A Methodology for Automated Test Generation for LISA Processor Models

Olaf Lüthje
CoWare, Inc.
Aachen, Germany

Abstract— CoWare's LISATek enables for modelling processor cores at a high level of abstraction. In this paper, a new methodology for analyzing LISA processor models is presented. It is further shown how the results of this analysis can be used to automatically generate sets of test vectors that guarantee a full code coverage of a LISA model.

1 Motivation

CoWare's LISATek [1][2] enables for modelling processor cores at a high level of abstraction that can be refined down to cycle accuracy. Generators exist to automatically create the required software development tools on the basis of the LISA model. These comprise an instruction or cycle accurate simulator, a linker, assembler, profiler, debugger and compiler [3]. Furthermore, a synthesizable hardware description can automatically be generated [4].

At each step towards a synthesizable specification, verification of the refined models is essential for a high quality and fault free design. The standard way to do this is to run a set of test instructions at defined system states on both, the executable specification and the model under test and compare the resulting changes in system state after the execution of each instruction. The achieved confidence in the functional correctness depends on the number and quality of the test cases. The generation of test cases cannot reasonably be done all manually and therefore requires some kind of automation. Different approaches have been presented in literature [5][6][7][8][9]. However, these approaches require the user to manually provide substantial testing knowledge, causing additional cost, time and man power and introducing additional sources of error.

A test generator that perfectly fits into the LISATek design flow ideally works on LISA input files and generates high quality test cases. However, the automated generation of high quality test cases is a very ambitious task that requires a deep "understanding" of the code to be tested.

Researchers have worked on program analysis techniques since the 1960s and there is, by now, an extensive literature [10]. There are two major approaches to program analysis.

- On the one hand there are static analysis techniques that analyze the program code at compile time. Usually sets of equations are set up according to the program semantics and solved by finding their fixpoint. One of the best known static approaches is Data Flow Analysis. It is treated in depth in standard compiler books [11][12][13]. Other techniques such as Constraint Based Analysis and Abstract Interpretation are also described in [14]. PAG [15] is a tool for generating interprocedural data flow analyzers that implement these techniques.
- On the other hand there are techniques for dynamic analysis that are used for examining the behavior of program code during execution. Typically these techniques are employed by profiling tools. Profiling information can for example be used by programmers to find critical pieces of code or as input to profile-driven optimizers.

> Dynamic program analysis techniques have been implemented in tools like Pixie [16] or QPT [17]. By principle, dynamic program analysis relies on input vectors to be processed during execution. Thus the results are of no general nature. They always correspond to the input vectors.

Analysis techniques of neither category are suited for the demands of white box test generation. Static analysis puts tight constraints onto the code to be analyzed. The use of pointers is usually not supported or yields very general results. I. e. every write access to memory is treated as a possible definition for any value read from memory. Implementations of processor behavior models usually make extensive use of pointers though, e. g. for accessing register files or memory. Static analyzers may answer the question, which values the expressions in the code may take on (not *can* take on). But they never will answer the question under which conditions the values are taken on. But that really is what is needed in order to generate tests by a white box approach. Furthermore, the results of static analyzers usually are a superset of the reachable states of the code, i. e. they include impossible states that can never be reached. A white box approach based on that kind of analysis would spend a large part of its computational effort trying to reach impossible states. The same holds for dynamic analysis, but the other way around. The encountered states of profilers are always only a subset of the reachable ones and therefore insufficient for achieving 100% code coverage. Furthermore, profilers require input, i. e. test vectors. But that actually is what is to be generated! In this paper, a new methodology for code analysis is presented that is applied to LISA processor models in order to gain the information that is required to automatically synthesize test cases guaranteeing full code coverage at least redundancy.

