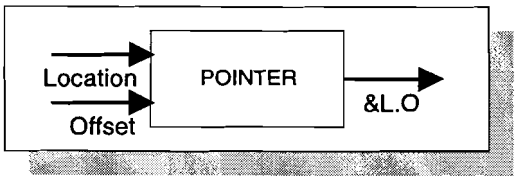


Figure 21: POINTER box.

Note: the POINTER box is the inverse of the EXTRACT_LOCATION/_OFFSET box.

- FETCH

Inputs: (3) First a Location pin, this pin has a keyword connected to it which holds the size (amount of bytes) to be fetched. The second pin is a graph pin that is an input from the CF. The last pin is an index-pin to indicate the index in the given location.
Outputs: (1) The output pin gives the fetched bytes from location-pin at index-pin and with length, size-keyword.

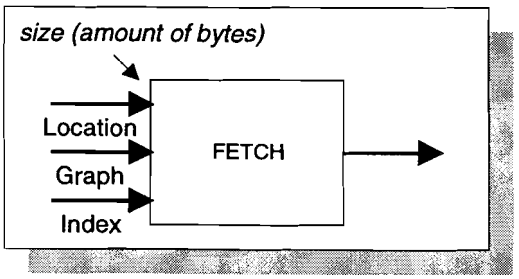


Figure 22: FETCH box.

- EXTRACT_LOCATION

Inputs: (1) Input-pin, a net connected from a valid pointer-location.
Outputs: (1) The output-pin gives the location of the pointer-location-input.

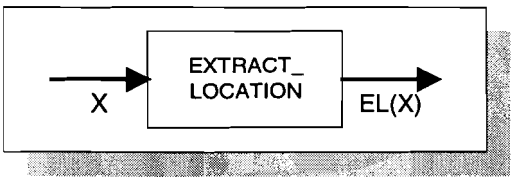


Figure 23: EXTRACT_LOCATION box.

- EXTRACT_OFFSET

Inputs: (1) Input-pin, a net connected from a valid pointer-location.
Outputs: (1) The output-pin gives the offset of the pointer-location-input.

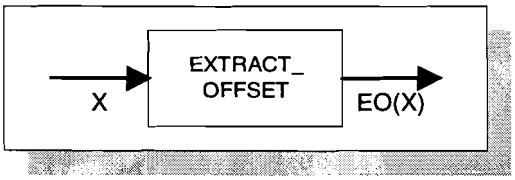


Figure 24: EXTRACT_OFFSET box.

Note: the EXTRACT_LOCATION & EXTRACT_OFFSET boxes are the inverse of the POINTER box. The two boxes are not combined because this will lead to a box with two output-pins. In general boxes with multiple output-pins are harder to simplify than single output-pin boxes. For this reason EXTRACT has been split up into two separate boxes.

C.1.3. Assignment boxes

- ASSIGN

Inputs: (4) The first input is the LHS-Location of the assignment, this pin has a keyword attached which indicates the element-size. The Value-pin is next which represents the RHS of the assignment. For the CF there is a third graph-pin and the last pin is the LHS-Index.
Outputs: (1) The output of the ASSIGN box is a pin with an expression as follows:

Location.Index(elementsize) = Value(elementsize)

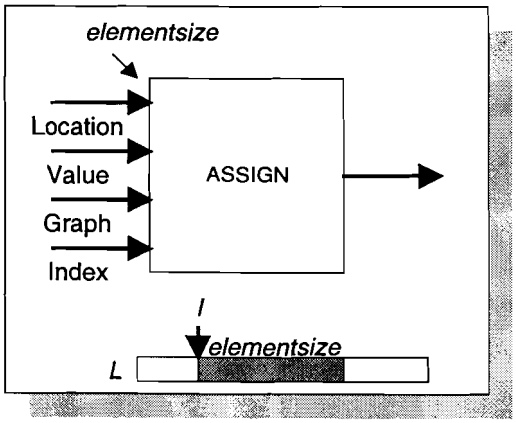


Figure 25: ASSIGN box.

- COPY_MEM

Inputs: (6) LHS Location and Offset-pins. RHS From-Location and From-Offset pins. A separate Size-pin is supplied to indicate the number of bytes to copy. The last pin is the CF-pin.

