Constraint Satisfaction for Test Program Generation

(Preliminary Version)

Daniel Lewin* Laurent Fournier Moshe Levinger Evgeny Roytman Gil Shurek
IBM Israel Science and Technology, Haifa Research Laboratory
MATAM, Haifa 31905, Israel
*dlewin@haifasc3.vnet.ibm.com

Abstract

A central problem in automatic test generation is solving constraints for memory access generation. A framework, and an algorithm that has been implemented in the Model-Based Test-Generator are described. This generic algorithm allows flexibility in modeling new addressing modes with which memory accesses are generated. The algorithm currently handles address constraint satisfaction for complex addressing modes in the PowerPC, x86, and other architectures.

1. Introduction

The goal of processor verification is to ensure equivalence of a processor and its architectural specification. In practice, design verification is carried out by simulating a relatively small subset of selected test programs. These are run through the design simulation model, and the results are compared with the output predicted by the architecture simulation model.

A recent approach to test generation is Model-Based Test-Generation (cf. [LMA94]). The MBTG is a generic, architecture oblivious test generator. At the heart of the MBTG system lies a formal model of the architecture. A central problem in automatic test generation is solving constraints for memory access generation. Constraint satisfaction is a difficult problem that arises in test case generation both for hardware verification (cf. [CI92]) and for software testing (cf. [DO91]). This paper describes a generic solution for memory access generation implemented in the MBTG.

The subsystems that control memory accesses comprise a substantial part of processor design complexity. Therefore, the need to diligently verify the implementation of these subsystems motivates much of test generation methods. In order to effectively probe mechanisms such as caches and virtual translation, the addresses of memory accesses are biased to cause interesting events. Such events are: cache hit/miss, page fault, memory partition crossing, or any combination of these. Therefore, a test generator must have the ability to generate memory accesses that yield such events. In addition, a test engineer may supply a specific scenario for a test that includes further restrictions on memory accesses. A test generation system must be capable of satisfying such restrictions.

Access to memory in processors is generally carried out through a number of addressing modes. The complexity and power of these addressing modes varies greatly from processor to processor. For example, the x86 (cf. [INTEL]) and VAX (cf. [DEC]) architectures offer a large number of powerful addressing modes, while PowerPC architectures (cf. [MSSW84]) generally have a more modest set of simple modes. A typical addressing mode specifies how an address is computed from certain resources which are accessible to the processor (e.g. registers, memory, immediate values).

2. Generating Memory Accesses

Addressing modes in the MBTG are modeled by an ***address expression***, which is a simple algebraic expression over a set of variables. Each variable represents a subset of processor resources. An address expression, where each variable has been substituted by a processor resource, represents how an address is computed from the resources that appear in the expression. For example, the following expression, taken from the x86 architecture, makes use of two register variables (BASE and INDEX), and three field variables (SCALE, DISP, and IB):

$$EA = (BASE) + (INDEX) \times 2^{SCALE} + DISP + (IB/2^5) \times 2^2$$

The MBTG generates tests dynamically by a generation-simulation cycle for each instruction. At the beginning of the test all the processor resources are *free*, meaning that the generator can initialize resources in any way desired. As the test progresses, the processor resources gradually become *occupied* as a consequence of the generation-simulation cycle of previous instructions. The *Resource State* is defined to be the status of all processor resources at a given point during the generation process. A resource on the resource state is labeled as free, occupied with a value or as partially occupied with a value.

While generating a memory access at a certain stage in a test, the set of addresses that the MBTG may choose from is constrained by three major factors:

1. Constraints on the set of addressing modes that may be employed during a particular access. Such a restriction could be a result of the type of instruction that is being generated or of the current running mode of the processor.
2. Constraints on the address itself. These constraints either originate from biasing on the address that is to be generated (cache events, reusage of memory addresses, etc.) or from restrictions on the range of permissible addresses.
3. Constraints on the resources that take part in various addressing modes. The first, and most important type of resource constraint, stems from the fact that a value assigned to a particular resource may not contradict the current *Resource State*. The second type of resource constraint originates from biasing demands on resources.

Example: Consider the following address expression: *(BASE) + (INDEX) +DISP*. Assume that: BASE is the register R3, INDEX may be any one of the registers R1,R2, and DISP is a 2-bit immediate field which is a positive displacement. Generation and simulation have proceeded until a point where the following holds: (R1) = 10, (R2) = 15, and (R3) is constrained to be in the set {4,7}. DISP may be any value in the set {0,1,2,3}. The set of addresses that may be reached using the above expression is then {14,15,...,24,25}. Having an address biasing demand that the address be 4-byte aligned limits the set of addresses to {16,20,24}. There are four possible combinations of resources and values that give an address in the above set: 4+10+2 = 16; 4+15+1 = 20; 7+10+3 = 20; 7+15+2 = 24. For example, 4+10+2=16 corresponds to the choices (R3) = 4, INDEX = R1, DISP = 2.