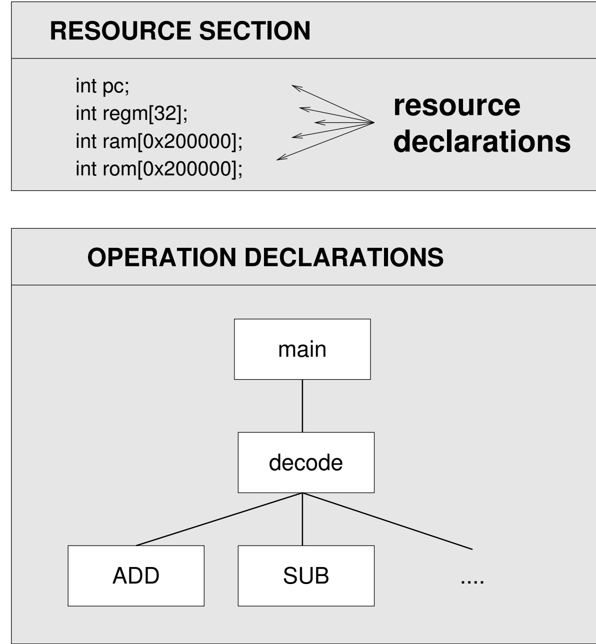


Figure 1: Structure of LISA Language

2 LISA processor models

A LISA processor model is a description of a processor at a high level of abstraction. Among other things, it describes its storage elements, its assembly language and the coding and behavior of its instructions. Fig. 1 depicts the structure of a LISA processor description. The storage elements like registers and memories are instanciated similar to C variables inside the `RESOURCE` section. The instruction set is modelled by a multitude of so called `LISA` operations. Each `LISA` operation comprises different kinds of sections. The behavior of the instructions is modelled as C code inside the `BEHAVIOR` section. This code may contain references to the `BEHAVIOR` sections of sets of other operations, similar to function calls. This way, the `LISA` operations form a tree as indicated in fig. 1. Root of such a tree is always a special operation named `main`. Each processor instruction corresponds to a different path through this tree and its behavior is defined by the C code inside the `BEHAVIOR` sections of the operations along this path. Hence, in order to analyze the behavior of a LISA processor model, C-code has to be analyzed with a few additional LISA semantics.

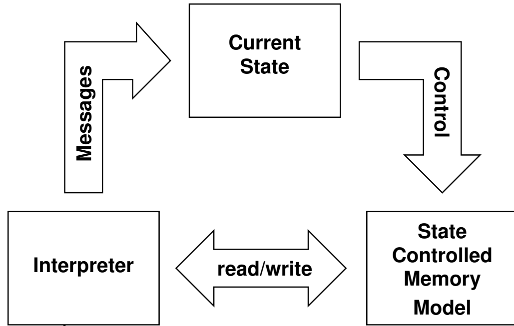


Figure 2: Building blocks of the analyzer

3 The Methodology of Abstract Execution

In the following, we will present a new methodology for analyzing C-code denoted as *abstract execution* that we use to analyze the behavior of LISA operations. Unlike the ones above it is well suited for a white box test pattern generation approach. The requirements for the analysis of processor models with the purpose of test generation are different from those of standard tools like e. g. compilers. The code to be analyzed is far less complex than that of general software projects. The complexity of C code modelling the behavior of a processor is limited to what can reasonably be implemented in hardware. Moreover, the prospective of being able to generate tests that yield 100% code coverage in minutes of simulation time where else hours or weeks would have been required when applying random tests justifies a much more aggressive and computationally expensive approach. In that context, analyzing times of minutes up to hours are well acceptable.

The idea of abstract execution is to track information while interpreting the program. However, in contrast to code being profiled, code being abstractly executed does not process any concrete data (test vectors). Instead, abstract incomplete data is processed. As a consequence, conditional statements may have ambiguous impact on the control flow. This fact is taken into account by the ability of the interpreter to operate in different states. Fig. 2 shows the logical components of the system for abstract program execution. The *Interpreter* executes the code. The *Current State* block allows for alternative executions of ambiguous statements. The *State Controlled Memory Model* combines write accesses in different states to ambiguous data.