Outputs: (1) Outputs this expression:

Location.Offset(Size) = FromLocation.FromOffset(Size)

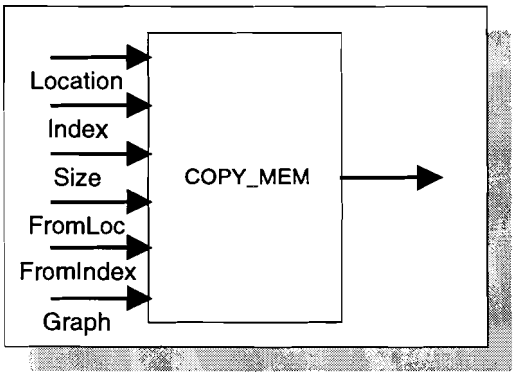


Figure 26: COPY_MEM box.

Note: the ASSIGN box is most commonly used, however in more complicated mem_copy and realloc cases, the COPY_MEM box is used. The reason for this is that the size (which would normally be a keyword on the Location input-pin) is not a previously known value.

- CALL

Inputs: (2/more) An input from the CF, second an input supplying the function address. More input-pins may be present if the function has parameters.

Outputs: (1/2) One (always there) output just continues the CF. Another output-pin is optional and may be supplied if the called function returns a value.

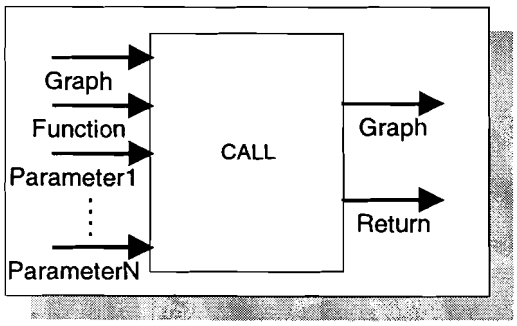


Figure 27: CALL box.

Note: the CALL box is used for both function calls and loop calls. Virtual functions are not yet supported because it is hard to figure out which function is being called, the default virtual function is taken instead.

C.1.4. if/switch boxes

- INT_MUX

Inputs: (3/more) The first input-pin represents the condition under which the other input-pins are chosen. There are two possibilities for the other input-pins. First, the input-pins are connected with edges from the CF. This is the case after an if or switch

statement. A second possibility is if the edges are from the DF. For example with a statement like:

(a ? 3 : 5)

will result in only DF edges/nodes. The input-pins will be LITERAL_INT boxes with value 3 and 5.

All input-pins will have a keyword attached, which represents a set of conditions under which input is taken.

Outputs: (1) Will be either the continuation with a CF-arc or a DF-arc representing the decided MUX-input.

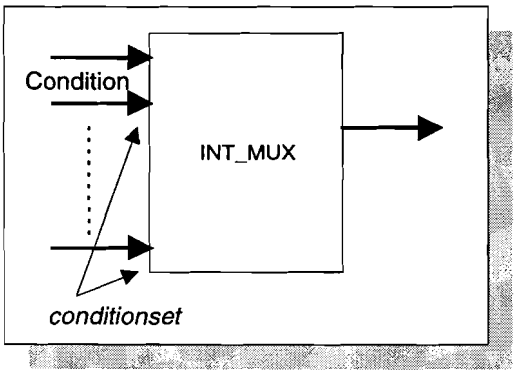


Figure 28: INT_MUX box.

Note: in optimized cases the input-pins on the INT_MUX box may have been reduced to one.

C.2. Important proto-box pins

- CF_PIN

Outputs: The CF starting point in the Graph (the CF_PIN is always present).

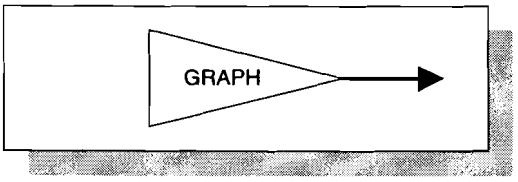


Figure 29: GRAPH pin.

- PARAMETER_PIN

Outputs: A parameter input of the function or loop, simply numbered from 1..N, with N as many as needed (there may be zero or more PARAMETER_PINS).

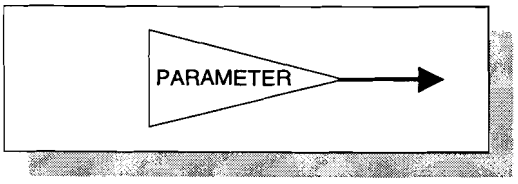


Figure 30: PARAMETER pin.

C.3. Complete list of all the possible boxes

This is a list of all possible boxes that may be generated by BEAM. These boxes may thus be encountered in the Graph.

