PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode

Elvira Albert
Complutense University of Madrid
elvira@sip.ucm.es

Miguel Gómez-Zamalloa
Complutense University of Madrid
mzamalloa@fdi.ucm.es

Germán Puebla
Technical University of Madrid
german@fi.upm.es

Abstract

PET is a prototype *P*artial *E*valuation-based *T*est case generation tool for a subset of Java bytecode programs. It performs white-box test generation by means of two consecutive *Partial Evaluations* (PE). The first PE decompiles the Java bytecode program into an equivalent CLP (Constraint Logic Programming) counterpart. The second PE generates a *test-case generator* from the CLP program. This generator captures interesting test coverage criteria and it is able to generate further test cases on demand. For the first PE, PET incorporates an existing tool which decompiles bytecode to CLP. The main contribution of this work is the implementation of the second PE and the proof of concept of the approach. This has required the development of a partial evaluator for CLP with appropriate control strategies to ensure the required coverage criteria and to generate test-case generators. PET can be downloaded as free software from its web site, where a repository of examples and a web interface are also provided. Though PET has to be extended to be applicable to larger programs, we argue that it provides some evidence that the approach can be of practical interest.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Symbolic execution; F.3.2 [*Logics and Meaning of Programs*]: Partial evaluation

General Terms Languages, Theory, Reliability

Keywords Testing, Test-case generation, Partial evaluation, Symbolic execution, Constraint Logic Programming

1. Introduction

One of the most successful techniques to-date for reasoning about program correctness and detecting bugs is systematic program testing. In testing, the *System Under Test* (SUT) is run on a series of *test cases* and the result computed by the SUT is compared with that expected. A *test suite* refers to a collection of test cases which are applied to a SUT. Though program testing is a relatively lightweight technique when compared to full formal verification, it nevertheless implies a significant cost. In order to keep it as low as possible, it is essential to select the test suite in such a way that a certain *coverage criterion* (see e.g., [16] for a survey) is achieved by using a minimal number of test cases. Such coverage criteria are heuristics which try to estimate how well the program is exercised by a test suite. Examples of coverage criteria are *statement coverage*, which requires that each line of the code is executed, *path coverage* which requires that every possible trace through a given part of the code is executed, etc.

Test Data Generation (TDG) can be done *dynamically* [2], by executing the SUT for concrete input values, or *statically*, where no knowledge about the input data is assumed. On another dimension, test data generation can be classified into *black box* approaches, where the internals of the SUT are ignored and program specifications are used to guide TDG, and *white-box* approaches, where the internals of the SUT are exploited for guiding the TDG process. The standard way of performing static white-box generation of test cases is to perform a *symbolic* execution of the program [8, 5, 11, 12, 14] whereby instead of on actual values, programs are executed on symbolic values, sometimes represented as *constraint variables*. Such constraints are accumulated as each branch of the execution tree is expanded. The constraints in feasible branches provide pre-conditions on the input data which guarantee that the corresponding branch will be executed at run-time. Concrete input values which satisfy the constraints can then be obtained. These values become the input data for a test case which can be used for running the program. Then, an *oracle*, i.e., the user or some (partial) specification, should be consulted in order to decide whether the actual output of is correct and to modify it otherwise. Then, a complete input-output pair can be stored as a test case.

In this work we present PET, a prototype tool for static white-box TDG of Java bytecode. Our tool works at the bytecode [9] level because it is common in Java applications to have access to the bytecode, often bundled in *jar* files, but not to the source code. This is even more so in commercial software and in mobile code. PET is based on the approach proposed in [1] and its main novelty is that the TDG process is based on *Partial Evaluation* [6] (PE) of *Constraint Logic Programs* [10] (CLP). PE is a well-known program transformation technique which specializes programs w.r.t. part of their input data. A unique feature of PET is that the test-case generators it produces can be used for generating further test-cases on demand without having to start the TDG process from scratch.