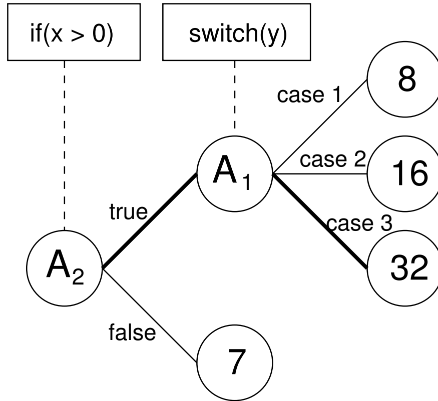
3.1 Data Abstraction

While in concrete execution numeric values are written to and read from memory, in abstract execution *objects* are. An *object* is a collection of information about possible values. The two most important elements are

1. the range, i. e. the minimum value and the maximum value, and
2. a reference to the expression in the code that corresponds to the value.

```
01 int func(int x, int y, int z){
02   int a, b, c, d;
03
04   switch(y){
05     case 1:
06       a = 8; break;
07     case 2:
08       a = 16; break;
09     case 3:
10       a = 32;}
11
12   if(z>0)
13     b = 0;
14   else
15     b = 1;
16
17   if(x>0){
18     c = 5;
19     d = a;}
20   else {
21     c = b;
22     d = 7;}
23
24   return c + d;
25 }
```

Consider the code example above. The only information available about parameters x, y and z is that



A: Ambiguous object
U: Unambiguous object

Figure 3: Representation of ambiguous data

they are in between the minimum and the maximum integer. Hence it cannot be decided which branches of the `switch`- and `if`-statements are executed. This results in an ambiguous content e. g. of variable `b`, namely values `0` [1] and `1`, referring to the expressions in lines `13` and `15` respectively. We combine both *objects* to an *ambiguous object*. In addition, *ambiguous objects* are associated with conditions, under which the alternatives are chosen. In the example, alternative `0` is chosen, if `(z > 0)` is `true`, alternative `1` if it is `false`. In general there may be more than two alternatives and conditions may be combined by a logical `AND`.

In the following we will describe how ambiguities are represented. *Objects* are arranged in trees whose leafs embody the *unambiguous objects*. An example is given in fig. 3.

In general objects are described by the following rules.

- An *object* is either an *unambiguous object* or an *ambiguous object*.
- An *unambiguous object* represents a possible content in memory during concrete execution of a program.
- An *ambiguous object* is associated with a control flow ambiguity in the code (dashed line) and matches each possible branch to an *object*.

Thus these trees do not only contain the alternatives, but also the conditions under which the alternatives are taken. The conditions are determined by all the ambiguities along the path from the root to the alternative. Each ambiguity contributes to the condition in that way, that the condition for the execution of the control flow branch must be fulfilled, that is associated with the link to the next *object* on the path. A logical AND is applied to the contributions of each ambiguity. E. g. the path to *value* 32 indicated by the bold line in fig. 3 represents the condition[2]:

```
(x > 0) == true && y == 3
```

3.2 The State Controlled Memory Model

Primarily, the *State Controlled Memory Model* serves as a regular memory that can be read and written to. Besides that, it is responsible for building the ambiguity trees described in the previous section.

As long as the *current state* is in initial state, the behavior of the state controlled memory model does not differ from a regular memory. Once the *current state* contains a condition, all changes done to memory contents only occur under that condition and result in appropriate ambiguity trees. The state is defined by a set of assumptions about the result of particular expressions in the code. A logical AND is performed on these assumptions. The initial state makes no assumptions at all. Other valid states could for example be '`(x > 0) == true`' or '`(x > 0) == true && y == 3`'. During abstract execution, the state can be changed by the *Interpreter* issuing the following messages.

```
- BeginCondition: <expression> == <value>
- AlternateCondition: <value>
- EndCondition
```

[1]When talking about a value, we mean an object with a range degenerated to a value.

[2]This notation is according to C syntax

The message `BeginCondition` adds an assumption about the result of an expression to the *current state*. A reference to the expression and the assumed value are passed along with the message, e. g. `BeginCondition: (x > 0) == true`. The affected expression `(x > 0)` becomes the *current expression*. The message `AlternateCondition` is used to alter the assumed value for the *current expression*. The message `EndCondition` removes the assumption about the *current expression* from the *current state* and resumes the *current expression* from just before the last `BeginCondition` message was issued. These message triples can arbitrarily be nested. Memory can be read and written to by the following actions.

```
- Read
- Write <object>
```

