SOFTWARE VALIDATION, H.L. Hausen (editor)
Elsevier Science Publishers B.V. (North-Holland)
© GMD, 1984

Symbolic Evaluation as a Basis for Integrated Validation

Erhard Ploedereder, Ph.D.

Tartan Laboratories

Pittsburgh, PA, USA

Abstract

Symbolic Evaluation derives information about the static and dynamic semantics of programs by means of a static and global program analysis This information is then deposited in a program data base to be used by various tools supporting program development and validation

The theoretical foundations of Symbolic Evaluation were developed at Harvard University. The Harvard Program Development System (PDS) included as a component for semantic analysis a prototype implementation of a Symbolic Evaluator.

This paper presents the model underlying Symbolic Evaluation and discusses its impact on tools for program development and validation[1]

1. Introduction

The need for more sophisticated programming tools to assist the process of developing and validating reliable, efficient and well documented software is widely recognized. A prerequisite for many of these tools is the ability to reason about the semantic meaning of programs.

In state-of-the-art implementations the derivation of such semantic information is specially adapted and inseparably linked to its use within the respective tool performing a particular task, such as exception detection, program verification, program debugging, validation or interactive derivation of program specifications, static performance analysis, test-case generation and so on In an integrated system of many such tools, however, it would be desirable to have a central analyzer producing the required semantic information for all these tools in a common program data base. This approach not only avoids the repeated derivation of the same information for different purposes, but also provides for an improved interaction of the different tools by means of the common program data

[1]Research reported herein was performed at Harvard University and in part supported by the Advanced Research Projects Agency under contract no N00039-78-G-0020

base. As a further benefit, the tools using the data base can be easily adapted to be applied to programs written in a variety of languages if the representation of information in the program data base is largely independent of the programming language used.

In our research we, therefore, have taken a significantly different approach by separating the task of deriving knowledge about programs from the utilization of this knowledge. We perform a static, global program analysis that examines each expression of the input program only once and attempts to derive a precise characterization of the values computed and the side-effects caused. *Symbolic expressions* are used to represent the values of data-objects and computations occurring in the program. The results of this analysis, called *Symbolic Evaluation* [3], are collected in a program data base to be used by various tools that assist program development and validation.

The notion of symbolic expressions is not a novelty in the software engineering area. Various researchers have proposed approaches that use symbolic expressions as descriptions of program values occurring during an arbitrary evaluation of the program [1,6,15,17,20]. However, these approaches usually provide only for a symbolic execution of a single path through the program at a given time, rather than for a symbolic evaluation of the complete program in a single analysis. Symbolic Evaluation also provides a basis for analyzing the behaviour of loops [26] and for deriving *templates* characterizing the results and side-effects of user-defined subprograms [21]. The latter ability allows us to assess the effects of individual calls efficiently.

Symbolic execution can be viewed as a special case of Symbolic Evaluation; it can be trivially implemented as a tool based on the results of Symbolic Evaluation.

2. Local and Global Program Analysis

The typical analyzing methods incorporated in verification systems, compilers and many other semantic analyzers truncate the flow of computation prescribed by a program into small segments (*e.g.*, paths between assertions [10,12], basic blocks and regions [11]) and analyze the dynamic semantics of these parts in isolation. Facts that cannot be derived from these isolated parts, but may influence their actual evaluation, have to be described by a suitably weak representation correct for any instance. If loops and recursions are contained in the analyzed segments, the derivation of their effects is usually limited to the determination of guaranteed invariances; the propagation of less trivial knowledge across these constructs is usually left entirely to the verification of the analyzed program. Furthermore several features often found in higher-level programming languages create substantial problems for a localized analysis of programs. Examples include storage sharing (aliasing), pointer semantics, procedure and type variables, and the locative use of non-trivial program expressions, *e.g.*, function calls. The predominant theoretical basis for a localized analysis of dynamic semantics of programs for the purposes of verification is the inductive assertion method due to Floyd [10], and the deductive logical systems subsequently developed by Hoare [12, 13, 14] and others (*e.g.*, [2, 8, 16, 13]).

Such an approach is not particularly suited for a central analyzer, since the truncation parameterizes the obtained results, and thus has to be observed by all peripheral tools using these results The integration of results obtained by differently focused analyses is often difficult or impossible without a re-analysis of the program. Also, the exclusive dependency on verification to derive knowledge about the effects of loops and procedure calls is not very attractive, since, in many situations, facts about the behaviour of these constructs can be derived from the program text without resorting to user-provided assertive specifications

There is, however, a seemingly convincing argument in favor of a local analysis: due to unknown input or parameter values, many conditionalities arising during the evaluation of the program cannot be decided by the analysis: the cumulative effect of unresolved conditions may cause a combinatorial explosion in the description of values obtained by the analysis In a local analysis, the number of such conditionalities is substantially smaller.