In its current form, PET has the following limitations: (1) It can only generate test data for numeric arguments (not objects nor arrays), (2) floating-point numbers are not handled, (3) static fields are not handled, and, (4) native code is not handled. As mentioned later in Section 4, we are are currently working towards the extensions to overcome such limitations.

2. Architecture of PET

Fig. 1 shows the overall architecture of PET. The dashed frames represent the two main phases of the process: the transformation of the bytecode into a CLP representation (**PE 1**); and the actual test

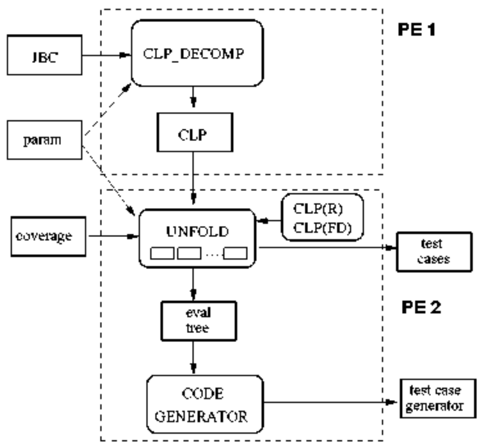

Figure 1. Architecture of PET

case generation (**PE 2**). Input and output of the system are depicted, respectively, on the left and the right: PET takes a Java bytecode program *JBC* and a description of the *coverage criterion*, and yields as output a set of *test cases* which guarantee that the selected coverage criterion is achieved and, optionally, a test case *generator*. There are several parameters, named *param* in the figure, which can be used for deciding which intermediate steps are viewed in the output. We now discuss the three steps in the TDG process.

CLP_ Decompilation. During **PE 1**, the incoming *JBC* is transformed into an equivalent *CLP* program by slightly adapting an existing decompiler [4]. In particular, CLP_DECOMP is an *interpretive decompiler* (i.e., based on the first Futamura projection [3]) which partially evaluates a JVM-interpreter written in Prolog w.r.t. an input bytecode and produces as a result a CLP program. The only modifications to this decompiler have been to make it accept methods as decompilation units, since it was applied on whole classes (or packages), and to replace in the output Prolog arithmetic with *constraints*. Let us consider method intExp in Fig. 2. The Java source is shown only for clarity. PET performs TDG from the bytecode, which is shown in Fig. 3 inside its control flow graph (CFG). Method parameters and local variables in the program are referenced by consecutive natural numbers starting from 0. Observe also the use of the operand stack in the bytecode, e.g., conditionals are performed against the value at the top of the stack. We refer to [9] for further details on the bytecode language.

Fig. 4 shows the decompiled version of the intExp method. It contains CLP(FD) constraints such as N #>=0, in SWI-Prolog syntax. Rules which correspond to method entries have two arguments which represent the input and output information. The first argument is a list of two elements. In turn, the first one is a list which contains the input parameters of the corresponding method (i.e., A and N) and the second one is the input heap HIn. The output parameter is a list with three elements, the return value Ret, the output heap HOut and the exceptional output behavior EFlag. We can observe that blocks in the CFG in Fig. 3 are represented by corresponding rules in the CLP program. The for loop has been converted into a recursion. The rule for loop_init corresponds to the block where the loop is initialized. Bytecode instructions are decompiled and translated to their corresponding operations in CLP; conditional statements are captured by introducing multiple rules with disjoint guards. For instance, the conditional ifge 12 at *pc* 1 results in two rules for predicate intExp: one for the case when n$\geq$0 and another one for n<0. Since we have explicit rules for exceptional executions, we can generate test-cases for them as well. Note that during decompilation we treat the heap as an abstract data type with a set of operations which manipulate it. For instance, this is the case of the atom new_(H1,C,R,H2) in the code above where H1 and H2 are the input and output heaps, respectively, C is a class identifier and R is a reference to the created object.

```
static int intExp(int a,int n){
   if (n < 0) // Exponent must be non-negative
      throw new ArithmeticException();
   else if ((a == 0) && (n == 0)) // 0 to 0 is undefined
      throw new ArithmeticException();
   else {
      int out = 1;
      for (;n >= 0;n--) out = out*a;
      return out; }
}
```

