GamaSlicer: an Online Laboratory for Program Verification and Analysis*

Daniela da Cruz, Pedro Rangel Henriques and Jorge Sousa Pinto

Informatics Department, University of Minho
Braga, Portugal

Abstract

In this paper we present the GamaSlicer tool, which is primarily a semantics-based program slicer that also offers formal verification (generation of verification conditions) and program visualization functionality. The tool allows users to obtain slices using a number of different families of slicing algorithms (precondition-based, postcondition-based, and specification-based), from a correct software component annotated with pre and postconditions (contracts written in JML-annotated Java). Each family in turn contains algorithms of different precision (with more precise algorithms being asymptotically slower). A novelty of our work at the theoretical level is the inclusion of a new, much more effective algorithm for specification-based slicing, and in fact other current work at this level is being progressively incorporated in the tool.

The tool also generates (in a step-by-step fashion) a set of verification conditions (as formulas written in the SMT-lib language, which enables the use of different automatic SMT provers). This allows to establish the initial correctness of the code with respect to their contracts.

1 Introduction

The goal of program verification is to establish that a program performs according to some intended specification. Typically, what is meant by this is that the input/output behavior of the implementation matches that of the specification (this is usually called the *functional* behavior of the program), and moreover the program does not 'go wrong', for instance, no errors occur during evaluation of expressions (the so-called *safety* behavior).

Modern program verification systems rely on the use of a *Verification Conditions Generator* (VCGen for short), a program that reads in a piece of code together with a specification, and produces a set of first-order proof obligations (called *verification conditions*) whose validity will imply that the code is

*This work was supported by project RESCUE, funded by FCT (PTDC/EIA/65862/2006).

(partially or totally) correct with respect to the specification. The underlying theoretical framework is some program logic, typically Hoare logic [12] (but other more recent and sophisticated logics exist, such as separation logic [17]). Specifications are expressed in terms of preconditions and postconditions, which are in general formulas of first-order logic.

In this context, the *Design by Contract* approach to software development [15], which proposes that a program's procedures / methods be annotated with contracts that can be checked at runtime, may be fruitfully combined with program verification, since the notion of contract coincides precisely with the above notion of specification. The idea behind contract-based verification is that a program (defined as a set of mutually recursive procedures with contracts) is correct if all its individual procedures are correct with respect to their own individual contracts. Thus procedures can be independently verified in a modular way.

The combination of *slicing techniques* with program verification has been proposed [11, 5, 4] with a double goal: on one hand, slicing may help in optimizing the verification process, allowing one to delete the statements that do not contribute to the validity of a given specification (the resulting slice is smaller and thus easier to verify than the original program). On the other hand, verification inspires a novel, much more powerful way of slicing programs (with respect to traditional, syntax-based slicing), based on their axiomatic semantics.

In this paper we introduce GamaSlicer, a tool that includes both traditional program verification functionality (it is a Verification Conditions Generator) and also a highly parameterizable semantic slicer. It includes all the published algorithms (precondition-based slicing [4], postcondition-based slicing [5], specification-based slicing [4] and their variants) as well as a new, more effective algorithm that we have developed. The tool is extremely useful for comparing the effectiveness of slicing algorithms; it also allows users to perform diverse workflows such as for instance

- First verifying a program with respect to a given specification; then slicing it with respect to that specification to check if there are irrelevant commands (with respect to that specification).
- Then, from a verified program, producing a specialization by weakening the specification and slicing the program with respect to the weaker spec. This may be useful in a program reuse context, in which a weaker contract is required of a component than the actually implemented contract.

