Formalized Symbolic Execution

ZAFIRIOS KARAISKOS CONSTANTINE LAZOS

Department of Informatics
Aristotle University of Thessaloniki
55006 Thessaloniki
GREECE

Abstract: In this paper we present a semantics-directed description of symbolic-execution of programs. In the field of program analysis, most of the methods have been applied on optimizing compilers. It is our belief that there is enough theory that could support the application of program analysis methods on software construction.

Most program analysis methods are syntax-directed, thus, they should base the description of the analysis on some form of semantics, which is syntax-directed, such as the denotational semantics. The description, then, should be augmented with information specific to the method, which finally would be implemented by a tool. Tools built from semantic descriptions of programming language's constructs serve two purposes: make the semantics useful and popular.

Symbolic execution is a program analysis method used to reconstruct logic and computations along a program path by executing the path with symbolic, rather than actual values of data. The results obtained by symbolic execution are the symbolic formulae, representing the values of the program variables, and the path condition for each path examined.

The symbolic execution approaches, traditionally act upon a control flow graph in which nodes represent program segments, which are to be executed linearly, and the only control flow structures are (conditional) branches represented by arcs. The control flow graph is normally structured by using an intermediate representation of the program, such as *triples* or *quadtruples*. This is a "low level" representation of the program.

We follow the syntax-directed approach, which is considered to be "high level". It acts upon a program representation, called *abstract syntax tree*, which includes all of the high-level control structures and the data structures present in the source program. First, we define the abstract syntax for an example language. Then we give a semantic description of symbolic execution, augmented with information specific to the method, in order to be able to maintain the symbolic formulae and path conditions.

Key–Words: Program analysis, Symbolic execution, Syntax-Directed, Semantics-directed, Program representation, Abstract syntax tree.

1. Introduction

Indisputably, there has been a lot of theory developed for each one of the various computing science fields like programming (e.g. formal program design, object-oriented programming), programming language semantics, program analysis. In spite of this, most of the contemporary research effort is wasted in enriching the existing theory (with new methods, new models) instead of using it as a means for the development of tools. More and more voices [15], [9], [8] are heard from the same direction.

In the field of program analysis, most of the methods have been applied on optimising compilers. It is our belief that there is enough theory that could support the application of program analysis methods on software construction. Embedded and safety-critical systems but also large and complex software systems demand sophisticated tools for design, testing and maintenance of software. Program analysis consists of designing and writing program analysers. Program analysers are tools, which produce results about the nature of a program from consideration and analysis of a complete model of some aspect of the program. An important characteristic of such tools is that they do not necessitate execution of the subject program, yet produce results relevant to all possible executions. A very straightforward example of such a tool is a syntax analyser. At the end of a syntax scan it is possible to infer that the program is free of syntactic errors.

Symbolic evaluation is an advanced static program analysis method with a lot of interesting applications: symbolic testing, infeasible path detection, test data generation, program verification, program reduction, and software maintenance [2]. Recently, it has been applied to the reaching definition problem, to worst-case execution time analysis, to cache hit prediction, to alias

analysis, to pointer analysis for detecting memory leaks, and to parallelizing compilers [1], [3]. Symbolic evaluation is used to reconstruct logic and computations along a program path by executing the path with symbolic, rather than actual values of input variables. A path through the program is selected and the statements on the path are executed. Initialised variables are assigned the corresponding constant value; uninitialised variables are assigned the undefined value Λ. Input values needed as read-in values, are represented by symbolic names. Throughout the execution, the representations of all program variables are maintained as algebraic expressions in terms of these symbolic names. The algebraic expressions are formed by the evaluation of any assignment statement along the path. A branch predicate, evaluated over the algebraic expressions of the variables at a decision point, results in a constraint which is then conjoined with all previously evaluated constraints for this path to form the *path condition*, denoted by PC. The path condition is used to determine the feasibility of the program path been examined.