3. The Address Constraint Problem

The general framework of address constraint satisfaction is formally described in the following section.

A ***resource constraint*** is a pair *(Resource ID, Values)* where *Resource ID* is the name of a processor resource (e.g. a register, a memory location), and *Values* is a list of values.

A ***variable constraint*** is a list of resource constraints.

An ***expression constraint*** is a variable constraint for each variable in an expression.

An ***assignment*** from an expression constraint assigns to each variable x in an expression a resource R, and a value V, such that R appears in a resource constraint for x, and V appears in the *Values* of R. An assignment may not provide two different values for the same resource.

A combination of an address expression and an assignment represents an address. This address is obtained by evaluating the expression with the assignment.

Using these terms the ***Address Constraint Problem*** is defined as follows:

Given an address expression E, an expression constraint C, and a set of addresses B; output a pair (A,S) where A is an address, and S is an assignment from C such that:

1. *A is an address from the set B.*
2. *A is the result of evaluating E with the assignment S.*
3. *The output is chosen from a uniform distribution over all outputs that satisfy 1, and 2.*

The Address Constraint Problem is NP-Complete as it is obviously in NP, and is NP-Hard by a simple reduction from the SUBSET SUM problem (cf. [GJ79]). In spite of its apparent difficulty, a powerful test generator must contain a solution for a variant of this problem. The rest of this paper describes an algorithm for an Address Constraint Solver that has been implemented in the MBTG.

The Address Constraint Solver algorithm was designed to solve a particular version of the more general Address Constraint Problem. The set of address expressions is restricted to a subset which is powerful enough to encompass the various addressing modes used by processors. The core of this subset is generated by the following simplified grammar:

$$S \rightarrow S + S | S - S | S \times 2^k | S / 2^k | Resource$$

Furthermore, it is assumed that the number of algebraic operations in address expressions is relatively small. These two restrictions allow us to introduce a practical solution which successfully handles complex memory access generation

4. The Algorithm

The data structure used to represent constraints is a *mask-list*. Each mask in a mask-list represents a set of values which are described by a bit-pattern consisting of 0's 1's, and x's (where x stands for "0 or 1"). A mask-list represents the set of values that is the union of the values represented by each of its masks. The sets of addresses, that represent both legal address ranges, and address biasing, generally lend themselves to compact representation by mask-lists. Resource constraints are also easily represented mask-lists.

A number of mathematical operations on mask lists are defined: addition, subtraction, intersection, multiplication by a power of two, and division by a power of two. For mask lists *A* and *B*, the addition is defined as the mask list that represents the values that can be obtained by adding one of the values represented by *A* with one of the values represented by *B*. The operations subtraction, multiplication, and division are likewise defined. The intersection of two mask lists *A* and *B*, is a list that represents the values that are in both *A* and *B*.

Efficient algorithms, both in time and space, have been implemented for each of these elementary operations on mask-lists. The Address Constraint Solver algorithm described here is based on these mask-list operations.

The algorithm does not give a satisfactory solution for the case where a ***given*** *non-occupied* resource appears more than once in the expression. One solution to this problem using the current scheme involves disposing duplicate resources in the expression by bringing them together. Another solution is to restrict variable constraints to include only disjoint resources. The latter, and less powerful solution is currently implemented in the MBTG.

The idea behind the algorithm is to create an equation from the address expression and a new variable which represents the address, and then to find a random solution to this equation. The solution is found by eliminating a variable at each stage. Elimination is carried out in two phases. First the constraints that the equation induces on the variable to be eliminated are computed. Then, a random value for this variable that conforms both to these constraints and to the variable constraints is chosen.

The input to the algorithm is an expression *E*, an expression constraint *C*, and a set of addresses *B*. The algorithm is described step-by-step as follows:

1. Create a new expression *E' = E - b* where *b* is a new variable.
2. Augment the expression constraint *C* with the set of addresses *B*, as values for the new variable *b*. The new expression constraint is denoted *C'*.
3. Create an equation *E' = 0*.
4. Find an assignment for *E'* which gives a random solution to this equation.
5. The value assigned to the new variable *b* is the address *A* which is output.
6. The assignment without the new variable *b* makes up the assignment *S* which is output.

