ABSTRACT EXECUTION OF PROGRAMS

J. SARBÓ

Department of Measurement and Instrumentation Engineering,
Technical University H-1521, Budapest

Received June 15, 1985
Presented by Prof. D. L. Schnell

Summary

Compilation time analysis of programs is usually incomplete. One of the basic methods for static determination of the program's dynamic properties is symbolic execution.

Symbolic execution still fails to satisfy practical requirements, mainly because of the high execution time and memory requirement, theorem proving and program termination problems.

In this paper new methods are presented which can make symbolic execution applicable in everyday work, e.g. in programming microprocessor equipment.

Introduction

Compilation time analysis of programs is usually incomplete. One of the basic methods for static determination of the program's dynamic properties is symbolic execution [6]. Some experimental systems based on symbolic execution were developed in the past [1], [4], [5]. Abstract program evaluation is also considered to be a technique of symbolic execution, where the program is executed with abstract values of program variables.

Symbolic execution still fails to satisfy practical requirements, mainly because of the high execution time and memory requirement, theorem proving and program termination problems.

In this paper new methods are presented which can make symbolic execution applicable in everyday work, e.g. in programming microprocessor equipment. The technique developed for symbolic execution is in many ways similar to abstract program evaluation, but has also new elements. For application of the methods, the SIMEX — static program analysis system was developed by the author. Important elements of the system are available and it appears to be useful in practice.

Basic characteristics of the methods

— The program is decomposed to parts based on the objects of the abstract data structure. The obtained program parts are executable, their static analysis can be effected independently.

— Program variables are modelled with abstract values, which can be represented with finite expressions. The instructions of the program are actually executed on these values.
— In program branches the information obtained in the determination of the branch predicate is propagated backward through the actual program path, by inverse execution of the instructions. This technique of symbolic execution simultaneously applies forward and backward execution strategies.

Formal semantics of program errors

The program is a formal description, its meaning is defined mainly by abstraction. It is known that each level of abstraction defines different semantics of the program. Abstraction may affect both instructions and data elements.

In most programming languages instruction abstraction does not generate new structural properties, elementary and composed objects do not differ from a structural point of view. E.g. wherever an instruction is accepted, a subroutine can be also used. On the other hand, attributes generated by data abstraction are usually different from those of the elementary object.

One part of program errors is related to the attribute 'generation' ability of data abstraction, and can be considered as 'attribute-violation'. In execution time the program — for efficiency reasons — is mostly stating without questioning. In principle this is not correct because instruction execution should always be preceeded by a check on whether its execution is safe or not.

The analysis of program semantics requires some formal definition, which describes the potential attributes of the program. There are many kinds of such systems which differ mainly in the subject and/or elaboration of the applied specification (e.g. algebraic, axiomatic definition etc.). It is however questionable whether one can specify all significant properties of a data structure. The specification is called useful redundancy if it is part of the program to be compiled. If the specification relates only to the programming language elements a self-consistency check of the program may be obtained. Our aim is to define a system which is capable for the most detailed semantic analysis possible, but requires minimal useful redundancy.

The aim of program analysis is to locate program errors. A program error is considered to be an ordered sequence of events that in run-time can cause undesired effects (defects). The error model is defined as a set of errors. Consequently, a program analyser consists of two main parts: one, that follows the specified events and the other that discriminates the erroneous sequences of these events.

Usually, errors can be classed into syntactic or semantic cathegories. It can be shown that according to our definition all errors of a system, analysable by some automaton — the semantic ones, too — are detected after all as syntactic errors.

Symbolic execution

Symbolic execution is a dataflow analysis method, that analyses progam behaviour by monitoring its actions on the symbolic input data. In this process the performed manipulations are represented as algebraic expressions over the input data [3].

Symbolic execution system

Symbolic execution analyses distinct program paths. In general programs contain an infinite number of paths. From the execution of a path (P_H), the symbolic value of the program variables ($V[P_H]$) and of the path condition ($PC[P_H]$) are gained. A path is executable if the corresponding path condition is consistent. Such a path is called a possible run of the program.

Several symbolic execution systems have been developed [1], [2], [5], using either: forward expansion or backward substitution implementation techniques.

In the forward expansion approach symbolic expressions are built as each statement in the path is encountered.

The backward substitution technique starts at the single exit point of the program and develops $V[P_H]$ by inverse execution of program statements of P_H.