While the basic truth of this argument cannot be refuted, some amendments have to be made:

First, given the appropriate program structure, the analysis of isolated paths replaces the potentially exponential complexity in result descriptions by a guaranteed exponential number of paths to be explored

Second, for all but the most restricted languages, the indiscriminate use of weak representations of knowledge can be a contributing cause for an accelerated combinatorial explosion, and may weaken the results obtained to a degree in which they are virtually useless for any reasoning about the program. (Some examples, relating in particular to usage of pointers can be found in [22]). The required worst-case assumptions consider many cases a more global analysis can eliminate knowing that they cannot occur during any actual evaluation of the program.

Finally, the exponential explosion of value-descriptions is caused by keeping the strongest possible representation of values in a string-like form. This explosion of the representation can be almost entirely avoided, if a context-independent, graph-oriented representation is chosen. Hence, a global analysis can be performed with reasonable space- and time-requirements, producing substantially stronger results than a series of local analyses. The tools reasoning about the information thus represented have to cope with the problem of deciding whether a value representation (*i.e.*, a sub-tree) should be treated as a weak symbolic token or dealt with at higher levels of strength. Combinatorial explosion of the complexity of the reasoning can be controlled at this point by heuristics of the individual tool, deciding not to use the full strength of the information available. If, on the other hand, the information obtained by the global analysis has a representation simple enough to be used by a tool without substantial cost, the tool can proceed with considerably improved efficiency. Hence, the tools are not impeded by consequences of the limitations of a local analysis. The weakening of information does not take place during analysis, but rather during the reasoning about the results of the analysis. At this point it is controlled heuristically based on the complexity of the information and the importance of the task to be accomplished rather than during analysis based on some global criterion of program structure that may be unrelated to complexity.

3. The Model

The formal model for Symbolic Evaluation [22] was originally developed for the language EL1 [18], a very flexible locative expression language including as a subset the majority of data and control structure concepts - with the exception of "goto" - present in modern higher-level programming languages for sequential programming (*e.g.*, ALGOL60, ALGOL68, PASCAL, EUCLID, ALPHARD) Instead of unrestricted "goto"s, EL1 supports "exit"s from loops and blocks and "return"s from subprogram expressions and marked expressions.

The model was developed after a preliminary implementation of a Symbolic Evaluator had already been in existence. This implementation provided valuable insights into the requirements of a program analysis with respect to a semantic model; in return, the model allowed substantial improvements to be made to the implementation.

The model is described as a set of semantic equations. It bears strong resemblances to denotational semantics, but is adapted to the needs of a static program analysis. Most importantly, a distinction is made between functions whose interpretation is statically decidable and important for an efficient analysis, and functions whose interpretation cannot or should not be decided during the static analysis and, instead, is left to the reasoning processes in peripheral tools. We refer to the latter category as the pragmatics of the language. Typical examples of pragmatic functions are functions that represent the application of a built-in operator to its arguments, *i.e.*, "plus(1.0, 2.9)" which represents the computed result of a program expression "1.0 + 2.9". Tools can apply different interpretations to such pragmatic functions, *e.g.*, as operations over idealized or machine-dependent domains.

Our model distinguishes only two semantic domains:

1. the set of L-value descriptions ("*places*") which represent the locative results of program expressions; the interpretation of this semantic domain is fixed and decided as part of the analysis.
2. the set of R-value descriptions ("*symbolic expressions*") which represent computed data values; the interpretation of this domain is generally left to the reasoning tools.

Given a fixed interpretation of symbolic expressions, this structured approach can be disregarded; our model can then be viewed as a mathematical, denotational semantics.

Another major distinction from denotational semantics as described in [24] is a quite different approach towards continuations; they are modelled by template applications (for loops and subprogram calls) and parameterization of the semantic equations inhibiting state transformations by program expressions following any form of unconditional control transfer The effects of synchronous run-time faults are also included in the model.

In the following sections we give a brief and slightly simplified overview of the model by introducing the underlying concepts and providing some examples.

3.1. Symbolic Expressions

Symbolic expressions are an algebraic denotation representing R-values. A symbolic expression is one of the following:

- a constant token representing a literal in the program or an address value returned by the allocation of a heap object;
- a variable token representing an unknown input value;
- a constrained variable token representing a set of values denoted by other symbolic expressions, *e.g.*, symbolic expressions in loops and subprogram calls;
- a token representing a bound variable in a quantified symbolic expression;
- a symbolic function applied to the appropriate number of symbolic expressions;
- an equality applied to symbolic expressions; or
- a quantified symbolic expression

Each built-in operator in the language has a corresponding symbolic function.

Some additional symbolic functions and their interpretation are predefined as part of the model. These are structural functions that allow the denotation of structured R-values (arrays and records) and of components thereof, and conditional functions that allow the denotation of conditional symbolic expressions. Constrained variable tokens are used in template generation for loops and subprograms Such a token can also be viewed as a decidable function parameterized by a loop cycle or subprogram call index, yielding (a possibly undecidable) symbolic expression.

