Mutation of Model Checker Specifications for Test Generation and Evaluation

Paul E. Black
National Institute of
Standards and Technology
Gaithersburg, MD 20899
paul.black@nist.gov

Vadim Okun Yaacov Yesha
University of Maryland
Baltimore County
Baltimore, MD 21250
{vokun1,yayesha}@cs.umbc.edu

Abstract

Mutation analysis on model checking specifications is a recent development. This approach mutates a specification, then applies a model checker to compare the mutants with the original specification to automatically generate tests or evaluate coverage. The properties of specification mutation operators have not been explored in depth. We report our work on theoretical and empirical comparison of these operators. Our future plans include studying how the form of a specification influences the results, finding relations between different operators, and validating the method against independent metrics.

Keywords: specification mutation, mutation operators, test generation, model checking.

1 Introduction

Mutation analysis is typically performed on program code. However, a specification provides additional valuable information. For instance, specification-based testing may detect a missing path error [15], that is, a situation when an implementation neglects an aspect of a problem and a section of code is altogether absent. Further, code-based analysis is not possible for some systems because testers do not have access to the source code. Analysis on a specification can also proceed independently of program development, and any results should apply to all implementations of the specification, e.g. ports to other systems. Model checking and specification-based mutation analysis are combined in a novel method to automatically produce tests from formal specifications [3] and measure test coverage [2]. We briefly introduce model checking here.

1.1 Model Checking

Model checking is a formal technique based on state exploration. Input to a model checker has two parts. One part is a state machine defined in terms of variables, initial values for the variables, environmental assumptions, and a description of the conditions under which variables may change value. The other part is a set of temporal logic expressions over states and execution paths.

Temporal logic is an extension of classical logic for dealing with systems that evolve with time. The properties such as "It will be the case that p", "It is always the case that p" can be compactly specified in temporal logic.

Conceptually, a model checker visits all reachable states and verifies that each temporal logic expression is consistent with the state machine, e.g., satisfied over all paths. If an expression is not satisfied, the model checker generates a counterexample in the form of a trace or sequence of states, if possible.

The SMV Model Checker

We use the SMV [19] model checker. Its temporal logic is Computation Tree Logic (CTL) [11]. Typical formulas in CTL include:

- `AG safe`

  All reachable states are safe.

- `AG (request → AF response)`

  A request is always followed by a response sometime in the future.

Figure 1 is a short SMV example. "Request" is an input variable, and "state" is a variable with possible values "ready" and "busy." The initial value of state is "ready." The next state is "busy" if the state is "ready" and there is a request. Otherwise the next state is "ready" or "busy" nondeterministically. The SPEC clause is a CTL formula which states that whenever there is a request, state will eventually become "busy."

Some might object that SMV's description language is at too low a level for wide-spread use, and we agree. A

```
MODULE main
VAR
    request : boolean;
    state : {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) := case
        state = ready & request : busy;
        1 : {ready, busy};
    esac;
SPEC AG (request -> AF state = busy)
```

Figure 1. A Short SMV Example

practical system must extract state machines and temporal logic expressions from higher level descriptions such as SCR specifications [4], MATLAB stateflows [5], or UML state diagrams.

Notice that choosing a different model checker naturally leads to a different specification language and therefore potentially different mutation operators and effects. We comment on the interaction between the form of a specification and the results of mutation analysis in Section 4.

In Section 2, we describe how we use a model checker and mutation analysis on specifications to generate tests. We report some of our findings of which mutation operators are better than others in terms of their mutation coverage and the number of mutants generated in Section 3. Finally in Section 4 we present some open questions and the research directions we have planned to take to address them.

2 Mutations for Test Generation

Ammann and Black used mutation analysis, along with model checking, to automatically produce tests from formal specifications [3] and measure test coverage [2]. Since our work has been in the framework of this approach, we briefly explain it here.

One begins with a finite state machine representation, or specification, of the system to be tested. Each transition of the state machine is reflected as a CTL clause. For instance, the first case of the state variable in Figure 1 may be expressed as the following clause.

SPEC AG (state = ready & request $\rightarrow$ AX state = busy)

Methods of turning order-dependent guards into order-independent CTL, expressing constructs which have no parallel in CTL, making the default case explicit, and minimizing expressions are given in [2]. The set of clauses derived from all state machine transitions, which are consistent with the state machine, are combined with any pre-existing clauses to serve as the specification.

Although other test criteria[1] could be applied [8, 14], we confine ourselves here to a specification mutation adequacy criterion. Simply stated, the criterion is that a test set must kill all mutations of a specification produced with some set of mutation operators. A mutant is killed if the mutant CTL clause is shown inconsistent with the trace, or history of execution states, of a test case. An equivalent, or consistent, mutant is true over all traces.