Previous symbolic execution approaches act upon a control flow graph in which nodes represent program segments, which are to be executed in linear fashion, and the only control flow structures are (conditional) branches represented by arcs. The control flow graph is normally constructed using an intermediate representation of the program, such as *triples* or *quadtruples*. This is a "low level" representation of the program. A "high level" approach acts upon a program representation, called *abstract syntax tree (AST)*, which includes all of the high-level control structures and the data structures present in the source program. Another advantage of this approach is that the AST representation of the programs is suitable for different program processing tools such as editors, translators, debuggers, optimisers, and so forth [4].

We took the latter a little further according to the tool-oriented approach [8]. Software is written in some programming language and the language has syntax. Thus, the program analysis method needs to base the description of the analysis on some form of semantics, which is syntax-directed, such as the denotational semantics. The language definition then augmented with information specific to the method becomes the input of the tool generator. This way a semantics-directed[1] tool is developed. Tools built from semantic descriptions of programming language's constructs serve two purposes: make the semantics useful [8] and popular [12]. There is also a nice side effect; such tools do not have to be proven correct, as they are semantics-directed.

This paper reports on the research performed for the formal description of symbolic execution, in the framework of constructing a symbolic evaluation tool, which will eliminate drawbacks such as restricted program models and analysis that covers only linear symbolic expressions. The described system symbolically executes programs written in a small example programming language.

2. The Symbolic Evaluator

Our system has been built in stages (in a Linux environment). First the lexical analyser was constructed using a lex-like tool, and then the syntax and static-semantic analyser were constructed with the help of a yacc-like tool. During the third stage the formal description of symbolic execution was written using techniques from continuation semantics. The formal description was implemented in ML. For the C programs we used the C compiler provided by the system. The ML program has been compiled with the Camlc[2] compiler. All object files have been linked together to produce the symbolic evaluator. The symbolic evaluator reads a program written in the example programming language, performs the lexical, syntax, and static-semantic analysis, and if the program is found to be correct, it produces the AST. Finally, the AST is traversed and the actions are executed, which produce the symbolic (algebraic) expressions for the variables and the path conditions for the executed paths.

2.1 Syntax of Example Programming Language

2.1.1 Abstract Syntax

The concrete syntax contains details needed only for the syntax analysis. For the definition of the meaning of the syntactic forms these details may be ignored and only the structure of possible language constructs that can occur in the programs may be used. The abstract syntax delineates the needed structure. The variety of abstract syntax and its tolerance of ambiguity raise questions concerning its nature and its relation to the language defined by the

[1] Semantics directed means that the structure of the program analysis reflects the structure of the semantics [10].

[2] Camlc compiles and links Caml Light source files. The Caml Light is a dialect of the functional programming language ML. INRIA holds all ownership rights to the Caml Light system.

concrete syntax. Answers can be found by analysing the syntax of the languages in an algebraic context.

We defined a signature G that corresponds exactly to the BNF definition of the example programming language [5], [13]. Each nonterminal became a sort in G and each production became an operator whose syntax captures the essence of the production. The terminals do not appear in the signature since they are embodied in the unique names of the operators.

The term algebra T_G, for the signature G, has as carriers the sets terms(s,G), which contain all ground terms of sort s constructed using the operators in G. The algebra T_G is initial in the collection of all G-algebras, thus, for any G-algebra A there is a unique homomorphism $h: T_G \rightarrow A$.