The random solution to the equation *E'= 0* in step 4, is found by repeating the following steps until there are no more variables to be eliminated:

1. Choose at random a variable *V* to be eliminated.
2. Compute the set of values that the equation induces on the variable *V*. This is done by carrying out the algebraic steps that would be taken to isolate, on the left side of the equation, the variable *V*. Whenever an operation is carried out, compute by elementary mask operations, the set of masks induced on the right side of the equation by the operation and the expression constraint *C'*.
3. The equation is now in the form *V* = *Q*. Where *Q* has associated with it a mask-list that was computed in step 2. Choose a random value *z* which belongs to the intersection of this mask-list, and the variable constraint of *V* from *C'*. Note that *z* belongs to the resource constraint of some resource *R*, on the variable *V*. If the above intersection is empty, then there is no solution.
4. Update all resource constraints for the resource *R* to contain the single value *z*.
5. Make the pair *(R,z)* the assignment for the variable *V*.

The implementation of the algorithm is described in Appendix A.

5. Results

An algorithm for address constraint satisfaction has been described. Elementary operations on mask-lists play a central role in this algorithm. The algorithm has been implemented in the MBTG system, and successfully handles address constraint satisfaction for a number of architectures. The flexibility of this algorithm enhances the capability of the MBTG to rapidly provide automatic test generation for new architectures.

Acknowledgments

We would like to thank Yossi Malka, and Moshe Molcho for their involvement in this project, and for many useful discussions on constraint solving.

Appendix A: Implementation

Expressions in the MBTG are represented by binary expression trees. Internal nodes are operations, and leafs are variables. All constraints are represented by mask-lists. Following, is pseudo-code for the algorithm implemented:

```
SOLVE(Root, Address_Set)
BEGIN
  Create new tree for expression E - b with the new '-' node as NewRoot.
  Place Address_Set on the new leaf 'b' as a variable constraint.
  WHILE there are still free variables
  BEGIN
    Choose at random a free variable V.
    Create a mask list Zero that contains only the zero-mask.
    ELIMINATE(NewRoot, Zero, V)
    IF elimination failed THEN
      return failure
  END
  Output the value on leaf 'b' as the chosen address A.
  Output the resource-value pair that is on each resource leaf as the assignment S.
END

ELIMINATE(Node, Masks, Variable)
BEGIN
  IF Node is a leaf node THEN
  BEGIN
    node_masks = union of all mask lists in the variable constraints on Node.
    intersected_masks = node_masks ∩ Masks
    IF intersected_masks = ∅ THEN
      return failure
    ELSE /* intersection not empty */
    BEGIN
      Choose a random value z that conforms to intersected_masks
      Choose at random one of the resources R from the variable constraints on Node, whose value list contains z.
      Update the variable constraints on Node to be (R, z)
      Update all other resource constraints for the resource R to be the pair (R,z)
    END
  END
  ELSE /* node is internal node */
  BEGIN
    IF Variable is in left sub-tree THEN
    BEGIN
      right_masks = GSEC(Node->right)
      Let inverse_op be the inverse operation to the operation on Node
      next_masks = inverse_op(masks, right_masks)
      ELIMINATE(Node->left, next_masks, Variable)
    END
    ELSE /* variable is in right sub-tree */
    BEGIN
      /*
      similar to the above code, where "left" and "right" are interchanged.
      */
    END
  END
END

GSEC(Node) /* Get Subexpression Constraints */
BEGIN
  IF Node is not a leaf node THEN
  BEGIN
    left_masks = GSEC(Node->left)
    right_masks = GSEC(Node->right)
    Let op be the operation on Node
    return op(left_masks, right_masks)
  END
  ELSE /* Node is leaf */
    return union of all mask lists in the variable constraints on Node.
END
```

References

[LMA94] Y. Lichtenstein, Y. Malka and A. Aharon, “Model-Based Test Generation for Processor Design Verification”, Innovative Applications of Artificial Intelligence, AAAI Press, 1994.

[INTEL] Intel, *Pentium Processor User’s Manual, Volume 3: Architecture and Programming Manual,* 1993.

[MSSW94] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture*. Morgan Kaufmann, 1994.

[DEC] *Digital Technical J.* 4 (March), Hudson, Mass, 1987.

[GJ79] M. Garey and D. Johnson, *Computers and Intractability A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[CI92] A.K. Chandra, V.S. Iyengar, “Constraint Solving for Test Case Generation”, Proceedings of ICCD-92, Cambridge Mass. 1992.

[DO91] R.A. DeMillo, A.J. Offutt, “Constraint-Based Automatic Test Data Generation”, IEEE Trans on Software Engineering, Vol. 17 No. 9, Sept. 1991.