The remainder of this paper is organized as follows. Section 2 briefly introduces the background on program verification and slicing to give theoretical support to GamaSlicer. Section 3 presents the architecture of GamaSlicer, and its features. Section 4 discusses applications and usage of GamaSlicer. Section 5 closes the paper with some remarks and future work.

$$
\begin{array}{rcl}
\mathsf{wp}\,(\mathbf{skip}, Q) & = & Q \\
\mathsf{wp}\,(x := e, Q) & = & Q[x \mapsto e] \\
\mathsf{wp}\,(C_1; C_2, Q) & = & \mathsf{wp}\,(C_1, \mathsf{wp}\,(C_2, Q)) \\
\mathsf{wp}\,(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) & = & (b \rightarrow \mathsf{wp}\,(C_t, Q))\ \&\&\ (\ !b \rightarrow \mathsf{wp}\,(C_f, Q)) \\
\mathsf{wp}\,(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ C, Q) & = & I \\
\mathsf{wp}\,(\mathbf{call}\,\mathbf{p}, Q) & = & \mathtt{Forall}\ \overline{x_f}.\ (\mathsf{pre}(\mathbf{p}) \rightarrow \mathsf{post}(\mathbf{p})\,[\overline{x}/\overline{x^{\sim}}, \overline{x_f}/\overline{x}]) \rightarrow Q[\overline{x_f}/\overline{x}] \\
 & & \\
\mathsf{VC}(\mathbf{skip}, Q) & = & \mathtt{true} \\
\mathsf{VC}(x := e, Q) & = & \mathtt{true} \\
\mathsf{VC}(C_1; C_2, Q) & = & \mathsf{VC}(C_1, \mathsf{wp}\,(C_2, Q))\ \&\&\ \mathsf{VC}(C_2, Q) \\
\mathsf{VC}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) & = & \mathsf{VC}(C_t, Q)\ \&\&\ \mathsf{VC}(C_f, Q) \\
\mathsf{VC}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ C, Q) & = & (I\ \&\&\ b) \rightarrow \mathsf{wp}\,(C, I)\ \&\&\ \mathsf{VC}(C, I)\ \&\&\ (I\ \&\&\ !b) \rightarrow Q \\
\mathsf{VC}(\mathbf{call}\,\mathbf{p}, Q) & = & \emptyset
\end{array}
$$

Figure 1: VCGen algorithm: weakest precondition and verification conditions. The operator $\mathcal{N}(\cdot)$ returns a sequence of the variables occurring free in its argument assertion.
Given a sequence of variables $\overline{x} = x_1, \ldots, x_n$, we let $\overline{x_f} = x_{1f}, \ldots, x_{nf}$ and $\overline{x^{\sim}} = x_1{}^{\sim}, \ldots, x_n{}^{\sim}$.

2 Theoretical Background

We give here a brief overview of the theoretical framework underlying GamaSlicer, but we omit formal definitions and proofs. The reader is directed to [10, 7] for more details.

Program Verification. The verification infrastructure consists of a program logic used to assert the partial correctness of individual procedures, from which a VCGen algorithm is derived (Figure 1). The VCGen is proved correct, i.e. it is guaranteed to generate, from a Hoare triple $\{P\}\,C\,\{Q\}$, a set of proof obligations whose validity is sufficient for the triple to be valid, i.e. for the program C to be correct with respect to precondition P and postcondition Q (technically, the correctness result states that there must exist a Hoare logic derivation having the triple as conclusion).

The standard method for generating verification conditions relies on an algorithm that uses the weakest precondition strategy. Our VCGen algorithm is based on the usual function $\mathsf{wp}\,(C, Q)$ that calculates, from a command C and a postcondition Q, the weakest precondition required to grant the truth of Q after terminating executions of C; the auxiliary function $\mathsf{VC}(C, Q)$ returns a set of verification conditions sufficient to ensure the validity of the Hoare triple $\{\mathsf{wp}\,(C, Q)\}\,C\,\{Q\}$. Notice that determining weakest preconditions of loops is straightforward in our context, since all loops are annotated with invariants that are taken into account.