Consider the following signature:

- Sorts: progr, type, decl, dcltr, expression, stmt.
- Operators:

astEmptyProgram	:	progr
astProgram	:	decl, stmt -> progr
astEmptyDeclSeq	:	decl
astDeclSeq	:	decl, decl -> decl
astDeclaration	:	type, dcltr -> decl
INT	:	type
astVarDcltr	:	dcltr
astArrayDcltr	:	dcltr, expression -> dcltr
astDcltrSeq	:	dcltr, dcltr -> dcltr
astWhileStmt	:	expression, stmt -> stmt
astIfElseStmt	:	expression, stmt, stmt -> stmt
astIfStmt	:	expression, stmt -> stmt
astEmptyStmtSeq	:	stmt
astStmtSeq	:	stmt, stmt -> stmt
astExprStmt	:	expression -> stmt
astNullStmt	:	stmt
astReadStmt	:	expression -> stmt
astPrintStmt	:	expression -> stmt
astArrayElem	:	expression, expression -> expression
astConst	:	expression
astId	:	expression
astMinus	:	expression -> expression
astNot	:	expression -> expression
astMult	:	expression, expression -> expression
astDiv	:	expression, expression -> expression
astMod	:	expression, expression -> expression
astAdd	:	expression, expression -> expression
astSub	:	expression, expression -> expression
astGreater	:	expression, expression -> expression
astGrEq	:	expression, expression -> expression
astLess	:	expression, expression -> expression
astLeEq	:	expression, expression -> expression
astEq	:	expression, expression -> expression
astNotEq	:	expression, expression -> expression
astAnd	:	expression, expression -> expression
astOr	:	expression, expression -> expression
astAssign	:	expression, expression -> expression

Figure 1. The operators of the new signature (also an abstract syntax for the example programming language).

We defined an algebra A with carriers consisting of terms constructed using the above operators, and a mapping $h: T_G \rightarrow A$ defined inductively using a family of functions $h_{A,S}$. h is a homomorphism and because the G-algebra T_G is initial, the homomorphism is unique and the algebra A is a G-algebra too.

The operators, in figure 1, form a version of the abstract syntax for the example programming language. Thus, each version of the abstract syntax of a programming language is an algebra for the signature associated with the grammar that forms the concrete syntax of the language. Algebras acting as abstract syntax will contain confusion (i.e. the mapping h will not be one-to-one), reflecting the abstracting process. By confusing elements in the algebra, we are suppressing details in the syntax.

The syntax analysis concludes by constructing the AST. We let the AST be constructed by creating a node for each operator of the abstract syntax (see figure 2-1). Each node in an AST has been implemented as a record with fields: the name of the operator, a pointer to the symbol (table) node, and pointers to the subtrees. We wrote the function MkNode() to create the nodes of AST's using the underlying productions of the grammar to schedule the calls of it. The function returns a pointer to a newly created node.

2.2 Semantics-directed Symbolic Execution

We shall define symbolic execution using the techniques of denotational semantics and especially continuation semantics. The reasons behind our decision are: a formal definition is precise, it would guide us to the (possibly) best way of implementing symbolic execution, it would provide us techniques to cope with functions and complex data-structures, it would make it convenient to experiment with methods handling simplification of constraints. We will take the engineering approach as M. Gordon did in [6]. For the theoretical basis of denotational semantics the reader is referred to the books by Stoy [14], Schmidt [11], Gunter [7].

A denotational definition consists of three parts: the abstract syntax definition of the language, the semantic domains, and the valuation functions.

The abstract syntax gives the structure of the syntactic constructs of the language being described and is presented in the previous subsection. The *semantic domains* provide the *denotations*, which are usually abstract mathematical objects such as numbers, functions, or terms, and are used to define the meaning of the syntactic constructs of the language. The denotation

of a syntactic construct must depend only on the denotations of its subconstructs (this is called *compositionality*) and so it contributes to the semantics of any complete program in which it may occur. The *valuation functions* map programs, fragments of programs, or particular constructs of the language to their denotations. Given that syntactic constructs have an unambiguous structure, the valuation functions may be defined inductively by specifying, for each syntactic construct, its denotation in terms of the denotations of its components (if there are any). The conventional way of writing such an inductive definition in denotational semantics is as a collection of *semantic equations*, one for each operator of the abstract syntax. The notation for expressing the semantic equations is traditionally based on λ-notation.