In order to reason about symbolic expressions, an appropriate model consisting of an axiom system and an interpretation is needed The interpretation maps constant tokens into elements of the corresponding domains and symbolic functions into functions over these domains. Variable tokens are considered universally quantified over any predicate formed from symbolic expressions. The choice of an appropriate model is left to the reasoning tool with the exceptions noted above with regard to structural and conditional symbolic functions.

3.2. Places

Places are a generalized notion of *locations*. Locations characterize a storage area for R-values. Place denotations are used to describe the locative results of program expressions.

A place is one of the following alternatives:

- a "pure value place", typically the L-value of an intermediate result not accessible by variables or pointers;
- a "location place", typically the L-value of the evaluation of an identifier;
- a "selected place", typically the L-value of the evaluation of a selection such as "RecordObject Component"; or
- a "conditional place", typically the L-value of a conditional expression or of dereferencing a union-pointer with conditional R-value

The denotation of places has a predefined interpretation up to the interpretation of symbolic expressions involved in selected and conditional places Predicates involving places can always be reduced to predicates involving only symbolic expressions.

3.3. Store, Environment and Memory-State

The store is a set of locations. Each location in the store has an *address*, which is associated with the name of a declared variable or a pointer R-value or set of such values. These associations are specified in *descriptors*. The set of descriptors forms the *environment*. Since our model allows partial and conditional aliasing, descriptors associate variable names with the denotation of a place rather than a location.

The pair (environment, store) is referred to as the *memory-state*.

The connection between L-values and R-values is established by the "value"-function which is decided as part of the analysis. It yields the denotation of the R-value stored in the given place for a given memory-state

value: {Place} x {Memory-State} -> {Symbolic Expr}

The memory-state is altered as described by the semantic equations for declarative constructs and assignments The assignment semantics use the "write"-function to alter the R-value denotation of a given place in a given memory-state: this function which yields a new memory-state is decided as part of the analysis.

write: {Memory-State} x {Place} x {Symbolic Expr} -> {Memory-State}

The place associated with an identifier within the program is established by the "plc"-function which is decided as part of the analysis. It yields the denotation of the place for the given name in a given memory-state. The interpretation of the "plc"-function is decided as part of the analysis: it is language-dependent to the extent of accommodating different scoping rules

plc: {Identifier} x {Memory-State} -> {Place}

3.4. The Symbolic Evaluator Function SE

The function "SE" describes the semantics of the various expressions of the programming language in terms of the semantics of their sub-expressions and of functions operating on the semantic domains of the model. It maps the respective program expression and a given memory-state in which the expression is to be evaluated into the denotation of its locative result and of the resulting memory-state. The latter reflects potential side-effects of the expression.

SE: {expression} x {Memory-State} -> {Place} x {Memory-State}

The SE-function is constructive in nature Given an efficient implementation for the representation of memory-states and operations on this representation, the majority of SE-equations can be directly transliterated into an efficient implementation of a semantic program analyzer (the exceptions being loop and subprogram analysis for which some less obvious strategies must be chosen for obtaining an efficient implementation).

Although the SE-function is oriented towards locative expression languages, it is equally applicable to statement-oriented languages The syntactic constraints imposed by the latter languages will cause certain combinations of semantic *equations never to appear*, while the equations in isolation are still valid for the purposes of program analysis.

Symbolic Evaluation assumes a strict order in the evaluation of expressions, *thus overspecifying the language semantics. This is unavoidable to prevent a* totally uncontrolled and often unnecessary explosion in the complexity of R-value descriptions. In an implementation of a Symbolic Evaluator it is fairly easy to ascertain whether this assumption had any effects on the results obtained and, if so, issue warnings to the user or accomodate the non-determinism by adjusting value descriptions retroactively.

3.5. *Examples of the SE Equations*

In the following, the notations "SE(X,MS).PLC" and "SE(X,MS).MST" are used to denote the place and memory-state obtained by applying SE to an expression X and a memory-state MS. "value(SE(X,MS))" will be used as abbreviation for "value(SE(X,MS) PLC, SE(X,MS).MST)". The equations given are slightly simplified; fully detailed equations are given in [22].

A) Non-Repetitive Constructs:

Identifiers:

```
SE(ID, MS) = (plc(ID,MS), MS)
   where ID is the name of a declared entity
```

Selections:

```
SE(E[i], MS) = ( (SE(E,MS).PLC, <value(SE(i,MS1))>), MS2 )
   where MS1 = SE(E,MS).MST
         MS2 = SE(i,MS1).MST
```

Assignments:

```
SE(E1 := E2, MS) =
  ( SE(E1,MS).PLC, write(MS2, SE(E1,MS).PLC, value(SE(E2,MS1))) )
   where MS1 = SE(E1,MS).MST
         MS2 = SE(E2,MS1).MST
```

Statements within expressions:

```
SE( (E1; E2), MS) = SE(E2, SE(E1,MS).MST)
```