For a given Hoare triple, the VCGen simply collects the verification conditions generated for each procedure and function with the VC function using the contract's postcondition, with an additional condition stating that the contract precondition must be stronger than the calculated weakest precondition. Let Π

be a program consisting of procedures C_i with $i \in \{1, \ldots, n\}$, each annotated with a contract consisting of precondition P_i and postcondition Q_i. Then the verification conditions for Π are given as

$$\bigcup_{i=1\ldots n} \{P_i \rightarrow \mathsf{wp}\,(C_i, Q_i)\} \cup \mathsf{VC}(C_i, Q_i)$$

Thus the validity of the verification conditions generated from Π implies the correctness of all the procedures in Π with respect to their contracts.

A final note with respect to establishing the validity of first-order logic formulas: the VCGen produces verification conditions whose validity must be checked by some external proof tool; the slicing algorithms described next also require the use of an external proof tool. GamaSlicer uses SMT-solvers for this purpose.

Program Slicing. The basic idea of *slicing* – a code analysis technique introduced by Weiser [18] – is to isolate a subset of program statements, either those (directly or indirectly) *contributing to* the value of a set of variables V_s at a program location p, or those *influenced by* the value of that set of variables at location p. These two forms are known as *backward slicing* and *forward slicing* respectively; $C(p, V_s)$ is called a *slicing criterion*. Statements not interfering with the set of variables isolated are removed, enabling software engineers to concentrate on the statements that are relevant for the task at hand.

Comuzzi et al [5] and Chung et al [4] have provided algorithms for code analysis enabling to identify spurious commands (commands that do not contribute to the validity of the postcondition). The former paper presents a variant of program slicing, called *p-slice* or *predicate slice*, using Dijkstra's weakest preconditions to determine which statements will affect a specific predicate. The latter argues that the information present in the annotations helps to produce more precise slices by removing statements that are not relevant to that specification.

These two papers lay the foundations of slicing based on preconditions (a semantic form of forward slicing) or on postconditions (a semantic form of backward slicing). Both notions can be implemented at different levels of precision (more precision requires slower execution times; for this reason GamaSlicer allows the user to select between linear-time and quadratic time algorithms on the length of the program). The notion of *specification-based slicing*, also introduced by Chung, can be informally described as follows: given a program C which is correct with respect to precondition P and postcondition Q, it is safe (in the sense that the resulting program will still be correct with respect to the same specification) to remove from C all the statements that will never be executed (because P precludes that execution), as well as all the statements whose execution does not affect the truth of Q in the final state of the program.

The algorithm proposed [4] for implementing specification-based slicing consists of first slicing the program with respect to the precondition (ignoring the postcondition), and then with respect to the postcondition (ignoring the precondition). In our theoretical work we have found that a much more precise specification-base slice is obtained by algorithms that use simultaneously the

precondition and postcondition. The trade-off between execution time and precision is still present, so different algorithms can be given based on this principle.

Examples comparing all these notions of slicing and the algorithms used to calculate them, together with the definition of our own specification-based slicing, can be found in [7]. Note that since semantic slicing relies on first-order logic (a given statement is sliced off if some first-order formula can be proved), it is of course a conservative transformation – if some formula (possibly valid) cannot be proved, then the corresponding command will not be sliced off.

Visualization. The problem of *visualizing program slices* is a part of the larger problem of *program visualization* – the relevant question here is how to display the interdependencies and relationships among program components in an user-friendly way, that effectively helps understanding programs. As slicing removes non-relevant statements from the source code, it is important that the visual representation clearly distinguishes sliced statements from other, remaining statements. This is a challenge.

The visualization of a program, using a System Dependency Graph (SDG) actually helps in perceiving the relationships holding among program components, and has been widely used by many other tools [13, 3, 1, 2]. However, as we are dealing with programs with contracts, we felt the need to extend this notion of SDG. We call this extension *Annotated System Dependence Graph*, SDG_a for short. Essentially, an SDG_a is an SDG in which some nodes are *annotated*. An annotated node is a block composed by a statement or a predicate (a control statement or entry node) and one or more annotations (a precondition, a postcondition, or an invariant). We are currently supporting the CFG_a in order to visualize assertion-annotated programs as well as calculated program slices. We are now looking for a suitable way to exhibit the SDG_a.