Figure 2. Source code of running example

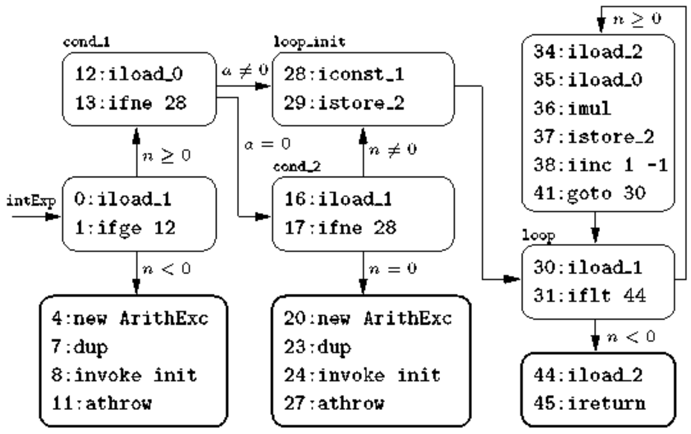

Figure 3. Input to PET: Bytecode of running example

Unfolding. The aim of this phase is to generate a *test-suite* which traverses as many different execution paths as possible. For this, and as discussed in Sect. 1, we will perform *symbolic execution*. A key advantage of the CLP decompiled programs w.r.t. their bytecode counterparts is that symbolic execution does not require to build a dedicated symbolic execution mechanism and we use standard execution. However, we need to supervise execution in order to guarantee termination while performing useful unfoldings. This is exactly the problem that *unfolding rules*, denoted *UNFOLD* in

```
intExp([[A,N],HIn],[Ret,HOut,EFlag]) :- N #>= 0,
    cond_1(A,N,HIn,Ret,HOut,EFlag).
intExp([[_A,N],HIn],[_Ret,HOut,exception(R)]) :- N #< 0,
    new_(HIn,'ArithException',R,H2),
    'ArithException.<init>()V'([[ref(R)],H2],[HOut,_]).

cond_1(A,N,H1,R,H2,ok) :- A#\=0, loop_init(A,N,H1,R,H2).
cond_1(0,N,H1,Ret,H2,EFlag):- cond_2(N,H1,Ret,H2,EFlag).

cond_2(N,H1,R,H2,ok) :- N#\=0, loop_init(0,N,H1,R,H2).
cond_2(0,H1,_Ret,H3,exception(R)) :-
    new_(H1,'ArithException',R,H2),
    'ArithException.<init>()V'([[ref(R)],H2],[H3,_]).

loop_init(A,N,H1,Ret,H1) :- loop(A,N,1,Ret).

loop(_A,N,Out,Out) :- N #< 0.
loop(A,N,Out,Ret) :-  N #>= 0, Out' #= Out*A, N' #= N-1,
    loop(A,N',Out',Ret).
```