Problems of application

There are three aspects that restrict the applicability of symbolic execution systems: the bulk of information developed; PC consistency determination, and termination (loop analysis).

Abstract program models

In this part we shall present a program model appropriate for applying symbolic execution. Our starting points are the abstract data structure (ADS — program specification) and the program code. We are not concerned here with program specification and implementation issues.

Fundamental ideas of program development are: virtual machine and abstract program. Both models contribute to the better understanding of the task of the program, but they are inappropriate as operational models of the program. The program interpretation (e.g. analysis, execution) model must be operational. Programpart is a new concept for representing this model.

The concept of data structure

Program manipulations relate to variables. Accordingly, objects of input/output operations are also considered to be program variables. Programming languages usually define elementary and a few 'built-in' composed data objects. Moreover, construction operations are also provided to develop composed data elements. In execution time attributes, structures of composed data are defined exclusively by access algorithms. There is a one-to-one mapping between composed data objects and their access algorithms.

The representation of a composed object in terms of elementary data (its allocation) is called static data structure.

Data structure is realized by: static data structure and the access algorithms interpreted on it.

Relation of ADS and the access algorithms

Abstract data structure (ADS) as a program specification tool describes the logical structure of the composed data objects. In the program, however, not all of these objects are realized, because some objects are introduced only for abstraction reasons.

Abstract access data structure (AADS)

AADS is a part of the abstract data structure. All objects of AADS have at least one access algorithm in the program. Consequently AADS can be derived from ADS by fusing each o_j with its immediate predecessor (o_i) if there is no occurrence of any access algorithm related to o_j. In the same time the (o_i, o_j) edge is also deleted.

Algorithm of program decomposition

First we examine how the programpart corresponding to o_1 — an immediate successor of the root of AADS (ξ) — can be derived (o_1 is selected from the set of objects nearest to ξ. The distance of an object from ξ is measured by the number of edges of the path connecting them).

Generation of the programpart of o_1

We take the program code and delete all instructions, except: the allocation definition of static data structure; the access algorithms corresponding to o_1; all activations (calls) of these routines; program control instructions (e.g. jump, call etc.).

The resulting programpart may refer to variables (e.g. CPU registers) that are assigned or used by some other, now omitted part of the program. This kind of reference is considered to be abstract *i/o* operation to variables. The result of an abstract input is always the symbolic 'arbitrary' value, abstract output operation has no effect.

The resulting programpart — under the restrictions mentioned above — is operational, it can be used in analysing the semantics of the selected object (o_1). Because of the partitioning method the analysis reports also potential errors, besides actual ones.

The concept of path-break

Having developed the programpart of o_1, we now concentrate on the derivation of the programpart relating to an object which has at least one predecessor other than ξ. We show the steps of the algorithm for o_2, an immediate successor of o_1.

It is obvious that the programpart of o_2 must include the programpart of o_1 too, since o_2 (the corresponding access algorithms) can operate only under the control of o_1. However, in the analysis of the o_2 programpart only those activities of o_1 are of interest, which develop different copies of o_2 — according to some type definition — in the o_2 accessing points of the path under analysis, that still have not been encountered there. In these points of the path all other runs of the o_1 programpart may be eliminated, by breaking off the execution of the actual program-path.

Hierarchy of programparts

For any object (o_i) of AADS the corresponding programpart can be derived as follows:

Definition

A selector-path (p) is a finite, ordered sequence of selectors:

$$p=\langle s_1, s_2, \ldots, s_n\rangle \quad \text{and} \quad p(\check{\xi})=s_n^{\circ}s_{n-1}^{\circ} \ldots {}^{\circ}s_2^{\circ}s_1(\check{\xi})$$

where s_i is a selector (edge) of AADS. The objects passed by p are $o_1, o_2, \ldots, o_n$, where $s_1(\xi)=o_1$, $s_2^0 s_1(\xi)=o_2, \ldots, p(\xi)=o_n$.

Programpart of o_n is defined by the static data structure, the access algorithms of $o_1, o_2, \ldots, o_n$ and all their activations. If the number of possible different types of o_i $(1 \leqq i \leqq n)$ is $t(o_i)$, then in the analysis of the o_n programpart one has to take into consideration only the program paths producing $t(o_1)$, $t(o_2)$, ... $t(o_n)$ and different copies of $o_1, o_2, \ldots, o_n$, respectively.