3 GamaSlicer, an Overview

GamaSlicer is an online laboratory[1] that includes a VCGen, a parameterizable slicer with a choice of algorithms, and visualization functionality (not present in the current version). It works on Java programs with JML annotations (the standard specification language for Java [14]). Instead of programs consisting of sets of procedures, one has classes with their methods, sharing a set of class/instance variables instead of global variables. The fundamental idea remains the same: the verification task concerns a set of mutually recursive methods.

The architecture of GamaSlicer, inspired by that of a compiler (or generally speaking a language processor), is depicted in Figure 2. It is composed by the following blocks: a Java/JML front-end (a parser and an attribute evaluator); a verification conditions generator; a proof obligations generator; a theorem-prover (not included in Figure 2, as we call an external one instead of building our own); a slicer; and an annotated control flow graph (CFG_a) visualizer.

[1] Current version available at `http://gamaepl.di.uminho.pt/gamaslicer`

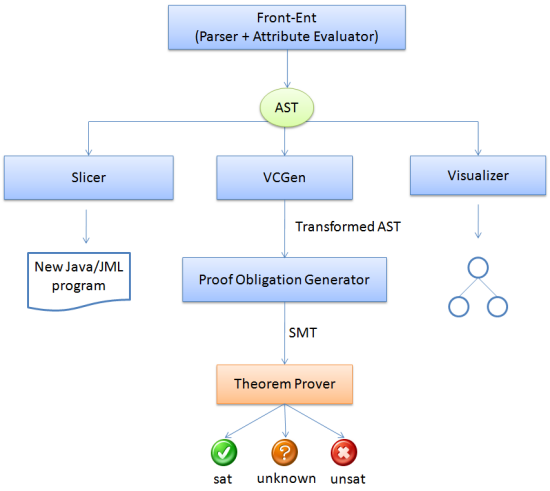

Figure 2: GamaSlicer architecture

Although the GamaSlicer works with Java/JML programs to take advantage from a previous project[2], the tool was developed under .NET environment in C# due to its easiness of web programming and because we intend to extend GamaSlicer to work also with Spec# programs.

The tool outputs proof obligations written in the SMT-Lib (Satisfiability Modulo Theories library) language. We chose SMT-Lib since it is nowadays the language employed by most provers used in program verification, including, among many others, Z3 [8], Alt-Ergo [6], and `Yices` [9].

After uploading a file containing a piece of Java code together with a JML specification, the code is recognized by the front-end (a C# analyzer produced automatically from an attribute grammar with the help of the ANTLR parser generator [16]), and is transformed into an AST. During this first step also an identifiers table is built.

In the next step, the VCGen, implemented in C# as a tree-walker evaluator, traverses the AST to generate the verification conditions, using a kind of tree-pattern matching strategy (represented in Figure 2); each time a tree matches a pattern (corresponding to the eight cases identified in the algorithm of Figure 1), it is transformed and marked as *visited*. Before this operation is performed, a tree traversal is done to look for methods with contracts; this search returns a set of subtrees, and the VCGen algorithm is applied to each one.

In a third step, the new AST (now with verification conditions attached to the nodes) is traversed by the SMT code generator (another C# tree-walker evaluator) to produce SMT proof obligations. If selected by the user, an external automatic Theorem-Prover is then called to prove the generated SMT formulae. Actually, when this facility is activated, three theorem provers are called, to

[2]An existing Java/JML grammar for ANTLR that simultaneously outputs C#.

allow for results and performances to be compared.

Alternatively to step three, a fourth step can be performed. Using the AST to derive the SDG_a for the Java/JML class under analysis, the contract-based slicer applies the user-selected algorithm in order to detect and remove a set of statements that are not relevant with respect to the specification annotated in the program, following the discussion in the previous section.