Figure 4. Decompiled CLP Program obtained by PET

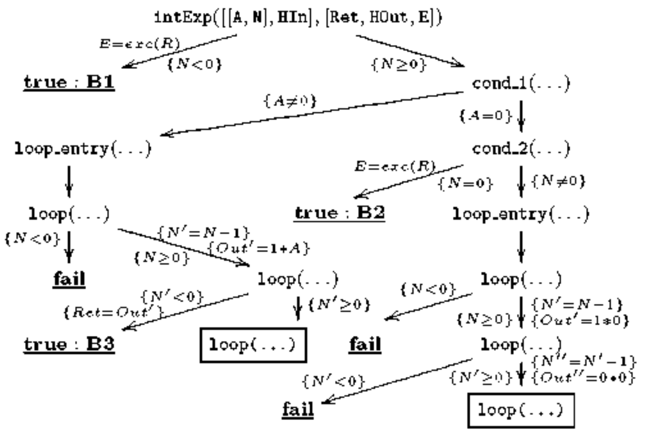

Figure 5. An evaluation tree generated by PET for `intExp/2`

the figure, used in partial evaluators of (C)LP, solve. In essence, partial evaluators are meta interpreters which given an atom evaluate it as determined by the so-called unfolding rule, obtaining an *evaluation tree*. Each non-failing branch in this tree corresponds to a computation path. This view of TDG as a PE problem, proposed in [1], has the important advantage that we can apply existing, powerful, unfolding rules developed in the context of PE. This is illustrated in Fig. 1 by small boxes which represent a bunch of unfolding strategies that can be plugged in the system. Currently, PET incorporates two unfolding rules: **level-**k, which limits the depth of the evaluation tree to at most k levels, and **block-**k, which ensures that the number of times each block is visited within each path does not exceed the given k.

Fig. 5 shows the evaluation tree built by PET when selecting **block-**k with k=2. I.e., the third time a rule is visited, the path is no longer expanded. In the example, PET executes the query `intExp([[A, N], HIn], [Ret, HOut, E])`. Along the execution, a constraint store on the program's variables is obtained which is used for inferring the conditions that the input values must satisfy for the execution to follow the corresponding path. Such conditions appear as labels on the arrows (e.g., $N \geq 0$, $A \neq 0$, etc.). We rely on an underlying constraint domain to handle the constraint store. CLP(FD) is currently used, which imposes an integer domain for the program variables. The tree contains both complete and incomplete branches. In turn, complete branches can be successful or failing, labeled respectively as **true** or **fail**. Incomplete branches have a framed atom as last element. They are no longer expanded because the unfolding rule prevent this. In particular, we can see that when an atom of the form `loop(...)` appears for the third time in the same branch, the branch is stopped. Note that **block-**k with k=1 will in general not visit all blocks in the CFG, since traversing the loop body of the `for` loop requires $k \geq 2$ in order to obtain a finished path.

Once an evaluation tree is computed, the constraint stores associated to successful branches can be used for obtaining associated test cases. For instance, the leftmost branch in the tree (the one which ends in an atom labeled as **B1**), captures the fact that for a negative value of N, the output is an exceptional behavior. This is associated to the constraints (N $<$ 0, E=exc(R)). Furthermore, our system allows providing a specific domain (e.g., N $\in$ [−10, 10]) and use the CLP(FD) predicate `labeling/2` to produce actual values in this domain compatible with the constraints. In order to get only one solution, `labeling/2` is called inside the meta-predicate `once/1`. For instance, for the above constraints, PET produces the input-output pair (⟨A=−10, N=−10⟩, E=exc(R)). For the path ending in **B2**, the constraints are (A=0, N=0, E=exc(R)). An input-output pair is simply (⟨A=0, N=0⟩, E=exc(R)). Finally, for the branch ending in label **B3**, the constraints obtained by PET are (N=0, Ret=A) and a possible input-output pair is (⟨A=−10, N=0⟩, Ret=−10). When confronted with this pair, the user or oracle should detect that `Ret` does not have the expected value, which indicates that there is a bug in the program, since `Ret` should take the value 0.

```
intExp([[_,N],H],[_,H',exc(1)]) :- N #< 0.          % B1
intExp([[0,0],H],[_,H',exc(2)]).                    % B2
intExp([[A,0],H],[Ret,H,ok]) :- A #\= 0, Ret = A. % B3
intExp([[0,N],H],[Ret,H,ok]) :- N #>= 1, N'' #= N-2,
                                loop(0,N'',0,Ret).
intExp([[A,N],H],[Ret,H,ok]) :- N #>= 1, A #\= 0,
                                N' #= N-1,
                                loop(A,N',A,Ret).
loop(_,N,Out,Out) :- N #< 0.
loop(A,N,Out,Ret) :- N #>= 0, Out' #= Out*A,
                     N' #= N-1, loop(A,N',Out',Ret).
```