For program-based mutation analysis, detecting equivalent mutants is, in general, an undecidable problem. However, to use model checkers we restrict ourselves to a finite domain in which equivalent mutant identification is decidable. In fact, model checkers are designed to perform this equivalence check efficiently. The model checker finds equivalent mutants to be consistent with the state machine, so they may be automatically discarded.

To generate tests, one mutation operator is applied to all the CTL clauses. Applying each mutation operator in turn yields a set of mutant clauses. The model checker then compares the original state machine specification with the mutants. When the model checker finds a clause to be inconsistent, it produces a counterexample if possible. The counterexamples contain both stimulus and expected values, so they may be automatically converted to complete test cases. To reduce the number of tests, duplicate counterexamples are combined, and counterexamples which are prefixes of others are discarded.

Note that the number and type of mutation operators, as well as the form of the CTL clauses, influences the number and breadth of tests produced.

A variant of this approach may be used to evaluate coverage of a test set. Each test is turned into a finite state machine constrained to express only the execution sequence of that test. The model checker compares each constrained finite state machine with the set of mutants produced previously. A mutation adequacy coverage metric is the number of mutants killed divided by the total number of mutants. This simple metric may be made more precise and accurate by removing consistent mutants and all but one of semantically duplicate mutants, as explained in [2]. Let N be the number of unique, inconsistent mutants generated by all operators, and k be the number of mutants killed. The coverage is k/N. We use this metric to compare operators in Section 3.4.

2.1 Applicability

Model checking, a vital part of the method, can be applied to specifications for large software systems, such as TCAS II [9].

[1]After [15], a test criterion is a decision about what properties of a specification must be exercised to constitute a thorough test.

To avoid the model checker's state space explosion problem, several approaches are used, such as abstraction, partial order reduction, and symmetry [10]. A reduction called finite focus was proposed to increase feasibility of model checking for test set generation [1]. Here, some finite number of states is mapped one-to-one to states in the reduced specification, while all other states are mapped to a single state.

2.2 Related Work on Specification Mutation

Gopal and Budd [16] applied a set of mutation operators to specifications given in predicate calculus form. The method relies on having a working implementation, as the program under test must be executed in order to generate test output. Woodward [24] investigated mutation operators for algebraic specifications. Weyuker et. al [23] proposed strategies for generating test data from the specifications represented by Boolean formulas and assessed their effectiveness using mutation analysis.

Fabbri et. al [12] devised a mutation model for finite state machines and used the mutation analysis criterion to evaluate the adequacy of the tests produced by standard finite state machine test sequence generation methods. Fabbri et. al [13] categorized mutation operators for different components of Statecharts specifications and provided strategies to abstract and incrementally test the components. Mutation analysis in the context of protocol specifications written in Estelle, an extended finite state machine formalism, was studied in [21].

3 Specification Mutation Operators

Ammann and Black defined some mutation operators, but did not consider the relative merits of the operators. We describe a set of mutation operators we developed for formal specifications together with their respective fault classes. We investigate the relationships between detection conditions for several fault classes analytically and compare the effectiveness of the mutation operators experimentally. A detailed description of the results presented in this Section can be found in [7].

3.1 Categories of Mutation Operators

Mutation categories should model potential faults [24]; therefore, it is important to recognize different types of faults. We design each mutation operator to uncover faults belonging to the corresponding fault class.

Some of these fault classes are related to the classes forming Kuhn's hierarchy. We use a term "simple expression" that closely corresponds to the Boolean variable in [18]. A simple expression is a Boolean expression that has no Boolean operators. For example, relational expressions and Boolean variables are simple expressions. (Other commonly used terms are "clause" and "condition").

Each fault class has a corresponding mutation operator. Applying a mutation operator gives rise to a fault in that class. For example, instances of the missing condition fault (MCF) class can be generated by a missing condition operator (MCO). Note that the abbreviation of the mutation operator ends in O, and the abbreviation of the corresponding fault class ends in F.

Although mutation operators are independent of any particular specification notation, here we present them for CTL specifications. Table 1 contains mutation operators for common fault classes and selected illustrative mutants generated from three formulas: the "`SPEC`" clause in Figure 1, the formula `AG (x & y → z)` (for ASO), and the formula `AG (WaterPres < 100)` (for RRO).