For simplification, the memory locations that are read and written to are not explicitly given here. In the following we will illustrates the behavior of the memory by a sequence of messages that cause the memory to assemble the ambiguity tree depicted in fig. 3. The current state is shown after each message. `Read`-messages are inserted to show which objects would be read in the different states.

```
Abbreviations:

B: BeginCondition
A: AlternateCondition
E: EndCondition
R: Read
W: Write
t: true
f: false
```

message/ action	current state	object read
B:(x>0)==t	(x>0)==t	-
B:y == 1	(x>0)==t && y==1	-
W:8	(x>0)==t && y==1	-
R	(x>0)==t && y==1	8
A:2	(x>0)==t && y==2	-
W:16	(x>0)==t && y==2	-
R	(x>0)==t && y==2	16
A:3	(x>0)==t && y==3	-
W:32	(x>0)==t && y==3	-
R	(x>0)==t && y==3	32
E	(x>0)==t	-
R	(x>0)==t	A1
A:f	(x>0)==f	-
W:7	(x>0)==f	-
R	(x>0)==f	7
E		-
R		A2

3.3 Iterating over ambiguities

When abstractly executing statements (section 3.4), one has to iterate over the alternatives of ambiguities. This is basically done by traversing the corresponding tree. However, the *current state* is taken into account, i. e. only those alternatives are *visible*, whose conditions are not contradictory to the *current state*. Furthermore when selecting an alternative from an ambiguity, the corresponding conditions are - if not yet included - added to the *current state*. This way, the following is achieved

- All data couplings are taken into account, i. e. no impossible cases are considered.
- Alternative executions of statements can be done without further thought about the *current state* (see section 3.4).
- The conditions of newly created ambiguities are automatically reduced to the input values of the code.

3.4 Execution of a program

3.4.1 Sequential Code

Fig. 4 shows two sequential statements. The solid lines represent the control flow of a concrete execution. Abstract execution also follows that control flow. However, statements that depend on ambiguous data are executed multiple times (dashed lines), once for every possible vector of the involved ambiguities. The vectors are iterated over as described in section 3.3. Thus every execution is performed in a different *current state*, such that changes in memory together with their corresponding states are stored in ambiguity trees.

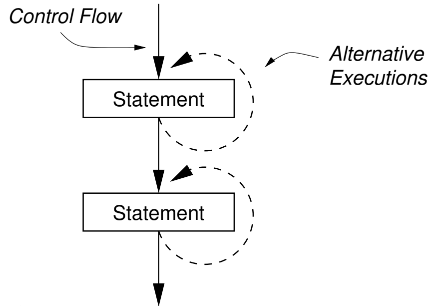


Figure 4: Abstract executions of sequential statements

3.4.2 Nested Statements

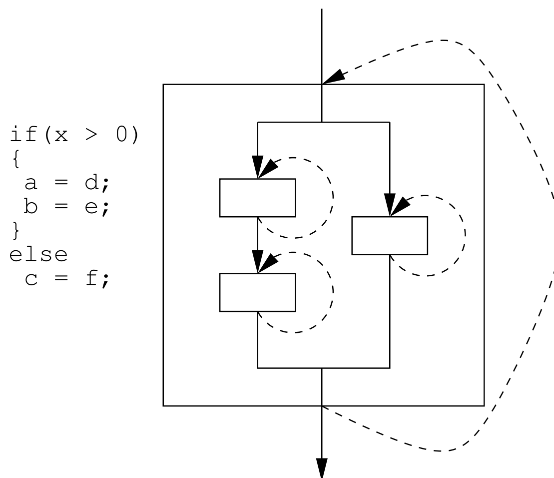


Figure 5: Abstract Execution of Nested Statements

In conjunction with nested statements like the `If`-statement in fig. 5, the algorithm of section 3.4.1 is applied recursively. Note that the iterations over outer statements may affect (i. e. reduce) the number of iterations of inner statements through the *current state* by masking the visible alternatives.

3.4.3 Loops

As long as the breaking condition of a loop is unambiguously not fulfilled, the loop body is (abstractly) executed and the next iteration is started. Is the breaking condition unambiguously fulfilled, the loop exits. This is exactly the same behavior as in concrete execution. In abstract execution, evaluations of the breaking condition may be ambiguous, though. Iterations with ambiguous evaluations of the breaking condition are treated as `If`-statements that include all further iterations. In general, this method is applied recursively. The number of iterations may depend on input data, but the loop must terminate deterministically. Loops with possibly infinite iterations have to be annotated by the user. However, they are uncommon to processor models.