The fifth step is a graphical add-on, that is currently being developed; it takes the SDG_a, before or after slicing, and exhibits the data and control flow inside the methods and inside the class. Statements deleted by the (contract-based) slicer are illustrated on the SDG_a graphical representation – although implemented and tested separately, this visualizer is not yet integrated in the publicly available version of GamaSlicer.

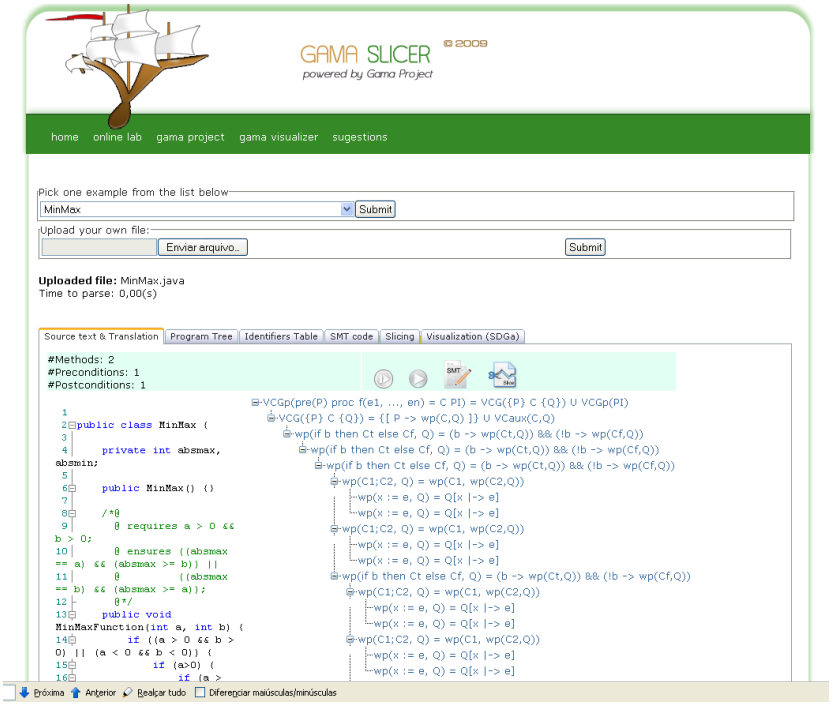

Figure 3: Tab 1: Source Program and VCGen Rules applied

The intermediate information that becomes available after each of the above steps is displayed in an internet browser window distributed by six main tabs:

- *Tab 1*: contains the Java/JML source program to be verified and also the rules applied along the verification conditions generation process; some statistical information is also displayed at the top of the window, describing the program size and annotation complexity (see Figure 3).

Figure 4: Tab 5: Sliced Program

- *Tab 2*: contains the syntax tree (AST) generated by the front-end.
- *Tab 3*: contains the identifiers table built during the analysis phase (also at step one).
- *Tab 4*: contains the generated SMT code; it will also display a table with the verification status of each formula (`sat`, `unsat`, `unknown`) after calling a theorem-prover; for each prover invoked, the time consumed to prove the formulae is also displayed.
- *Tab 5*: contains the new program produced by the slicer (applying the specification-based slicing techniques to the original program); notice that useless statements identified by the slicer are not actually removed, but shown in red and strike-out style (see Figure 4).
- *Tab 6*: displays the CFG_a as the visual representation of the program, giving an immediate and clear perception of the program complexity, with its procedures and the relationships among them; nodes in the program's graph that were sliced away are marked in red (helping to understand quickly which statements were removed / preserved). As navigation capabilities over the visual representation, GamaSlicer allows to expand and collapse nodes, and to jump into the source code by clicking a node.