Conditional expressions:

```
SE(IF Q THEN E1 ELSE E2 FI, MS) =
  ( (cond qv, plc1, pcl2), combine_conditionally(qv, MS1, MS2, MS3) )
   where
     qv  = value(SE(Q,MS))
     MS1 = SE(Q,MS).MST
     (plc1, MS2) = SE(E1, MS1)
     (plc2, MS3) = SE(E2, MS1)
```

"combine-conditionally" is an auxiliary function of the model which returns a memory state in which all R-value descriptions that differ in any two of the given memory states are replaced by conditional symbolic expressions based on the value of the branch predicate and the original values. (In an efficient implementation the memory-state is augmented by a flow-graph. The function extends the flow-graph without modifying R-value denotations. The "value"-function then uses the flow-graph to derive the conditionalities caused by the branch.)

SE Equations for Loops:

The semantic equations for loops are template-oriented A loop template is created by means of symbolically evaluating the loop in an "inductive memory-state" (IMS) The environment of the IMS contains entries for all non-local variables referenced in the loop: its store contains corresponding locations whose R-values are represented by constrained variable tokens. The IMS is sufficiently general to represent the memory-state at the beginning of each cycle of the loop. The SE-function establishes two memory-states. the state at the end of the loop body (IMSA), and the state after exiting the loop (IMSF) The R-values in locations of IMS, and the corresponding values in MS and IMSA describe a first-order recurrence relation of symbolic expressions, constrained by the negation of the loop-controlling predicate.

The semantics of the WHILE-loop are modelled as a concurrent assignment of the symbolic representation of the solutions for this recurrence relation A special tool [26] can later attempt to find closed-form symbolic solutions for these relations. The WHILE-loop has an empty L-value

```
SE(WHILE Q REPEAT E END, MS) = ( (#, nothing),none) , MSF)
   where
      IMS = the inductive memory-state
      IMSA = SE( (Q; BEGIN R END), IMS).MST
      IMSF = SE(Q, IMS).MST
      <X1,...,Xn> is a list of the places (for non-local variables
                    referenced in Q and E) in IMS
      <X1',. .,Xn'> is the list of corresponding places in MS

      INITIAL = < value(X1',MS),.. ,value(Xn',MS) >
      BEGIN    = < value(X1,IMS), ..,value(Xn,IMS) >
      AGAIN    = < value(X1,IMSA),...,value(Xn,IMSA) >
      FINAL    = < value(X1,IMSF),...,value(Xn,IMSF) >
      EXIT     = not*( value(SE(Q,IMS)))
      FINAL*   = subst(FINAL, BEGIN, solve*(BEGIN, INITIAL, AGAIN, EXIT))
      MSF      = write(...write(MS, X1', FINAL*1). ., Xn', FINAL*n)
```

(An efficient implementation will not require establishing special inductive memory states. They can be overlaid with the memory-state in which the loop is evaluated by simply installing a constrained variable token in the respective locations retroactively whenever such locations are first referenced within the loop. The token acts as a representation for the solved recurrence relation)

3.6. The Algorithmic Semantics Functions WSP and SSP

The SE-function explains the semantics of the language constructs, but does not allow relating these semantics to assertive specifications. The functions WSP and SSP accomplish this connection. They are variants of Dijkstra's "predicate transformers" [9] for the derivation of weakest preconditions (WSP) and strongest postconditions (SSP), transposed into the setting of denotational semantics and augmented to allow the generation of practical verification conditions based on inductive assertions for partial correctness.

```
WSP: {expression} x {Memory-State} x {Symbolic Expr} -> {Symbolic Expr}
SSP: {expression} x {Memory-State} x {Symbolic Expr} -> {Symbolic Expr}
```

WSP(PROG,MS,P) yields a symbolic expression representing a pre-condition that guarantees both the correctness of the program fragment PROG with respect to assertions contained in PROG, when PROG is evaluated in MS, and the truth of the post-condition P after this evaluation

SSP(PROG,MS,Q) is a symbolic expression representing a post-condition that can be inferred from the validity of Q prior to the evaluation of PROG in MS and the correctness of this evaluation with respect to assertions contained in PROG.

In order to provide useful information to a verifier, we induce some structure upon symbolic expressions in the image domains of the WSP and SSP functions by means of the following symbolic functions of fixed interpretation:

1. vcb*(P,Q) indicates the necessity of proving Q from a premise equal to, or stronger than P.
2. vcf*(P,Q) indicates the necessity of proving Q from a premise equal to, or weaker than P.
3. ax*(P) indicates a restriction on the applicability of the program to states satisfying P; this restriction may be assumed on faith.
4. t_ax*(P(kl)) indicates a restriction to states satisfying "exists kl: P(kl)"; this restriction guarantees termination.

vcb*(P,Q) corresponds to a verification condition (P -> Q) generated by backward analysis, vcf*(P,Q) corresponds to a verification condition (P -> Q) generated by forward analysis, ax*(P) is the trivially successful verification condition P for an assumed predicate, and t_ax*(P(kl)) is either treated as a verification condition or an axiom, depending on whether an attempt is made to prove total or partial correctness, respectively.