Figure 6. Output of PET: Test-case generator

Code Generation. The final objective of partial evaluation is to generate optimized *residual* code. Thus, the unfolding rule discussed above can be complemented with a code generation phase and obtain a full partial evaluator (**PE 2** in Fig. 1). For instance, consider the successful branch labeled **B3** in Fig. 5. The code associated to this branch is a rule whose head is the original atom (applying the mgu's to it) and the body is made up by the constraints gathered along the path:

```
intExp([[A,0],H],[Ret,H,ok]) :- A #= 0, Ret = A.
```

As proposed in [1], the generation of a residual program composed by the rules associated to all non-failing branches in the evaluation tree returns a program which can be used as a *test-case generator* for obtaining further test-cases. In Fig. 6, we show a pretty printed test-case generator obtained by PET from the evaluation tree in Fig. 5. Basically, PET generates *constrained* rules which integrate the *store* of constraints associated to their corresponding branch, as shown above. The first three rules correspond to the three successful branches (**B1**, **B2** and **B3**) in Fig. 5, from which we obtained the three test-cases shown before (after calling `labeling/2`). The other two rules are obtained, as explained above, from the two incomplete branches which finish in a framed atom. The constraints in the different rules, in addition to accumulating the arithmetic operations performed in along the path, act as guards which avoid the execution of the alternative paths previously computed.

Thus, the output of PET is a program which is a *generator* of test-cases for larger values of k. The execution of this concrete generator will return on backtracking the (infinite) set of computation paths for the `intExp` program and their corresponding constraints. Interestingly, in order to generate test-cases for say, $k = 5$, instead of starting the process from scratch, we can partially evaluate the generator with $k = 3$ and obtain (more efficiently) the same set of test cases that we would obtain by partially evaluating the original CLP program for $k = 5$.

3. Web Site and Experimental Evaluation

PET is available for download as free software at the PET web site `http://costa.ls.fi.upm.es/pet`. In addition, a web interface makes it possible to use PET without having to install it. PET can be executed on bytecode programs provided as examples on the web site or by uploading them.

We now present some preliminary experiments which aim at illustrating the time taken by PET in order to perform TDG and

Bench	T_{j2p}	$k = 2$				$k = 5$			
		N	$\mathbf{T}_{tcg}$	$\mathbf{T}_{gen}$	**Total**	**N**	$\mathbf{T}_{tcg}$	$\mathbf{T}_{gen}$	**Total**
intExp	13.70	3	4.20	0.00	17.90	9	6.00	0.00	19.70
intFact	11.10	3	0.00	0.00	11.10	6	4.20	3.80	19.10
lcm	11.70	7	8.20	0.00	19.90	19	72.40	0.00	84.10
gcd	14.20	4	2.00	0.00	16.20	10	17.80	2.20	34.20
varNoRep	12.00	4	2.00	0.00	14.00	7	4.20	0.00	16.20
varRep	11.10	4	5.80	0.00	16.90	7	2.00	0.00	13.10
combNoRep	12.90	4	4.00	0.00	16.90	7	10.00	0.00	22.90
combRep	18.00	4	2.00	0.00	20.00	7	10.00	0.00	28.00
perm	12.10	3	0.00	0.00	12.10	6	4.00	2.00	18.10
fib	14.00	4	2.00	0.00	16.00	7	4.00	0.00	18.00