4 GamaSlicer, Usage and Applications

The first motivation for the design and development of this online laboratory was to make available a tool that implements semantic slicing algorithms, with the well-known applications made possible by the combination of verification and slicing technology, while at the same time allowing us to test our improved slicing algorithms.

We now list some applications of the tool. Note that in a properly annotated program, slicing should leave the code intact, thus one first application is to remove useless code. Other applications involve slicing the program using to weaker specifications (with respect to the original satisfied by the program).

An obvious one is program comprehension: slicing a method according to different contracts (by weakening the original contract) allows one to understand the relation between each code section and each part of the initial contract, and also to slice programs for reuse or specialization purposes.

GamaSlicer can also be used to help in detecting and fixing errors, both in annotations and in the methods' code, since it exhibits in a versatile and user-friendly tree fashion all the rules used to generate the proof obligations; this output can optionally be obtained by executing the VCGen algorithm step-by-step. The proof obligations, written in the SMT-Lib language, are also displayed, allowing the user to analyze the formulae and their verification status.

Due to the outputs delivered and the interaction modes, we have found that GamaSlicer can assist programmers with the improvement of their skills to annotate programs. A simple way to explore that idea is to take as input a simple and well-known procedure and then play with different annotations observing the consequences on proving process. This application trend is clearly reinforced by the slicer, which detects and displays statements that do not contribute to the specification.

From our experience, this system can also be a useful teaching tool, since it allows students to observe step-by-step how verification conditions are generated. For example, since loop invariants and procedure contracts are annotated into the code, the only arbitrary choice is in the rule for the sequence command $C_1 \,;\, C_2$, in which an intermediate assertion R must be guessed. This is the motivation for introducing a strategy for the construction of proof trees, based on the notion of weakest precondition. The result is a mechanical method for constructing derivations, which is inbuilt in the VCGen. This transition from program logic to VCGen is not trivial to understand, and this tool clearly helps in that process. In an advanced course, it is also useful that students can easily modify the underlying algorithms (both VCGen and slicing).

5 Conclusion

In this paper we propose a demonstration of a tool that is intended, primarily, as a development laboratory to test ideas and algorithms in the context of *verification condition generation* and *semantics-based slicing*; even so, it is, to

our knowledge, the only available tool that implements semantic slicing based on contracts, as well as integration of verification and slicing capabilities. The tool is web-based, which means that everyone can use it without having to download any source or executable code. Its interface is very simple and its usage completely intuitive. A repository of Java/JML example programs is available from our online Laboratory.

As future work, we intend to work on the scalability of the tool to handle real-size code, as well as improving the visualizer component and including other slicing algorithms which are currently being developed.

References

[1] Paul Anderson and Tim Teitelbaum. Software inspection using Codesurfer. In *Workshop on Inspection in Software Engineering*, 2001.

[2] Giuliano Antoniol, Roberto Fiutem, G. Lutteri, Paolo Tonella, S. Zanfei, and Ettore Merlo. Program understanding and maintenance with the CANTO environment. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 72, Washington, DC, USA, 1997. IEEE Computer Society.

[3] FranCcoise Balmas. Displaying dependence graphs: a hierarchical approach. *J. Softw. Maint. Evol.*, 16(3):151–185, 2004.

[4] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 605–609, New York, NY, USA, 2001. ACM.

[5] Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pages 557–575, London, UK, 1996. Springer-Verlag.

[6] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.

[7] Daniela da Cruz, Jorge Sousa Pinto, and Pedro Rangel Henriques. Specification-based slicing and slice graphs. `http://alfa.di.uminho.pt/~danieladacruz/techReportCPH09b.pdf`, October 2009.

[8] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[9] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[10] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. Technical Report DI-CCTC-08-01, Universidade do Minho, 2008.

[11] Mark Harman, Rob Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. *icsm*, 00:138, 2001.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[13] Jens Krinke. Visualization of program dependence and slices. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.

[14] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, 2004.

[15] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[16] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[17] John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.

[18] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.