A program is correct under some model for symbolic expressions if for all vcb* or vcf* functions in WSP(PROG,(),true) or SSP(PROG,(),true), respectively, the truth of the corresponding verification condition can be deduced in the model. The model provides a basis for attempting to prove total correctness.

3.7. Examples of the WSP Equations

The following examples are simplified by neglecting the possibility of implicit assertions generated by the evaluation of expressions, such as assertions that guarantee array indices to be within the prescribed bounds. We omit examples for SSP equations since they are quite analogous.

A) Identifiers, Selection and Assignment

```
WSP(ID, MS, P) = P

WSP(E[i], MS, P) = WSP(E, MS, WSP(i,MS1,P))
  where MS1 = SE(E,MS).MST

WSP(E1 := E2, MS, P) = WSP(E1, MS, WSP(E2, MS1, P))
  where MS1 = SE(E1,MS).MST
```

The equation for assignment is surprising at first glance, since it implies WSP("Q:=3", MS, P) = P, *i.e.*, the assignment has no effect on the predicate its justification lies in the fact that an assignment influences the memory-state in its location-value bindings, but never an existing symbolic expression

B) Conditional Expressions:

```
WSP(IF Q THEN E1 ELSE E2 FI, MS, P) =
  WSP( Q, MS, and*( value(SE(Q,MS)) -> WSP(E1, MS1, P) ,
                    not*(value(SE(Q,MS)) -> WSP(E2,MS1, P) ) )
      where MS1 = SE(Q,MS).MST
```

A normalizing transformation causes the implying symbolic predicates to become premises of vcb*-applications within the weakest symbolic pre-conditions of sub-expressions, *i.e.* the branches.

C) Assertions:

```
WSP(ASSERT(V), MS, P) =
   and*( vcb*(true, value(SE(V,MS))), value(SE(V,MS)) -> P )

WSP(ASSUME(V), MS, P) =
   and*( ax*(value(SE(V,MS)), value(SE(V,MS)) -> P )
```

3.8. Properties of the WSP and SSP Functions

The following properties can be proved for the WSP and SSP functions. The notation "E1 == E2" is used for semantic equivalence of the symbolic expressions E1 and E2 under any interpretation that maps boolean symbolic functions to the respective boolean functions.

Theorem 1: (Invariance)

If the expression E contains no loops, procedure calls, and implicit or explicit *assertions, then*

WSP(E,MS,P) == P

SSP(E,MS,P) == P

This theorem guarantees the capability of propagating predicates across many types of expressions without requiring a reanalysis of the expression.

Theorem 2: (Separation of Concerns)

For all expressions E:

a) *WSP(E,MS,P) == and*(SI(SSP(E,MS,true)) -> P, WSP(E, MS, true))*

b) SI(SSP(E,MS,Q)) == and*(SI(Q), SI(SSP(E,MS,true)))

c) SSP(E,MS,Q) == and*(Q, SI(Q) -> SSP(E,MS,true))

where SI maps all occurrences of vcb*, vcf*, ax*, and t_ax* into the respective verification conditions. It thereby determines the strongest symbolic post-condition without the structure induced by the symbolic functions isolating verification conditions.

This theorem guarantees the capability of analyzing each language expression in isolation by means of the WSP and SSP functions, and of integrating the results of this analysis into verification conditions for expressions enclosing the expression. It also guarantees that all strongest post-conditions, and by corollary *post-conditions in general, can be propagated across following expressions with*out alterations. Post-conditions propagated out of branches receive the branch predicate as an implying premise.

Theorem 3: (Equivalence)

For all expressions E:

t(WSP(E,MS,true)) == t(SSP(E,MS,true))

where t is the normalizing transformation and replaces vcb* and vcf* by vc*.

This theorem guarantees that forward or backward directed generation of verification conditions will yield **identical** verification conditions for all assertions contained in E.

It is a consequence of these theorems that

1. an implementation of a verifier can collect premises for a verification condition from all preceding program expressions without modifying them except by adding branch enabling predicates leading to the respective expression;

2. there are as many verification conditions as there are assertions: some of the verification conditions containing constrained variable tokens may require inductive proofs;
3. an implementation can allow assertions to be inserted at any place in the program without a re-analysis of the program, except for determining the symbolic value of the predicate in the context in which it is asserted

4. The Implementation of the Symbolic Evaluator

The implementation of the Symbolic Evaluator within the Harvard PDS [5] is described in detail in [3,21,22]. This section provides a brief overview of the underlying principles of an efficient implementation.

An implementation of the SE-function must optimize the representation of memory-states with appropriate trade-offs in the complexity of the functions operating on memory-states, as well as optimize the representation of symbolic expressions representing computed R-values.