2.2.1 Semantic Domains

2.2.1.1 The one element domain

Domain: Unit
Operations: (): Unit

2.2.1.2 The SymbExp domain

Let the algebraic system consisting of the set *Values* :

Values = Int ∪ SymbVal
where SymbVal = {$1,$2,...},

together with the collection of operations in *Values*:

umin, neg: Values → Values
times, div, mod, plus, minus, ls, le, gr, ge, eq, ne, and, or: Values x Values → Values

The domain *SymbExp* consists of the ***algebraic expressions*** (with respect to the algebraic system), which are defined inductively as follows (using prefix notation):

1. Each member of *Values* is an algebraic expression.
2. If e is an algebraic expression then so is $\vartheta_i(e)$, for each unary operation ϑ_i.
3. If e_1 and e_2 are algebraic expressions then so is $\vartheta_i(e_1, e_2)$, for each binary operation ϑ_i.

Defined in this fashion, algebraic expressions are completely parenthesised.

2.2.1.3 Computer Store Locations

The computer store consists of consecutive locations, which are capable of holding storable values (see next section).
Domain: Location
Operations: *nextloc* and some more. *nextloc* returns the next unused location in the list of locations.

2.2.1.4 Denotable, Storable, and Expressible values

The Dvalue domain contains all the values that identifiers may represent. The Svalue domain contains the storable values - the values that can be held in locations. The Evalue domain contains the values, which expressions can produce.

Dvalue=SymbExp+Location+Array+NoVal
Svalue=Dvalue
Evalue=Svalue

where Array = Location
NoVal = Unit

2.2.1.5 Output

The output is the domain *PathCond* which is defined recursively as the domain of path conditions, each one been either *error*, *telos* or an algebraic expression paired with another path condition - 'unwinding' this we see a path condition is either a finite string of algebraic expressions ending with *error* or *telos*, or an infinite string of algebraic expressions.
Domain: PathCond = {error,telos} + (PathCond x SymbExp)

2.2.1.6 Environment

The environment consists of two components. The *Map* component tells what the identifiers mean in the expression - that is, what values the identifiers denote - and is specified by a function from *name* to *Dvalue*. The *Location* is the pool of storage locations. We shall use e, e', e_1, e_2 etc. to range over Environment.
Domain: Environment = Map x Location
where Map=name → Dvalue
Operations: *accessdv, reserveloc*.

2.2.1.7 Store

Associates locations with storable values. We shall use s, s', s_1, s_2 etc. to range over Store.
Domain: Store = Dvalue → Svalue

2.2.1.8 Statement continuations

Statement continuations were first developed for modelling unrestricted branches ("gotos") in general purpose programming languages, but their utility in developing nonstandard evaluation orderings has made them worthy of study in their own right. In case of statement sequence, say st1;st2;, the option must exist of deciding whether the execution can go on from st1 to st2. This is realised by making the value of a statement a function taking one more parameter, called *continuation*. This new parameter, say c, is the dynamic effect of the remainder of the program, which may be ignored, as in

the case of goto, or never reached, as in the case of a "loop forever" situation. A continuation represents a program's complete computation upon a store, so the continuation may contain some final "cleaning up" instructions that produce a final output. The output can be the domain of stores, buffers, messages, or whatever. In our case the output is the domain *PathCond*. We shall use c, c', c_1, c_2 etc. to range over Cont.

Domain: Cont = Store $\rightarrow$ PathCond

Operations: The *fork* operation will be used to realise the parallel noninterfering processing of statements. Hereto we will use the operation:

kai: PathCond x PathCond $\rightarrow$ PathCond.

fork: Cont $\rightarrow$ ***Cont*** $\rightarrow$ ***Cont***

fork = $\lambda c_1.\lambda c_2.\lambda s.(c_1\ s)$ kai $(c_2\ s)$

Computing is divided between c_1 and c_2 and the partial results are joined using *kai*. This is a nontraditional use of parallelism on stores; the traditional form of parallelism allows interference and uses a single-store. The *fork* operation suggests that noncommunicating continuations can work together to build answers rather than deleting each other's updates.