Operators and Example Mutants	
ORO	Operand Replacement
	AG (request → AF state = ready)
SNO	Simple Expression Negation
	AG (!request → AF state = busy)
ENO	Expression Negation
	AG (!(request → AF state = busy))
LRO	Logical Operator Replacement
	AG (request & AF state = busy)
RRO	Relational Operator Replacement
	AG (WaterPres <= 100)
MCO	Missing Condition
	AG AF state = busy
STO	Stuck-At
	AG (request → AF 1)
ASO	Associative Shift
	AG (x & (y → z))

Table 1. Mutation Operators and their Illustrative Mutants.

The function of some operators can be easily guessed from the name; we briefly explain what other, less obvious operators do. ORO replaces an operand by another syntactically legal operand. It does not replace a number with another number, since this may result in too many mutants. The current implementation of the operator handles two kinds of operands: state variables and symbolic constants. State variables may be of Boolean, scalar or integer type. The value of a scalar variable is drawn from a finite set of constants. An integer variable takes values from a finite range. An SMV specification may also contain symbolic constants defined by the user to represent integers.

MCO deletes simple expressions from conjunctions, disjunctions, and implications. STO replaces a simple expression with 0 and 1. ASO changes the association between variables, e.g., $x \to y_1y_2y_3$ is replaced with $(x \to y_1)y_2y_3$.

If the number of atoms (variables and constants) in a specification is V and the number of value references is R, ORO results in $O(V * R)$ mutants, whereas SNO, LRO, MCO, STO, ASO and RRO result in $O(R)$ mutants.

ORO$^+$ operator, a combination of ORO and RRO, generates a class of faults closely matching VRF in [18].

Additionally, we defined Simple Expression Replacement Operator (SRO) which replaces a simple expression by every other syntactically valid simple expression of atoms in the model. This operator generates a class of faults identical to VRF. SRO sometimes generates higher order mutants, so by Woodward's principle [24], it should not be used for test generation.

We analyzed the relationships between several fault classes and studied the mutation operators experimentally.

3.2 Analysis of Fault Classes

Our operators model fault classes similar to those analyzed in Kuhn [18]. By comparing the conditions under which different types of faults are detected, Kuhn derived a hierarchy of fault classes. We extended Kuhn's analysis and tied it to mutation operators.

The detection conditions for a predicate P are the conditions under which a change to P affects the value of P. A test detects an error if and only if a faulty predicate P' evaluates to a different value than the correct predicate P. To simplify analysis, we only considered specifications S with formulas in disjunctive normal form (DNF).

Let S_{FAULT} be the detection conditions for fault class $FAULT$. We discovered the following relationships:

$$\begin{tikzcd}
S_{SRF} \arrow[r] & S_{SNF} \arrow[r] & S_{ENF} \\
S_{STF} \arrow[r] \arrow[ur] & S_{MCF} &
\end{tikzcd}$$

Formal analysis is presented in [7]. Informally, to determine the detection conditions for an arbitrary fault in a particular fault class, an exclusive-or of an original specification and its faulty version is computed.

It follows from the relationships, for instance, that a test that detects a Simple Expression Replacement Fault (SRF) for a simple expression in a predicate, also detects a Simple Expression Negation Fault (SNF) for the same simple expression. Hence, SRO detects SNF. Also since ORO$^+$ can be considered as a practical approximation to SRO, ORO$^+$ is very likely to detect SNF.

3.3 Mutation Generator

To study the mutation operators and empirically confirm the theoretical results above, we developed an extensible tool for systematically making small syntactic changes to SMV specifications.

The tool uses portions of SMV code: the parser, abstract syntax tree (AST) manipulation routines and low level functionalities, such as dynamic memory allocation and manipulation of data structures (e.g., hash tables).

Mutation generator performs the following steps:

1. Parse a given SMV file and build a tree data structure in memory.

2. Process the tree to extract information necessary for performing mutations, e.g., collect information about types and domains of variables.

3. For each selected mutation operator, traverse the tree invoking the corresponding mutation routine. When the routine recognizes an opportunity for a mutation, it creates a mutant. The mutant is then written to a file.

Resulting individual mutations may be left in individual SMV files or written to a single file. The former yields a large number of files. The overhead of starting a new SMV process for each mutant is intolerable even for specifications of moderate size. Since SMV builds a state machine transition relation for a given input file only once and checks CTL formulas independently, using an option that writes mutations into a single file results in very efficient processing.

The tool allows us to selectively apply mutation operators. It can be extended to add new operators. In addition, the mutation generator optionally mutates state machines to generate tests which a correct implementation is supposed to fail.

The source code and documentation are available from the authors.

