PRAGMATIC TECHNIQUES FOR PROGRAM ANALYSIS AND VERIFICATION*

Erhard Ploedereder

Harvard University
Cambridge, Mass. 02138 USA

Abstract

The Program Development System (PDS) is a collection of programming tools created as an extension of the ECL programming system[23]. It contains components that assist the programmer in the definition and modular structuring of large programs at different levels of algorithmic abstraction. These components are supplemented by a program analysis package that produces an information pool to be used for such tasks as source-to-source optimization, semi-automated program documentation, fault detection and program verification.

This paper describes the core of the analyzing package, the Symbolic Evaluator. In its implementation we have incorporated pragmatic methods for handling data sharing patterns, and for characterizing and reasoning about the behaviour of loops and procedures.

The impact of these methods upon program verification techniques is briefly discussed.

I. Introduction

Despite major advances in the science of software engineering, the process of developing reliable, efficient, and transparent software is still very frustrating and often of rather limited success. We consider the absence of software tools assisting the programming process beyond a very elementary level to be the major cause for this situation. Lacking these tools, we are all too frequently inclined to undisciplined programming style.

In recent years various techniques have evolved to combat these problems. Proposals range from the educational level of programming disciplines (Bauer[2], Dahl[12], Dijkstra[13], Wirth[32]) to the development of linguistic constructs in programming languages. These constructs support the programmer in an improved specification of his intentions and force him to comply with certain restrictions eliminating language features considered harmful to program reliability. Examples include work on specification languages (e.g., Balzer[1]), modules and module interfaces (e.g., Parnas[24], Wirth[33]), abstract data types (e.g., Guttag[16], Wulf[34], Liskov[22]), elimination of aliasing (e.g., Horning[18]), restricted access to global variables (e.g., Shaw[26]) and so on.

Although these constructs and constraints can contribute significantly to the reliability of programs and facilitate the development of more sophisticated programming tools, there are limitations to this approach.

First, only so many restrictions can be imposed before a language becomes clumsy to use. Second, not all the proposed constraints can be enforced by purely syntactic means. For example the banning of sharing among reference parameters cannot be enforced syntactically whenever sharing depends on program values. Third, there are few language features that are not detrimental to program reliability, if used in undisciplined ways. Even the elementary concept of variables has been considered harmful (Bauer[3]).

While we believe in the importance of modularly structured and encapsulated software, as supported by our Program Development System (PDS)[9], we have adopted the viewpoint that, if applied cautiously and properly documented, a much wider range of language features than generally assumed can be used safely for the development of reliable and transparent software.

Therefore in our research we have emphasized pragmatic techniques for a powerful program analyzer permitting a wide spectrum of language features some of which have previously created substantial problems for practical program analysis.

Usually the notion of program analysis has been directly linked to methods of program verification. We have chosen to separate these

---

*Research reported herein was supported in part by
Naval Electronics System Command
Under Contract N00039-78-G-0020

Reprinted from PROCEEDINGS OF 4th SOFTWARE ENGINEERING, September, 1979

tasks. We scan the input program and attempt to derive a precise characterization of the values computed and side-effects created by the individual instructions of the program. Symbolic expressions represent the values of data-objects and computations occurring in the program.

We use the phrase "symbolic evaluation" to describe this process. The results of the analysis are collected in a program data base, which can be shared by the various tools that assist the program development. Among these tools are, or will be, the derivation and interactive validation of simple program specifications (e.g., lists of free variables in procedures), fault detection, program verification, interactive debugging, source-to-source optimization and static performance analysis.

The notion of symbolic evaluation of programs is not a novelty to the software engineering field. Various groups have proposed related approaches to derive and reason about symbolic expressions that describe program values occurring during an arbitrary evaluation of the program (Boyer[4], Clarke[10], Howden[19], King[21]).

Our efforts (Cheatham[8]), however, differ substantially from earlier work on symbolic evaluation :

1) We perform a static program analysis. Each instruction in the program is examined only once. Conditionalities that arise from control branches are absorbed in conditional symbolic expressions representing program values in join contexts.
2) Our work has focused on programs written in EL1, the base language of the ECL-system[23]. This language has compound data structures, pointers, facilities for data type abstraction and syntax extension, generic modes, procedure variables and a number of ways to exploit storage sharing (through reference parameters, pointers, identity declarations, and through "locative" results of procedure and function calls, blocks and other expressions in the language). The interesting semantic features of most higher-level languages are contained in EL1. Therefore our analyzing techniques are conceptually applicable to such languages as well.
3) We attempt to analyze the behaviour of loops (Cheatham[6]).
4) We derive templates that characterize the result and side-effects of user-defined procedures and we can thereby assess the effects of individual calls efficiently (Ploedereder[25]). This ability is of crucial importance for the applicability of our methods to large program systems.