In our implementation we have chosen to represent the memory-state sparsely by adding lists of so-called *value cells* to the locations of the store. These value cells contain a context characterization and the symbolic expression representing the R-value of the location in this context. The context relates to an acyclic flow-graph, the *context graph*, whose branching nodes are labelled with the respective symbolic branch predicate. This graph is acyclic because loops and function calls are modelled as multiple assignments to the affected entities, as determined by the loop or subprogram templates. The context graphs for the loop and subprogram bodies are isolated sub-graphs with special root and leaf nodes identifying these contexts as the respective special start, end, recursion or repetition contexts used in establishing the loop or subprogram templates.

Value cells are created by the "write"-function modelling assignments and by the procedure call and loop analyzing parts of the implementation The cost of a single application of the "write"-function is essentially constant in time and space; conditional places, which rarely occur in programs, increase this complexity.

The "value"-function establishes the symbolic expression representing the value of a given location in a given context by an intersecting traversal of the context graph and the value cells found in the location The algorithm employed is reminiscent of path compression as described by Tarjan [25]. The algorithm terminates when for each path leading to the context a value has been found. The applicable values are combined in a conditional symbolic expression reflecting the respective path predicates. The algorithm we use is optimal in the sense that it produces the minimal representation of the R-value of a location up to attempting to prove the falsehood of path-enabling predicates. It is guaranteed that the algorithm never needs to search beyond the root nodes of an isolated context graph of a loop or subprogram. The complexity of the algorithm is linearly bounded by the number of value cells and squared bounded by the number of preceding branches in the context graph. The constant factor is very small. This worst-case complexity can only arises for programs in which the number of paths is linear in the number of branches. For programs with an exponential number of paths the worst-case complexity is linear in both the number of value cells and branches. Due to the locality of variable usage, most invocations of the "value"-function require substantially less time and space than under worst-case assumptions.

An efficient representation of symbolic expressions is accomplished by using a graph-oriented representation whose nodes represent the R-values computed by program expressions. Each expression in the program whose R-value is needed causes the generation of such a node. It can be shown that the total space requirement for symbolic expressions is bounded by the number of expressions and assignments in the program, if the "value"-function is not decided by the Symbolic Evaluator but left to reasoning tools. If it is decided, then nodes for R-values of identifiers require non-constant space which is sub-linearly proportional to the complexity of the corresponding value-retrieval.

An added benefit of the chosen representation is that it facilitates the detection of all common subexpressions in the program regardless of intervening branches. Furthermore, it allows minor modifications of the program and a corresponding incremental adjustment of the program data base to be made without a re-analysis of the program; the adjustment is trivial for the alteration of expressions that have no side-effects on the memory-state. Only alterations to the control structure of the program invalidate the results of the symbolic evaluation.

In preliminary tests of the pilot implementation, experience has consistently shown that the size of the program had virtually unnoticeable influence on the cost of individual value retrievals which consumed about 10% of total analysis time. This is attributed to the locality of variable usage. The same holds for overall performance in the absence of procedures: the time- and space-requirements are approximately linear in the size of the program. The presence of many subprogram calls degrades performance, since the template generation, storage, retrieval and instantiation is of non-trivial cost (although more efficient than a re-analysis of the subprogram body, except for trivial subprograms).

5. The Implementation of the Verifier

For a detailed description of the implementation, we refer the reader to [22]. In this section, we discuss only the most salient properties of the verifier.

The proven invariance of symbolic predicates when propagated across exceptions has far-reaching consequences for the verifier. Its implementation never reanalyses the program. The task of establishing verification conditions consists of traversing the list of asserted predicates and of collecting premises from contexts preceding an assertion under inclusion of enabling branch predicates. The implementation traverses a list of "assertion cells", each containing the symbolic value of the asserted predicate and the context in which it is asserted. This task is virtually identical to the task the "value"-function performs: the "value"-function traverses the list of value cells for a given location and combines the applicable contents in conditional symbolic expressions reflecting branch predicates. Verification condition generation uses precisely the same algorithm to obtain all premises provided by the contextually closest preceding assertions for each path leading to the assertion to be proved. It can, however, go beyond these delimiting assertions and retrieve information from assertions farther back in the program. With an appropriate set of heuristics, the choice of premises collected can be controlled to avoid an abundance of premises unrelated to the assertion under examination. The efficiency of the algorithm is substantially increased by the fact that branches not containing any assertions can be skipped in collecting premises without suffering any loss of information.

Furthermore, the program analysis which preceded the verification implicitly propagated value invariances across loops and procedures, so that many assertions required in other verification methods are not needed.

Lastly, if an assertion cannot be proved from the premises given, the user can insert additional assertions at arbitrary points in the program without causing a re-analysis of the program. The only action to be taken by the verifier is to symbolically evaluate the assertion in the context of its insertion to obtain its symbolic value. This value is then added to the assertion list, verified and used as premise for subsequent assertions.