Possible boxes		
• INT_ADD	• BITWISE_NOT	• INITIAL_RAM
• INT_CONST	• SHIFT	• LITERAL_INT
• ADD_FLOAT	• CALL	• LITERAL_INF
• MULT_INT	• IDENT	• LITERAL_FLOAT
• MULT_FLOAT	• CONVERT	• LITERAL_STRING
• MOD	• SUBRANGE	• POS_FLOAT
• DIVIDE_INT	• ASSIGNED_BEFORE	• POINTER
• DIVIDE_FLOAT	• EQUAL_LOCATION	• INT_MUX
• ASSIGN	• ERROR_DEF	• ASSERT
• COPY_MEM	• EXTRACT_LOCATION	• LOCATION
• MEMORY_OP	• EXTRACT_OFFSET	• DUMMY
• BITWISE_AND	• EXTRACT_SIZE	• LINE_NUMBER
• BITWISE_OR	• FETCH	• SOME_VALUE
• BITWISE_XOR	• FUNCTION	

Table 6: List of all possible boxes in a BEAM procedure graph.

Appendix D. Example graph

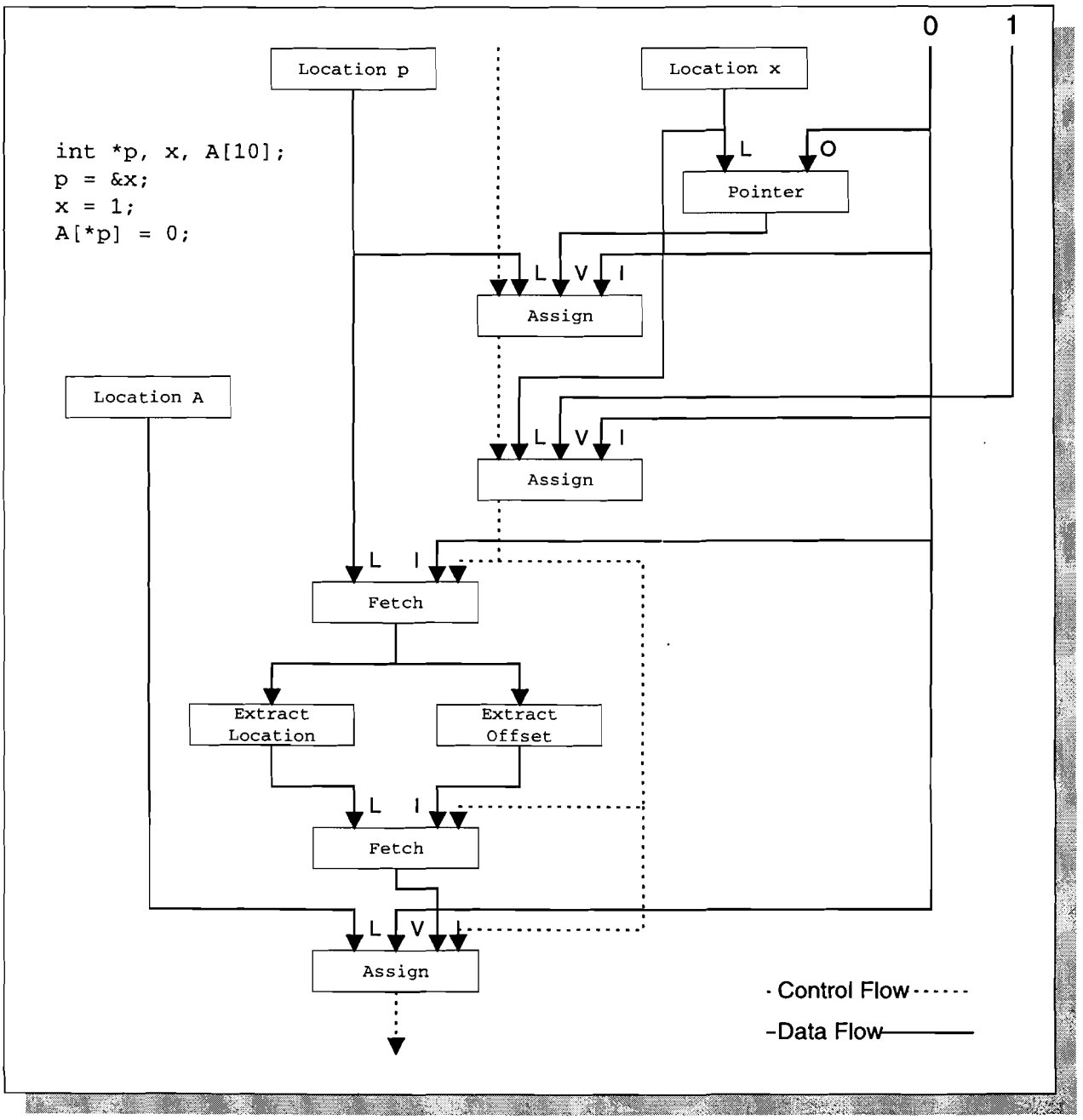


Figure 31: Example BEAM graph.