The next section contains a brief introduction to the Symbolic Evaluator we have implemented for a subset of EL1. We then focus on the analysis of procedures.

A more detailed description of the Symbolic Evaluator and its components is provided by Cheatham[7,8], Ploedereder[25] and Townley[28,31].

II. The Program Analysis Package

II.1. The Program Data Base

The program data base contains the results of the symbolic evaluation. It has four basic parts, called the context graph, the store, the shadow, and the set of templates.

A context is a basic block of the program, a sequence of computations with no control branching in or out. Loops and procedure calls are treated as concurrent assignments to all affected variables and heap-objects. The context graph encodes all possible sequences of contexts; the analysis of loops and procedure bodies produces separate context graphs. Nodes in these acyclic graphs created for contexts after a branch operation contain the branch-enabling predicate. Edges in the graphs link contexts to their immediate predecessor contexts.

In order to distinguish among program points within a single context, we use a second coordinate called "time". It is incremented whenever an assignment or a variable declaration is evaluated.

We model the store by a scoped environment and a heap each consisting of a set of locations. For each variable we postulate a location in the scoped environment. Its representation contains the name of the variable, its mode and dimensions described by symbolic expressions, and a scope-label. The latter is used in conjunction with a scope-tree to resolve name-conflicts among declared variables. The scope-tree encodes the block structure of the program or procedure.

In addition, a location contains a list of value cells. Each value cell holds a symbolic expression for a value assigned to the location at some point in the program and a context tag (i.e., context and time) which identifies this point.

Locations in the heap are allocated for newly created heap-objects; the names in these locations are internal tokens, which can be assigned to pointer-variables of appropriate mode. Heap locations also contain mode and dimensions as well as a list of value-cells.

Sharing among locations (as induced by reference parameters or identity declarations) is represented by share-links. These links may be augmented with context tags or predicates to reflect conditional sharing. EL1 also permits sharing of a variable with components of a compound object; such partial sharing is recorded by associating symbolic selectors with the appropriate share-links.

The value of a component of an object is always recorded as part of the value of the entire object rather than in a separate location. Special symbolic expressions provide the basis for the correct representation of compound values and their components.

The context graph, the context tags in value cells and the share-links among locations provide the basis for the correct derivation of the possibly conditional value for a variable or heap-object at any point in the program.

The shadow is a computation tree representation of the input program in which implicit computations (e.g., dereferencing of pointers, type conversion, etc.) are made explicit; each program expression in the shadow is tagged with shadow information that describes the locative result ("L-value") of the expression and relates it to the context graph and the store. For expressions that return results on the stack not accessible via variables (e.g., arithmetic expressions) the shadow information also contains an R-value. Thus, there is no need to create locations in the store for intermediate results of nested expressions.

The set of templates will be discussed in more detail below. A template essentially is a separate program data base for a procedure, plus information about the applicability of the template with respect to different call environments.

II.2. The Symbolic Evaluator

The Symbolic Evaluator has four major components:

A) The Symbolic Interpreter is the controlling component of the package. It scans the input program sequentially and builds the program data base. It consists of a set of handlers each of which models a built-in instruction or group of instructions of the language, such as arithmetic and boolean functions, declarations, selections on compound objects as well as control functions (branches, blocks, conditional statements, etc.). These handlers also detect certain potential program exceptions (e.g., selectors out of range, mode incompatibilities, dereferenced null-pointers) (Townley[30]). Special analyzers are called whenever loops or user-defined procedures are encountered.

B) The Simplifier reduces and normalizes the symbolic expressions generated by the Symbolic Evaluator. It assumes the properties of operators in their idealized domains, e.g., associativity, commutativity, transitivity etc.