For verification conditions that contain tokens introduced by loops, the proving component of the verifier has two options, if a straightforward attempt fails to prove the verification condition on the basis of preceding premises. First, it can call on recurrence relation solving tools which attempt to find closed form symbolic solutions for the tokens and then re-attempt a proof [7, 26]. Second, it can attempt an induction proof by generating two verification conditions for the first and the (n+1)-th cycle of the loop. These verification conditions are obtained by collecting the inductive premises and by replacing the tokens with the appropriate symbolic expressions recorded in the loop template. For assertions within procedure bodies that cannot be proved, the proving component can - in interaction with the user - establish derived entry-conditions for the procedure if the proof indeed failed because of a missing entry-condition. In the case of entry-conditions for recursive procedures, a recursion induction can be performed [19].

If an attempt is to be made to verify an assertion only with respect to a single path leading to it, the respective verification condition can be obtained by a trivial simplification of the general verification condition for the assertion after the appropriate branch predicates are replaced by 'true'. Generally, this strategy will be used by the proving component for splitting complicated verification conditions into isolated clauses.

6. Symbolic Evaluation as a Basis for other Tools

While verification is one of the most challenging applications for the results of a program analysis, a full-scale verfication based on predicative assertions is beyond the capabilities of tools for practical program validation. However, many other tools used in program development and validation can also be substantially assisted by the information contained in the program data base developed by the symbolic evaluation of the program. Some of these tools can be implemented by simply providing a user-interface into this data base. As an added pragmatic advantage, utilization of the information produced by the symbolic evaluation of the program guarantees that all tools assign the same (correct) semantics to the programming language - all to often this is not the case if tools independently analyze the original program text.

Some examples of such tools are:

- Set/use Lists: The lists of value cells for locations provide a record of all assignments to variables. For reasons having to do with non-deterministic evaluation order, the Symbolic Evaluator also has to keep track of usages of the values of variables. This record provides the necessary information about variable usage. The respective context relates the setting or usage of values back to the program text.
- Dynamic Lifetime Analysis: Based on the set/use analysis, it is trivial to determine the maximum life-time of variables within their scope.
- Description of all side-effects of a subprogram: Part of the template created for a subprogram is the record of all non-local variables referenced or modified by its body or any of its sub-called subprograms. Having a record of all such references and modifications regardless of the call stack depth at which they occur is a valuable asset in program validation.
- Cross-Reference Lists: Based on the information above, cross-reference

listings can be produced.

- Analysis of Aliasing: The environment component and the subprogram templates of the program data base produced by the Symbolic Evaluator contain a record of all declared aliases and of all potential aliases created by parameters in procedure calls
- Exception Detection: Part of the Symbolic Evaluation is the creation of implicit assertions whose truth guarantees the absence of run-time exceptions. The context of such symbolic assertion values identifies the points in the program at which exceptions may occur. A moderate amount of verification effort can discover the truth of many such assertions [16]. The user can be made aware of any assertion that cannot be proved, indicating the potential exisitence of a run-time error.
- Semi-Automated Derivation of Entry-Conditions: Any assertion in a procedure that cannot be proved correct can be easily propagated back to the initial context of the procedure body where it can be installed as an entry condition to be proved for each call situation. This propagation can be fully automated; the user only has to state whether the unproven verification condition indeed represents a restriction on the applicability of the subprogram or the failure of proving it was caused only by the restricted capabilities of the proving component
- Symbolic Execution: The results of a symbolic execution can be obtained from the program data base by instantiating branch predicates as desired. A certain amount of simplification is required for symbolic expressions that involve conditions depending on such branch predicates.
- Test Case Generation based on Symbolic Execution: Given the results of a symbolic execution as explained above, the same methods as known from symbolic execution can be applied.
- Executable Specifications: The Symbolic Evaluator has been designed to allow the extension of symbolic evaluation to executable specifications expressed in a style equal or similar to the REWRITE facility of the Harvard PDS [4]. While this extension has not been implemented, its implications are believed to be sufficiently understood.

7. Summary

Symbolic Evaluation performs a global semantic program analysis. Any part of the program is analyzed but once The results of the analysis are deposited in a program data base. An appropriate representation of this information prevents combinatorial explosion of the analysis. Despite the global analysis performed and the fact that the derived information is not weakened for a reduction in complexity, Symbolic Evaluation is believed to be implementable as a practical tool. Peripheral tools use the information in the data base to reason about the program without having to reanalyzing it. They control combinatorial explosion of the reasoning processes by heuristics. The interaction among peripheral tools is significantly improved by means of the shared program data base, since any information deduced by any tool can be stored in the data base for the benefit of all other tools. A large variety of tools important for program development, debugging and validation can be easily implemented on the basis of Symbolic Evaluation. To a varying extent, these tools can be implemented language-independently. The use of Symbolic Evaluation guarantees a consistent interpretation of the language semantics among all tools.

References

[1] Boyer, R.S., B. Elspas, and K.N. Levitt, *SELECT - A formal system for testing and debugging programs by symbolic execution*. in: Proc. of the International Conference on Reliable Software, Los Angeles, Calif., pp. 234 - 245, April 1975.