Finite model of program semantics

In this part a model of program semantics is introduced, where the formal meaning of the program is determined by a finite number of functions and their values. This definition is required because we wish to develop a symbolic execution system of programparts, where the size of information gained is limited. That is, only a certain part of the information corresponding to the symbolic expression of program variables, branch predicates is utilized in the semantic check.

Static analysis is performed on the basis of primitive semantics of the language. In the following, the language is supposed to be of assembly level, however, this does not restrict generality. Assembly languages play an important part in the programming of microprocessor devices chiefly for efficiency reasons.

In connection with the semantic program check first we analyse what properties of the program can be deduced from the elementary instructions of the language (of the processor). Program semantics can be expressed by attributes of variables that the analyser recognizes in the process of semantic check.

Formal meaning of elementary data objects

On instruction level the processor operates as a bit or bit sequence processor. Sometimes complex instructions are also provided but in general their function can be decomposed to primitive operations. The instruction, or elementary data structure has the following parts:

— access algorithm (elementary operation);

— static data structure (allocation of the operand (s));

— control structure (a graph containing a single node).

The program property set is determined by the operational semantics of instructions (functions over bits or bits sequences) and information of actual operands' value (again functions over the same domain). E.g.:

bit information: bit value is 0, 1, undefined, or arbitrary;

bit sequence information: the byte is nonzero, is of odd parity.

Generation of the property set of a program

Based on the semantics of elementary data structure one can deduce the attributes of a program comprising more than one instruction. Program semantics is obtained by getting the elementary data structures operated and by evaluation of functions related to composed objects. E.g.: information of composed data: type of the object, identification of access etc.

Functions determining the properties of a composed object are connected to variables. Thus a program variable may possess more than one meaning.

Consequences

According to the definition above, one can derive an actual property set of the program by the evaluation of functions. This method is applied in the realization of the analyser. Based on the events describing the elements of the error model the attributes of the program variables to be followed are determined. These attributes are coded and attached to the program variables (extending both their representation and semantics). Furthermore, formal semantics of the instructions of the program is given regarding the above described, extended (abstract) program variables.

Backward information propagation

Symbolic execution produces algebraic expressions for both $V[P_H]$ and $PC[P_H]$. In our method these expressions are partly evaluated and finite information is stored together with the variables.

E.g., if a branch predicate does not have a constant value (True or False) the abstract program state must be checkpointed. Execution may be continued with one possible value of the given branch predicate. Information used in the evaluation of this possible value can be propagated backward on the path by inverse execution of the instructions. As a result some variables get new semantics that can be used later.

In general, the information obtained in the evaluation of checkpoint causing variable(s) can be propagated backward. This method allows the simultaneous application of forward and backward symbolic execution strategies.

Example.

```
ld      A, input          ;A:=xxxx.xxxx 'arbitrary value'
ld      B, A
and     A, ↑B0011.1111    ;A:=00xx.xxxx
jp, eq  label1            ;Z=1 if equal to 0
Z=0 ↙  ↘ Z=1
...                       ;this part does not modify B
        tst     B:3
        jp, eq  label2
        Z=0 ↙  ↘ Z=1
        ...     ...
```

Here the first branch predicate cannot be evaluated, a possible value for Z must be chosen:

— Let us choose Z=1, in this case, in the previous 'and' operation operand A must have had 0 value on those bits which are set in the second operand (↑ B00111111), and consequently the bit information of A must have been A=xx000000. Continuing the execution, when control passes this point again (in the exhausting of graph paths), the other possible value (Z=0) has to be chosen. In this case 'nonzero (A)' can be deduced.

Due to the information propagation the second branch predicate becomes constant.

Application of symbolic execution

For the efficient application of symbolic execution a two-level program analysis method was developed which consists of large and small program analyses (LPA and SPA, respectively).

A program is considered 'large' if the structural relations of data can (or may) be considered. This distinction is merely a technical one, in certain cases large program analysis may be omitted.

In the first phase (large program analysis) the program is partitioned to programparts, which in the second phase (small program analysis) are individually checked.

Small program analyser

Programs contain the less non constant information in their compiled and linked, i.e. memory image form. E.g. the *vec1* [*n*] access of the source program can be treated only symbolically, even if the address of *vec1* and the value of *n* are constants. Basic purpose of SPA is the analysis of all program

paths according to a detailed error model. This can be done also with the memory image form of the program if input data are regarded to have a symbolic (abstract) value.