C) The Loop Analyzer replaces the values in all locations referenced but not declared within the loop by tokens, thus reflecting our ignorance about the values in these locations at the beginning of any but the first cycle of the loop; it then symbolically evaluates the loop statements, retrieves the final values in all these referenced locations and calls on Solvers to determine a closed-form representation for the actual value to be stored in these locations after the loop is exited.
The values before the loop, the tokens at the beginning of the loop and the corresponding symbolic expressions after the interpretation of the loop statements represent an n-dimensional first-order recurrence relation with a boundary condition described by the exit-enabling predicate.
Surprisingly many recurrence relations, even those involving conditionals, can be solved in closed form by our methods (Cheatham[7]). If solutions are found, they are assigned to the respective locations; otherwise the values in the affected locations are represented by a recursively defined symbolic expression (using a lambda-function[8]). To a limited degree, we are able to manipulate and reason about these recursive definitions as well.
As a side-product important for verification, the solvers determine symbolic expressions for the values at the beginning of the k-th cycle, where k is a token created by the Loop Analyzer. We denote this set of values, which are functions of the token k, as the general solution-set of the loop. We also obtain a symbolic expression for the total number of cycles taken by the loop.

D) The Procedure and Call Analyzer derives and applies templates that characterize the behaviour of user-defined procedures. Since this component is of crucial importance for the

practicability of the Symbolic Evaluator for large program systems, we will discuss its underlying principles in the following section.

II.3. The Procedure and Call Analyzer

There are two basic methods for analyzing user-defined procedures, commonly known as the "copy rule" and the "adaptation rule".

The copy rule corresponds to a textual expansion of each call by the respective procedure body augmented with instructions simulating the parameter passing. Although easy to implement, this rule is expensive in practice, since each call requires a re-analysis of the procedure. Also, it is restricted to non-recursive procedures.

The adaptation rule analyzes the procedure only once at declaration time and produces (or verifies) a complete characterization of the result and side-effects of the procedure applicable to any call. This characterization (henceforth called a template (Hantler[17])) has to be indeterminate with respect to influences exerted by the call-environments. At call time the template is retrieved and instantiated (i.e., the indeterminacies of the template are resolved up to indeterminacies in the call environment).

An analysis based on abstractions over all possible call-environments requires considerable effort. In particular, if tokens are not sufficient to represent the missing information (e.g., values of free procedure variables, aliasing) case-distinctions over all possibilities have to be made; the latter tend to make this type of analysis too costly to be practical.

In order to prevent the devastating effects of worst-case assumptions, one can impose restrictions on the language and require additional specifications for procedures.

We felt, however, that the required restrictions would be too stringent to be acceptable. Furthermore, in the spirit of interactive program development we prefer not to be dependent upon additional specifications during the process of symbolic evaluation. We rather consider them as voluntarily provided and possibly incomplete information, which is to be verified and completed in interaction with the programmer.

Hence, we have devised a strategy that strikes a balance between the generality and the complexity of the produced template. While we derive some information from a particular call-environment, we produce a template that is general enough to be applicable to subsequent similar environments. For substantially different call-environments we may have multiple templates.

We therefore distinguish the processes of
a) Procedure Analysis, which produces templates,
b) Template Matching, which determines whether an existing template is applicable to a given call-environment, and
c) Call Analysis, which applies the retrieved or newly generated template.

In order to demonstrate our methods, we define a small example in EL1-like notation:

```
BEGIN
  DECL PAIR:MODE LIKE VECTOR(2, INT);
  DECL SWAP:ROUTINE LIKE
    EXPR(LEFT:INT SHARED, RIGHT:INT SHARED)
      BEGIN
        DECL SAVE:INT BYVAL RIGHT;
        RIGHT := LEFT;
        LEFT := SAVE;
        IF LEFT # RIGHT THEN
           TRACE[I] := CONST(PAIR OF LEFT, RIGHT);
           I := I + 1
        FI
      END;
      .....
  DECL TRACE:SEQ(PAIR) SIZE 10;
  DECL I:INT BYVAL 1;
      .....
      calls on SWAP
      .....
END
```

This example introduces a mode-valued variable PAIR and a procedure SWAP, which exchanges two integers and keeps track of all relevant swaps on a free variable TRACE. Parameters are passed by reference.

The Symbolic Evaluator notes the existence of these variables and of the procedure constant. Upon encountering a call, it evokes the programs for Procedure and Call Analysis. At this point the information required about the call-environment is readily available. Nevertheless we analyze the procedure as if the call-environment were unknown. Only those facts that, when generalized, create undue complexity, are retrieved and used in generating a template of restricted applicability.