[2] Cartwright, Robert, and Derek Oppen, *Unrestricted procedure calls in Hoare's logic*. in: Proc. of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, pp. 131 - 140, January 1978.

[3] Cheatham, T.E., Jr., and Judy A. Townley, *Symbolic evaluation of programs --A look at loop analysis*. in: Proc. of the ACM Symposium on Symbolic and Algebraic Computation, pp. 90 - 96, August 1976.

[4] Cheatham, T.E., Jr., Glenn H. Holloway, and Judy A. Townley, *Program Refinement by Transformation*, Center for Research Computing Technology, Harvard University, TR-10-80, June 1980.

[5] Cheatham, T.E., Jr., *Overview of the Harvard Program Development System*. in: *Software Engineering Environments*, Huenke H. (Ed.), North-Holland Publishing Co., 1981.

[6] Clarke, L., *A system to generate test data and symbolically execute programs*, Dept. of Computer Science, University of Colorado, Boulder, Colo., Technical Report, CU-CS-060-75, February 1975.

[7] Cohen, Norman Howard, *Source-to-Source Improvement of Recursive Programs*, Ph.D. thesis, Division of Applied Sciences, Harvard University, May 1980.

[8] Cook, S., *Axiomatic and Interpretive Semantics for an Algol Fragment*, Dept. of Computer Science, University of Toronto, Tech. Report 79, February 1975.

[9] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, 1976.

[10] Floyd, R., *Assigning meaning to programs*. in: Proc. of the Symposium of Applied Mathematics, J.T Schwartz (Ed.), Mathematical Aspects of Computer Science, Vol. 19, pp. 19 - 32, American Mathematical Society, Providence, Rhode Island, 1967.

[11] Gries, David, *Compiler Constructions for Digital Computers*, John Wiley and Sons, Inc., New York, 1971.

[12] Hoare, C.A.R., *An Axiomatic Basis for Computer Programming*, CACM, Vol. 12, No. 10, pp. 576-583, October 1969.

[13] Hoare, C.A.R., *Procedures and Parameters: An Axiomatic Approach*. in: *Symposium on Semantics of Algorithmic Languages*, E. Engler (Ed.), Springer Verlag, New York, pp. 102 - 106, 1971.

[14] Hoare, C.A.R., and N. Wirth, *An axiomatic definition of the programming language PASCAL*, Acta Infomatica, Vol. 2, pp. 335 - 355, 1973.

[15] Howden, W.E., *Symbolic testing and the DISSECT symbolic evaluation system*, IEEE Transactions on Software Engineering, Vol. SE-3, No. 4, pp.266-278, July 1977.

[16] Igarashi, S., R.L. London, and D.C. Luckham, *Automatic program verification I: A logical basis and its implementation*, Acta Informatica, Vol. 4, pp. 145-182, 1975.

[17] King, J.C., *Symbolic execution and program testing*, CACM, Vol. 19, No. 7, pp. 385-394, July 1976.

[18] *ECL programmer's manual*, Center for Research in Computing Technology, Harvard University, TR-23-74, December 1974.

[19] McCarthy, John, *A basis for a mathematical theory of computation*. in: *Studies in Logic and the Foundation of Mathematics, Computer Programming and Formal Systems*, P. Brafford and D. Hirschberg (Eds.), North Holland Publishing Co., Amsterdam, pp. 33-70, 1963.

[20] Osterweil, Leon, *Using Data Flow Tools in Software Engineering*, Dept. of Computer Science University of Colorado, Boulder, Colorado, Technical Report, CU-CS-153-79, March 1979.

[21] Ploedereder, Erhard O.J., *Pragmatic Techniques for Program Analysis and* Verification. in: Proc. of the Fourth International Conference on Software Engineering, Munich, Germany, pp. 63-72, September 17-19, 1979.

[22] Ploedereder, Erhard O.J., *A Semantic Model for the Analysis and Verification of Programs in General, Higher-Level Languages*, Ph.D. thesis, Division of Applied Sciences, Harvard University, January 1980.
Also: Center for Research in Computing Technology, Harvard University, Technical Report, TR-02-80, January 1980.

[23] Schwartz, Richard, *An Axiomatic Semantic Definition of ALGOL 68*, Ph.D. thesis, Computer Science Department, University of California at Los Angeles, UCLA-34-P214-75, August 1978.

[24] Scott, Dana, and Christopher Strachey, *Toward a Mathematical Semantics for Computer Languages*, Oxford University Computing Laboratory, Technical Monograph PRG-6, Oxford, August 1971.

[25] Tarjan, R.E., *Applications of Path Compression on Balanced Trees*, JACM, Vol. 26, No. 4, pp. 690-715, October 1979.

[26] Washington Brown, Deborah, *The Solution of Difference Equations Describing Array Manipulation in Program Loops*, Ph.D. thesis, Division of Applied Sciences, Harvard University, February 1981.