2.2.1.9 Expression continuations

In continuation form, expression evaluation breaks into explicit steps. In each step intermediate values may be produced that must be preserved along the way. In our example programming language, the expressions may update the store, so the domain is:

Econt = Evalue $\rightarrow$ Cont

We shall use k, k', k_1, k_2 etc. to range over Econt.

2.2.1.10 Declaration continuations

Since declarations pass an environment, together with a possibly changed store, to the rest of the program we define the domain of declaration continuations by:

Dcont = Environment $\rightarrow$ Cont

We shall use u, u', u_1, u_2 etc. to range over Dcont.

2.2.2 Valuation Functions

The valuation functions, generally, map terms of the G-algebra T_G to algebraic expressions of the domain *SymbExp* as far as the evaluation of expressions is concerned, and members of the domain *PathCond* as far as the execution of statements is concerned.

An identifier declaration causes an environment update. A new location is reserved, which together with the current environment create a new environment in which the variable binds to the location. The new environment is passed to the rest of the program.

The valuation function for statements has the environment as argument in order to be used in the evaluation of expressions. It carries over a declaration continuation and an expression continuation to be used where needed. The statement continuation contains the point where the execution will be transferred. The statement execution produces a statement continuation given a store.

For the evaluation of expressions, the environment is needed mainly to look up the denotable values of identifiers (for detailed description see the section for semantic domains), the declaration continuation and statement continuation is carried over to be used where is needed, and the expression continuation which contains the point where the evaluation will be transferred. The expression evaluation produces a statement continuation given a store.

To distinguish the different values extracted from expressions we talk about lvalues and rvalues. The rvalue is obtained by dereferencing the value obtained with the valuation function E, and this is realised by another valuation function, called R. In the example programming language, the result of the evaluation of expressions may be an lvalue or not an lvalue (thus, an rvalue). When the evaluation produces not an lvalue the R semantic equations are the same with the E semantic equations.

In the light of the above explanations let us see the functionality of the valuation functions P, D, S, E, and R:

P: progr $\rightarrow$ Environment $\rightarrow$ Dcont $\rightarrow$ Econt $\rightarrow$ Cont $\rightarrow$ Store $\rightarrow$ PathCond

D: decl $\rightarrow$ Environment $\rightarrow$ Econt $\rightarrow$ Cont $\rightarrow$ Dcont $\rightarrow$ Store $\rightarrow$ PathCond

S: stmt $\rightarrow$ Environment $\rightarrow$ Dcont $\rightarrow$ Econt $\rightarrow$ Cont $\rightarrow$ Store $\rightarrow$ PathCond

E: exp $\rightarrow$ Environment $\rightarrow$ Dcont $\rightarrow$ Cont $\rightarrow$ Econt $\rightarrow$ Store $\rightarrow$ PathCond

R: exp $\rightarrow$ Environment $\rightarrow$ Dcont $\rightarrow$ Cont $\rightarrow$ Econt $\rightarrow$ Store $\rightarrow$ PathCond

2.2.3 Semantic Equations

Each semantic function passes the appropriate intermediate result to its continuation. The result of running a program is the output together with an indication of whether the program halted normally or via an error. We will present only the semantic equations that define the statements, in order not to extend the paper further.

The execution of the null statement goes on with the continuation.

S(astNullStmt) e u k c s = c s

The expression is evaluated and the execution goes on with the continuation. The evaluated result is not further needed. The evaluation of the expression has been done for the case it would cause side effects (update of the store).

S(astExpStmt(exp)) e u k c s = E(exp) e u c {λ n. λ s.c(s)} s

The expression is evaluated and the result is used to update the path condition for the "true" and "false" branches. This clause suggests parallel but noninterfering execution of statements. Computing is divided between (st1) and (st2), which goes on with own continuations. This way "all" execution paths can be followed.