3.5 Analyzing LISA code

In order to simulate the behavior of an instruction, the simulator executes the behavior code inside the `BEHAVIOR` section of the `main` operation. From here, the `BEHAVIOR` sections of other inferior operations are recursively called. Analyzing a processor model with the presented methodology, abstract execution starts with the `main` operation as well. However, in contrast to regular simulation, it is not known, which instruction is to be executed. Neither is the processor's state, i. e. register and memory contents. Resulting ambiguities are handled by subsequent execution of all alternatives, each in its own state, see section 3. Thus, abstract execution of a single *unknown* instruction represents the execution of *any* instruction at *any* state of the processor.

4 Test Generation

The methodology of analyzing processor models described in section 3.5 enables a white box approach to test generation.

So far, it has been described how an abstract instruction is executed on a processor model at absence of concrete knowledge about both the instruction itself and processor state. In the following it is presented how this technique is utilized to formulate conditions for each part of the code, that have to be fulfilled for it to be executed. These conditions are reduced to the primary input values of the processor

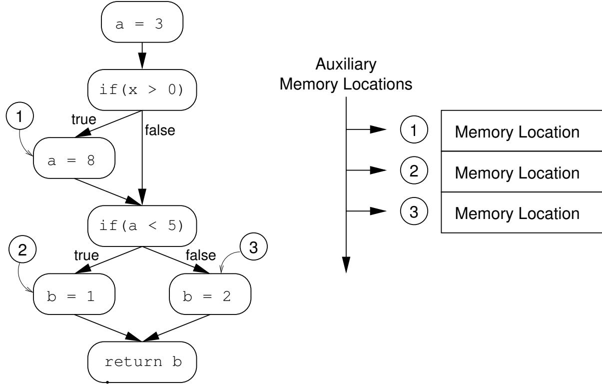


Figure 6: Basic Blocks with Auxiliary Memory Locations

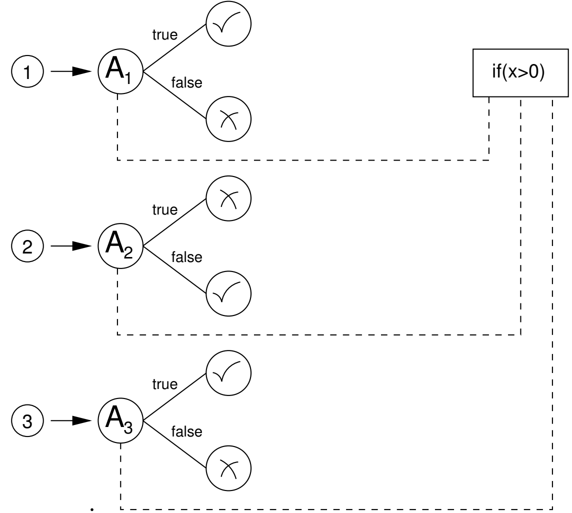


Figure 7: Ambiguities In Memory Locations According To Fig. 6

model forming a system of equations. This system of equations can be solved with a constraint satisfaction solver directly leading to the desired test vectors as a solution.

In order to find the conditions that have to be fulfilled for a full code coverage, the code of the LISA model is broken up into a set of basic blocks. For each basic block, an auxiliary memory location is allocated before analysis. This is shown in fig. 6. There are two kinds of pseudo values to be written to these memory locations: One value representing that the corresponding basic block has *not* been executed (cross) and one representing that it *has* been executed (check mark). The former is the initial value of the memory locations. The latter is written to the memory locations every time the corresponding basic block is executed during abstract execution. As described in 3.2 all these pseudo values written in different states to the memory locations are automatically combined to ambiguity trees. These do not only represent the values themselves but also the conditions under that they had been written. Therefore, the ambiguities stored in the auxiliary memory locations at the end of abstract execution represent the conditions, under which the corresponding basic blocks are executed or not. In the following they are referred to as *coverage values.*

Fig. 7 shows the coverage values of the three basic blocks. Each block may or may not be executed. In case the condition of the first `if`-statement is true, blocks one and three are executed, in case it is false, block two will. Note that the condition for the execution of blocks two and three has been reduced to the input `x` (condition `(x > 0)`) although `x` does not appear in the condition of the enclosing `if`-statement.