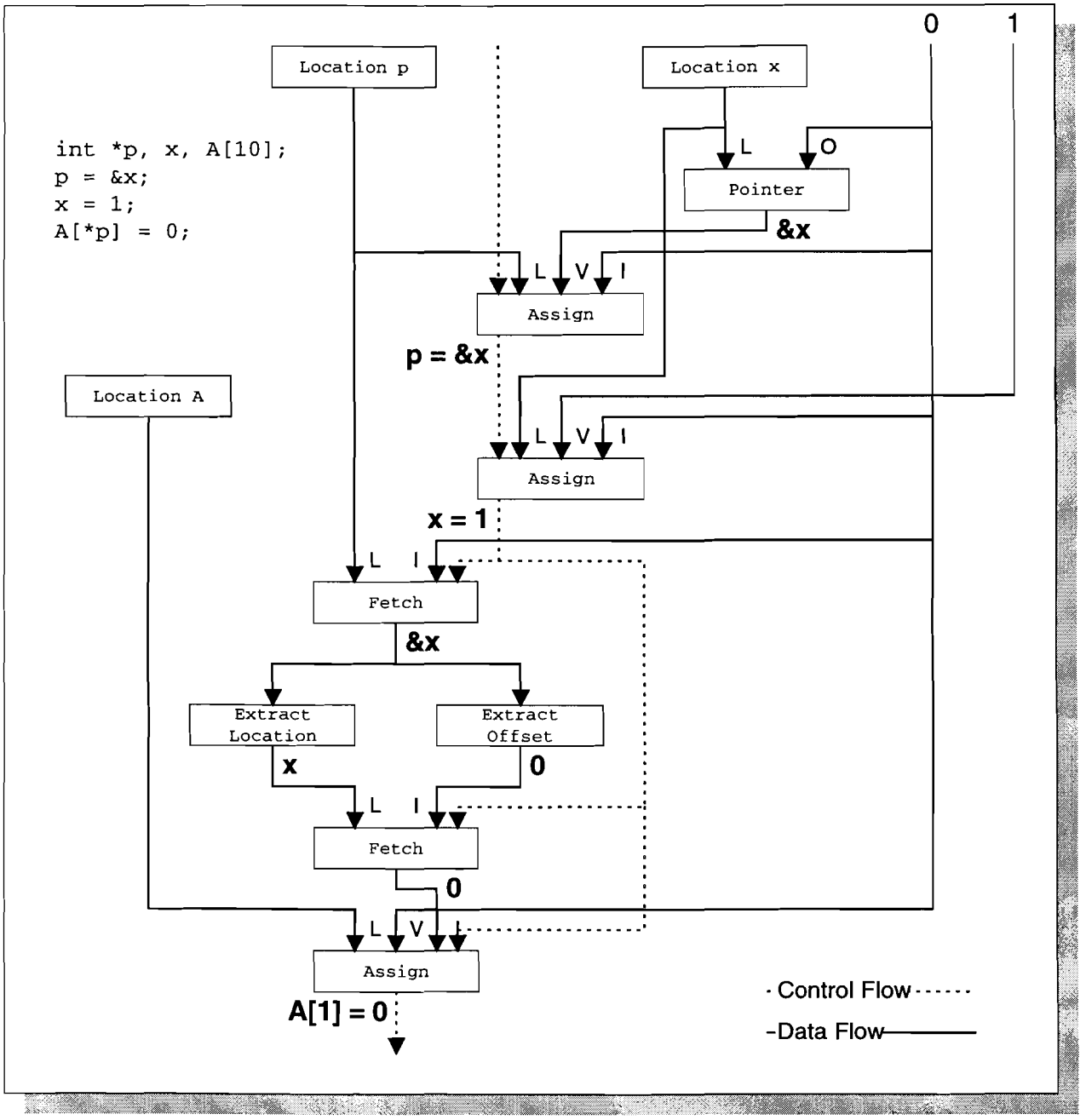


Figure 32: Example BEAM graph with alias/pointer info.