3.4 Empirical Comparison of Operators

We compared the mutation operators in terms of the number of test cases produced and the specification coverage. We ran experiments on several sample SMV specifications. Below we present the results for Safety Injection specification [6]. The results for other samples were similar. After reflection, this specification contains 22 CTL formulas and 5 variables, including a Boolean, 3 scalars, and an integer which takes values between 0 and 200, but is only compared with 2 different symbolic constants.

Out of a total of 730 mutants generated by applying all mutation operators to the specification (since SNO mutants

Operator	Mutants	Counter-examples	Unique Traces	Coverage
ORO^+	202	99	21	100%
ORO	130	63	17	94.2%
SNO	83	51	15	90.7%
ENO	144	104	15	90.7%
LRO	122	82	10	83.7%
RRO	72	36	10	50.0%
MCO	79	50	13	87.2%
STO	166	51	15	90.7%
ASO	17	17	5	47.7%

Table 2. Safety injection example results.

are a subset of ENO mutants, we did not include SNO mutants in the total), 86 were semantically unique, inconsistent. The method produced 21 unique test cases or traces.

We present details in Table 2. "Mutants" is the total number of mutants generated by each operator, including consistent and duplicate mutants. Next we give the number of counterexamples found in the SMV runs. "Unique traces" is the number of traces after duplicate traces and prefixes are removed. "Coverage" is the metric described in Section 2.

ORO^+ generates the largest number of mutants, but provides the same set of test cases as all the operators combined. Consequently, it has 100% coverage.

SNO provides second best coverage while generating significantly fewer mutants.

We define UT_{OPER} to be the set of unique traces generated by mutation operator $OPER$. For the Safety Injection specification, as well as several other examples, we found the following relationships between the sets of unique traces:

$$\boxed{\begin{array}{l} UT_{ORO} \supseteq UT_{SNO} \supseteq UT_{ENO} \\ \quad\;\; \rotatebox{-90}{$\supseteq$} \\ UT_{STO} \supseteq UT_{MCO} \end{array}}$$

These results agree with the analysis in Section 3.2. In particular, they support the idea that ORO is sufficient to detect faults in ORF, SNF, and ENF. Therefore, the Simple Expression Negation Operator (SNO) and Expression Negation Operator (ENO) are not needed if the Operand Replacement Operator (ORO) is used.

The above hierarchy is not guaranteed to hold for specifications with formulas not in disjunctive normal form. We discuss this in the following Section.

4 Open Questions

Based on our current understanding of the method and its challenges, we define the following research topics and questions.

4.1 How Does Form Influence Results?

Semantically equivalent specifications may be written in different ways. Since mutation analysis makes syntactic changes, the results may depend on what form the specification is in.

Form of Specifications

Kuhn's analytical technique applies to specifications in restricted form, i.e., with formulas in disjunctive normal form (DNF). Realistic specifications are generally not in DNF, and the mutants of a DNF representation are significantly different from the mutants of the original.

Consequently, if we apply mutation operators to the unaltered specification, the theoretical results do not strictly apply. We will empirically study the degree to which the test set generated from a specification with formulas in DNF differs from the test set produced from an original specification. To study this, we will mechanically convert specification formulas to DNF, then compare the test sets generated from original with those from the converted specifications.

Specification Languages

Although we use SMV, the method is not limited to any particular type of model checker. Most of the interest has been centered around two types of model checkers [22]: branching time model checkers for Computation Tree Logic (CTL) and linear time model checkers for the propositional Linear Temporal Logic (LTL). SPIN [17] is a popular LTL model checker. We plan to study whether our research is applicable to both CTL and LTL model checkers.

4.2 What is the Relation Between Operators?

This has different facets, in particular, what is the trade off between choosing some operators which produce more mutants, but give better coverage, and performing selective mutation without the most expensive operators, that is, choosing operators which produce far fewer mutants, but give slightly worse coverage. Using Kuhn's analytical technique, we found the subsumption relationship between several operators. Subsumed operator does not need to be used if the subsuming operator is applied. We will look for such relationships for other operators. We are also interested in discovering other analytical techniques for comparing mutation operators. Finally, are some operators better for different applications, sizes or forms of specifications? The latter was discussed above.

New Operators and Sets of Operators

Efficiency of the test generation and evaluation method is determined, in part, by the cost of mutation. We plan to extend our work on comparing mutation operators based on their coverage and the number of mutants generated. We found that ORO^{+} gives maximum coverage, and SNO gives very good coverage using far fewer mutants. We will look for other operators or sets of operators which provide high fault detection capabilities at reduced cost. We will extend the mutation generator program to apply a richer set of mutation operators to SMV specifications.