Abstract program execution needs the operational semantics definition of elementary (e.g. processor) instructions, the function definitions of symbolic information and the representation of their values. Operational semantics is most easily given by using a processor simulator program. The simulator is special in the sense, that each instruction is extended with error analyser functions.

Large program analyser

Large program analyser — using the formal definition of ADS — generates programparts, which are then checked by SPA. Technically an access algorithm and a data object composed of program variables contacts, when in the access algorithm the absolute (physical) address is evaluated. Accordingly the information of a composed object is connected to the program variable which stores this address. That is how a variable can also play the role of the abstract data of a given programpart.

To simplify the semantic check of a programpart type definition is heavily used. Path-break in LPA is effected by a return to the last checkpoint, where the next possible value of the given variable(s) is evaluated.

The applied error model

— Data structure validation
  In the static semantic check all possible program runs (paths) are analysed. Fundamentally, a structure check is performed, i.e. (using VDL terminology) an access is reported as illegal if it would apply a selector to the data object, which produces the result of an empty object (Ω).
— Variable access check
  The well known analysis of live/dead variables can be easily accomplished, too.
— Compactness validation

If there are several access algorithms (e.g. insert, search etc.) of a composed object (e.g. symbol table), then after starting the execution of one of these algorithms no other operation can use the object to be made access to until the algorithm ends. The access algorithms corresponding to a given data element must be primitive operations from the point of view of the object.

Program termination

The question of termination can be put in two ways
a) does the analysis of a nonterminating program terminate?
b) do the algorithms of the analysis terminate?

ad a)

The only tool that is used in termination analysis is time limit specification of the microprocessor equipment programs. Any program run exceeding this limit is considered erroneous. We note that such an error is only a potential one, since in general the program termination condition is not included in each programpart.

ad b)

The problem of termination is raised only in connection with the possible combinations of checkpoint and backward information propagation algorithms. It can be proved that these algorithms terminate in every case.

Summing up, the implementation technique proceeds as follows:

— the program is decomposed to parts, which are then executed (in memory image form) by a simulator;
— symbolic information of elementary and composed data is represented by functions and their values, and is attached to program variables;
— in abstract program evaluation if a condition or operand cannot be evaluated, the program state is checkpointed and execution is continued with one of its possible values;
— the information used in a checkpoint is propagated backward on the elements of the path; this method combines forward and backward execution techniques.

Acknowledgement

The author would like to thank C.H.A. Koster of Nijmegen University for the encouragement and direction in the course of this work.

References

1. BOYER, R. S., ELSPAS, B.—LEVITT, K. N.: SELECT—A formal system for testing and debugging programs by symbolic execution, Proc. Int. Conf. on Reliable Software, 4, 1975.
2. CLARKE, L. A.: A system to generate test data and symbolically execute programs, IEEE TSE, 2, (1976)

3. CLARKE, L.—RICHARDSON, D. J.: Symbolic evaluation methods for program analysis, (in: S. Müchnick, N. D. Jones: Program Flow Analysis, Prentice-Hall 1981.)
4. COUSOT, P.—COUSOT, R.: Static determination of dynamic properties of programs, IFIP WG. 24. M.O.L. Bulletin *5*, 9, (1976)
5. HOWDEN, W. E.: Symbolic testing and the DISSECT symbolic evaluation system, IEEE TSE, 3, 4, (1977)
6. KING, J. C.: Symbolic execution and program testing, CACM *19*, 7 (1976)
7. KOSTER, C. H. A.—FEUERHAHN, H.: Static semantic checks in an open-ended language, Rep. no. 6., Nijmegen Univ., 3, (1977)
8. KOSTER, C. H. A.: Modules and hierarchy, K. U. Nijmegen, Report No. 29, (1982)
9. LEE, J. A. N.: Computer semantics, Van Nostrand Reinhold Co., 1972.
10. OSTERWEIL, L. J.—FOSDICK, L. D.: DAVE—a validation, error detection and documentation system for FORTRAN programs, Software—Practice and experience, *6*, (1976)
11. WEISER, M.: Programmers use slices when debugging, CACM, *25*, 7 (1982)
12. VARGA, L.: A VDL-gráf és alkalmazásai, Akadémiai Doktori Értekezés, 1976. (The VDL-graph and its applications) (Academic doctoral thesis, 1976. In Hungarian)

János SARBÓ H-1521 Budapest