Table 1. Some Execution Statistics for PET

the number of test cases generated when using different criteria. Table 3 shows the times taken by the different phases performed by PET. All times are in milliseconds, and were obtained as the arithmetic mean of five runs on an Intel Core 2 Quad Q9300 at 2.5GHz with 1.95GB of RAM, running Linux 2.6.26 (Debian lenny). As benchmarks, we use a set of methods which perform different arithmetic calculations like the greatest-common-divisor, the least-common-multiple, the Fibonacci sequence, etc. They are all accessible through the web interface. Each row in the table corresponds to one benchmark. The second column $\mathbf{T}_{j2p}$ shows the times taken by **PE 1**, including parsing the corresponding `.class` files. The next four columns show different data about **PE 2** using as coverage criterion the **block-**k with $k = 2$. In particular, column **N** shows the number of test-cases obtained, columns $\mathbf{T}_{tcg}$ and $\mathbf{T}_{gen}$ show, respectively, the times taken by the generation of the test-cases, i.e. by the unfolding process, and the generation of the test-case generator, while column **Total** show the total time taken by PET. The last four columns show the same data as before, but using PET with **block-**k being $k = 5$. Though we need to experiment with larger programs, the execution times of PET are reasonable.

4. Conclusions and Future Work

As mentioned in Sect. 1, the standard approach to static white-box TDG is to perform symbolic execution. If the language considered is Java bytecode, this requires developing a symbolic JVM machine which integrates appropriate constraints solvers (e.g., [12]). This requires non-trivial extensions w.r.t. a JVM: (1) it needs to execute the bytecode symbolically, and (2) it must be able to explore non-deterministic executions, as without exact knowledge about the input data, execution may follow more than one path. Such multiple paths can be traversed via backtracking or by explicitly handling sets of paths. The approach taken in [12] is based on a backtracking mechanism which is essentially the same as in Prolog. The fact that the behavior of bytecode programs is captured as CLP programs greatly facilitates symbolic execution, since we can use the underlying execution mechanism directly, without the need of devising a symbolic JVM. Furthermore, the process of supervising execution to avoid non-termination can be formalized as a PE problem.

We argue that PET has several interesting features: (i) It is *generic*. Our tool can work with other imperative languages, provided that a CLP decompiler (possibly, but not necessarily based on PE) for them is available. In particular, once the CLP decompilation is done, the language features are abstracted away and, the whole part related to TDG generation is totally *language independent*. This avoids the difficulties of explicitly dealing with recursion, procedure calls, dynamic memory, exceptions, etc. that symbolic abstract machines typically face. (ii) It is *flexible*, as different coverage criteria can be easily incorporated to our tool just by adding the appropriate unfolding rule to the partial evaluator. (iii) It is *incremental*, since our tool can extend test suites with larger values of k starting from previously obtained test-case generators.

We plan to improve PET in several directions. As already mentioned, our system is not able to deal with nun-numeric input arguments. To overcome this, we expect to be able to model the heap using constraints, as well as along the lines of the recent proposal in [13]. Regarding handling floating point numbers, we are working on the integration of other constraint domains such as CLP(Q) and CLP(R). We will also consider generalized symbolic execution, as done for model checking and testing [7, 15], which performs symbolic execution on dynamically allocated structures (e.g., lists and trees). Another challenge that we plan to investigate is the generation of test-cases for programs which use native code, and not only pure bytecode. We also believe that our approach could be easily extended with support for generating parameterized tests [14].

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-231620 *HATS* project, by the MEC under the TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, by the UCM-BSCH-GR58/08-910502 (GPD-UCM), and the CAM under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] E. Albert, M. Gómez-Zamalloa, and G. Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *LOPSTR'08*, number 5438 in LNCS. Springer-Verlag, March 2009.

[2] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[3] Y. Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[4] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51:1409–1427, October 2009.

[5] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic*, 2000.

[6] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.

[7] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.

[8] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[10] Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[11] C. Meudec. Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.

[12] R. A. Müller, C. Lembeck, and H. Kuchen. A symbolic java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, pages 365–371, 2004.

[13] T. Schrijvers, F. Degrave, and W. Vanhoof. Towards a framework for constraint-based test case generation. In *LOPSTR'09*, 2009.

[14] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.

[15] W. Visser, C.S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA'04*. ACM, 2004.

[16] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.