The conditions that have to be fulfilled by a test vector for a certain basic block to be executed are easily found by iterating over the ambiguities of the corresponding coverage value until an *is executed* value is encountered. Two tests are necessary to cover this very simple example code.

5 Results

The presented techniques of processor model analysis and test generation have been realized as prototypes. The concepts have been proved on two processor models representing the complexity range of typical target processors. The LISA tutorial processor QSip (10 instructions) was chosen to represent an application of very low complexity. Analysis re-

quired less than two seconds[3] and yielded eleven test cases. The ARM7 processor was chosen to represent the more complex architectures. It required 15 minutes for analysis. 173 test cases were needed for a 100 percent code coverage. Keeping in mind that 50.000 randomly generated test cases only yield a 95.7 percent code coverage of the ARM7, this is an enormous reduction in verification effort. For each resulting equation system, more than one, most likely even hundreds and thousands of solutions i. e. test cases can be found. Thus, the number of test cases can arbitrarily be multiplied in order to create a large set of tests that is optimally balanced over the processor's functionality.

The case studies show that complexity of the presented methodologies is manageable. They also show, that the number of test cases required for a full code coverage can significantly be reduced by applying advanced methods of test generation. The presented methodology of processor model analysis has only been applied in order to generate test cases with a full code coverage. However, it is also well suited for pursuing other goals than that. This could e. g. be the test of corner cases.

[3]on a PC with an AMD Athlon processor at 1,2 GHz with 512 MB RAM

References

[1] Andreas Hoffmann, Tim Kogel, Achim Nohl, Gunnar Braun, Oliver Schliebusch, Oliver Wahlen, Andreas Wieferink, and Heinrich Meyr. A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIPs) Using a Machine Description Language. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 20:1338–1354, November 2001.

[2] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors with LISA*. PhD thesis, Institute for Integrated Signal Processing Systems, RWTH Aachen, November 2002.

[3] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun and H. van Someren, A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Paris, France, Feb 2004.

[4] O. Schliebusch, A. Chattopadhyay, M. Steinert, G. Braun, A. Nohl, R. Leupers, G. Ascheid and H. Meyr. RTL Processor Synthesis for Architecture Exploration and Implementation. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE) - Designers Forum*, Paris, France, Feb 2004.

[5] Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molche, and Gil Shurek. Test program generation for functional verification of powerpc processors in ibm. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 1995.

[6] L. Fournier. Genesys-X86: An Automatic Test-Program Generator for X86 Microprocessors. IBM Haifa Research Center VLSI Internal Publication.

[7] Ulrich Bieker and Peter Marwedel. Retargetable self-test program generation using constraint logic programming.

[8] J. Grabowski, B. Koch, M. Schmitt, and D. Hogrefe. SDL and MSC Based Test Generation for Distributed Test Architectures. In *Proceedings of the Nineth SDL Forum*, Montreal, Canada, June 1999.

[9] B. Koch, D. Grabowski, D. Hogrefe, and M. Schmitt. AutoLink - A Tool for Automatic Test Generation from SDL Specifications. In *IEEE International Workshop on Industrial Strength Formal Specification Techniques (WIFT)*, Boca Raton, Florida, October.

[10] Chris Hankin. Program Analysis Tools. *International Journal on Software Tools for Technology Transfer*, 2(1):6–12, 1998.

[11] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[12] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[13] Michael J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing, Redwood City, CA, 1996.

[14] Chris Hankin Flemming Nielson, Hanne Riis Nielson. *Principles Of Program Analysis*. Springer, 1999.

[15] F. Martin. PAG - An Efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer*, 2(1):6–12, 1998.

[16] MIPS Computer Systems. *UMIPS-V Reference Manual (pixie and pixstats)*, 1990.

[17] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, July 1994.