Experimental Base

The Safety Injection specification used in this paper has only five variables and a single module. However, many realistic specifications are composed of a number of modules and contain dozens of variables. To verify scalability of the method and applicability of the experimental results to realistic specifications, we plan to use larger specifications, such as the Flight Guidance System [20] and other specifications from industry. By improving the mutation generator to handle the general SMV syntax, we will extend the pool of specifications available for our experiments.

4.3 How Does the Method Compare with Others?

We want to compare the coverage of specification mutation analysis with existing methods to get some idea of the quality of this method.

An objective comparison of the specification-based mutation analysis with commonly accepted criteria is necessary. Our goal is to reduce the number of faults in the actual programs written from the formal specifications. Therefore, it is necessary to study usefulness of the tests generated from formal specifications for detecting bugs in the corresponding implementations.

Many coverage measures exist [25]. Branch coverage of generated tests for Cruise Control example was examined in [3]. Branch coverage (decision coverage) checks whether boolean expressions tested in control structures evaluated to both true and false.

We plan to investigate the program-based coverage of the tests using several coverage metrics, in particular, a practical variation of path coverage, such as length-n sub-path coverage which checks whether all sub-paths of length less than or equal to n in a program have been followed. Path coverage metric is powerful yet unrelated to mutation analysis, hence it is an important standard measure. Another possible measure is a fault-based coverage metric.

5 Conclusions

Standard mutation analysis is based on program source code. In contrast, a recent mutation analysis scheme uses model checkers to automatically generate complete test sets from formal specifications and to evaluate coverage of existing test sets. Using a model checker avoids the problem of equivalent mutants, since model checking is decidable, and takes advantage of 20 years of industrial model checking experience.

We described the specification-based mutation analysis method and reported our recent work: defining a set of specification mutation operators in the context of this method and comparing them based on their effectiveness and cost.

We posed three questions to confirm the practicality of this method: how does form influence results, what is the relation between mutation operators, and how does this method compare with others? We also outlined some of our research planned to address these questions.

References

[1] P. Ammann and P. E. Black. Abstracting formal specifications to generate software tests via model checking. In *Proceedings of the 18th Digital Avionics Systems Conference (DASC99)*, volume 2, page 10.A.6. IEEE, October 1999. Also NIST IR 6405.

[2] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248. IEEE Computer Society, November 1999. Also NIST IR 6403.

[3] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, Dec. 1998.

[4] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 280–292, Jan. 1996.

[5] C. Banphawatthanarak, B. H. Krogh, and K. Butts. Symbolic verification of executable control specifications. In *Proceedings of the Tenth IEEE International Symposium on Computer Aided Control System Design (jointly with the 1999 Conference on Control Applications)*, pages CACSD–581–586, Kohala Coast - Island of Hawai'i, Hawai'i, Aug 1999.

[6] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. Memorandum Report NRL/MR/5540-97-7999, U.S. Naval Research Laboratory, Washington, DC 20375, November 1997.

[7] P. E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *Proceedings of* 15^{th} *IEEE International Conference on Automated Software Engineering (ASE2000)*, September 2000. To be published.

[8] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN Workshop*, Rutgers, NJ, August 1996. Also WVU Technical Report #NASA-IVV-96-022.

[9] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498 – 520, July 1998.

[10] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[11] E. M. Clarke, Jr., E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[12] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Mutation analysis testing for finite state machines. In *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, pages 220–229, Monterey, CA, November 1994. IEEE.

[13] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proceedings of the Tenth International Symposium on Software Reliability Engineering*, pages 210–219, Boca Raton, Florida, November 1999. IEEE.

[14] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Toulouse, France, September 1999.

[15] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[16] A. Gopal and T. Budd. Program testing by specification mutation. Technical Report TR 83-17, University of Arizona, Nov. 1983.

[17] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[18] D. R. Kuhn. Fault classes and error detection in specification based testing. *ACM Transactions on Software Engineering Methodology*, 8(4), October 1999.

[19] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[20] S. P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Second Workshop on Formal Methods in Software Practice*, Clearwater Beach, FL, March 1998.

[21] R. L. Probert and F. Guo. Mutation testing of protocols: Principles and preliminary experimental results. In *Protocol Test Systems, III*, pages 57–76. Elsevier Science Publishers B.V. (North-Holland), 1991.

[22] W. C. Visser. *Efficient CTL* Model Checking Using Games and Automata*. Dissertation, The University of Manchester, June 1998.

[23] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[24] M. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *Software Engineering Journal*, pages 211–224, July 1993.

[25] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.