S(astIfElse(exp,st1,st2)) e u k c s = R(exp) e u c {λ n.fork(S(st1) e u k {λ s.<c(s),n>}) (S(st2) e u k {λ s.<c(s), neg(n)>})} s

The expression is evaluated and the result is used to update the path condition for the "true" and implied "false" branches. We have again parallel but noninterfering execution. The first executes the statement and goes on with the continuation. The second just goes on with the continuation.

S(astIf(exp,st)) e u k c s = R(exp) e u c {λ n.fork(S(st) e u k {λ s.<c(s),n>}) (λ s.<c(s),neg(n)>)} s

The expression is evaluated and the result is used to update the path condition for the one loop and no loop branches. We have again parallel but noninterfering execution.

S(astWhile(exp,st)) e u k c s = R(exp) e u c {λ n.fork(S(st) e u k {λ s.<c(s),n>}) (λ s.<c(s), neg(n)>)} s

3. Conclusion

The first of our objectives have been accomplished. We have given a formal description of the abstract syntax of the example programming language and a semantics-directed description of symbolic execution. We also have transformed the formal description of symbolic execution to an ML program. The ML program symbolically executes a program in the example language and produces, for each program path, the path condition and for each variable the respective algebraic expression. The results are very encouraging and it seems that the implementation is very efficient in contrast to other implementations found in the literature.

As future work we will transform the semantics-directed description of a "real" programming language to an ML program. This description will include all data structures and control structures encountered in the language. Next we will implement the constraints simplifier possibly using interval algebra.

4. References

[1] Blieberger, J., B. Burgstaller, and B. Scholz, Symbolic data flow analysis for detecting dealocks in Ada tasking programs, *Ada-Europe 2000 International Conference on Reliable Software Technologies.* (to appear).

[2] Coward, D. and D. Ince, *The symbolic execution of software: the SYM-BOL system*, Chapman & Hall, 1995.

[3] Fahringer, T., Symbolic analysis techniques for program parallelization, *Journal of Future Generation Computer Systems, Elsevier Science*, vol 13, 1997/98, pp. 385-396.

[4] Ghezzi, C. and M. Jazayeri, Syntax directed symbolic execution, *Proceedings of COMPSAC 80*, 1980, pp. 539-545.

[5] Goguen, J.A., J.W. Thatcher, E.G. Wagner, and J.B. Wright, Initial algebra semantics and continuous algebras, *Journal of the ACM*, vol 24, no 1, 1977, pp. 68-95.

[6] Gordon, M.J.C., *The denotational description of programming languages*, Springer-Verlag, 1979.

[7] Gunter, C.A., *Semantics of programming languages: structures and techniques*, MIT Press, 1992.

[8] Heering, J., and P. Klint, Semantics of programming languages: a tool-oriented approach, *ACM SIGPLAN Notices,* March 2000.

[9] Le Metayer, D., Program analysis for software engineering: new applications, new requirements, new tools, *ACM Computing Surveys*, vol 28, no 4es, 1996.

[10] Nielson, F., H.R. Nielson, and C. Hankin, *Principles of program analysis*, Springer-Verlang, 1999.

[11] Schmidt, D.A., *Denotational semantics: a methodology for language development*, Allyn and Bacon, 1986.

[12] Schmidt, D.A., On the need for a popular formal semantics, *ACM Computing Surveys*, vol 28, no 4es, 1996.

[13] Slonneger, K. and B.L. Kurtz, *Formal syntax and semantics of programming languages*, Addison-Wesley, 1995.

[14] Stoy, J.E., *Denotatioal semantics: the Scott-Stratchey approach to programming language theory*, MIT Press, 1977.

[15] Wilhelm, R., Program analysis – a toolmaker's perspective, *ACM Computing Surveys*, vol 28, no 4es, 1996.