II.3.1. Procedure Analysis

The procedure analysis achieves relatively inexpensive generality in the produced templates by the following techniques:

a) Pseudo-locations represent the store-locations of free variables, formal parameters and accessible heap-objects. The symbolic evaluation of the procedure body treats these pseudo-locations as locations created in an imaginary block enclosing the body. In order to assess the effects of individual calls they are (conceptually) super-imposed on the corresponding locations in the call-environment.
In our example pseudo-locations are created for the parameters LEFT and RIGHT, and for the free variables TRACE, I and PAIR.
Pseudo-locations make the template independent of the locations associated with actual parameters and free variables in the call-environment. Thus a template generated for a call SWAP(A,B) may be applicable to a call SWAP(C,D) or SWAP(E[J],F[J,K]) or even SWAP(findpartner(B),B), where the procedure "findpartner" returns some locative result. If the pseudo-location covers only part of the call-environment location (i.e., for selections as actual parameters), the derived side-effects are recorded during call-analysis as assignments to components of the call-environment location.

b) Tokens are created to represent the initial values stored in pseudo-locations. These tokens are the sine-qua-non of general templates. They are instantiated during call analysis with their actual values, unless the actual value has been required during procedure analysis in order to produce meaningful results (e.g., the values of sub-called procedure variables). The latter situation is referred to as "forced instantiation"; the applicability of the template becomes dependent upon this instantiated value.
In our example five value-tokens, say LEFT*, RIGHT*, TRACE*, I* and PAIR*, are generated and allocated as the values of the five pseudo-locations in the initial context of the procedure body (i.e., value-cells are added to these locations). During procedure analysis the token PAIR* will be forcibly instantiated, since we do not wish to abstract from modes.

c) Tokens are created to represent the dimensions of free variables, parameters and heap-objects. Appropriate procedure entry-conditions are generated to prevent illegal component selections within the procedure body.
In our example only the free variable TRACE is of compound type. Its dimensions are represented in the pseudo-location by tokens, say DT1* and DT2*. A procedure entry-condition "I* GT 0 AND I* LE DT1*" is generated to validate the selection TRACE[I] in the procedure body.
These tokens make the template independent of the size of arrays. They are especially important for multi-dimensional parameters, e.g., the template for a procedure that exchanges two rows of a matrix will be independent of the size of the matrix.

d) The predicates and selectors for conditional and partial sharing among pseudo-locations are also represented by tokens. Since the actual values of these tokens are functions of values in the call-environment, these abstractions rarely prevent simplifications during procedure analysis.
If a template were generated for a call SWAP(A[M],A[N]) with a call-environment in which "M=N" symbolically evaluates to neither 'true' nor 'false', conditional sharing takes place between the pseudo-locations of the formal parameters LEFT and RIGHT. This condition is represented in the share-link by a token, say SP*.
The resulting template is also applicable to the calls SWAP(A[K],A[J]), SWAP(A,A) or to SWAP(A,B) with unrelated A and B, since the token SP* can be instantiated with the symbolic result of "K=J", 'true', or 'false', respectively.
If a template is generated for a call SWAP(TRACE[4,1],TRACE[5,2]), the formals LEFT and RIGHT are partially shared with the free variable TRACE. We abstract from the actual values '4', '1', '5' and '2', and obtain a template equally applicable to SWAP(TRACE[7,2],TRACE[4,1]).

The modes of free variables++ (and of actual parameters whose corresponding formals have generic modes), the existence of sharing relations among free variables and parameters passed by reference, and forcibly instantiated values are taken from the particular call-environment. The names of free variables are collected during the symbolic evaluation of the procedure body.

++ We are dealing with a dynamically scoped language; static scoping rules could be accommodated with some minor modifications to the Symbolic Evaluator.

Since this information tends to be unchanged in subsequent call-environments, expensive generality is avoided, while the resulting template is still applicable to a large set of different call-environments.

We call the set of pseudo-locations a "pseudo-environment". It provides sufficient information for the symbolic evaluation of the procedure body, generating a separate program data base for the procedure.

Side-effects of the procedure are determined by comparing the initial and final values in pseudo-locations for free variables, parameters passed by reference and accessible heap-objects.

The produced template contains the collection of applicability constraints, the program data base and a description of the derived side-effects (the returned result is given in the shadow of the body).

In our example the relevant contents of a template produced for a call SWAP(A[J],A[K]) can be described as follows:

TEMPLATE FOR (the value of) SWAP:

Applicability constraints:

1) The free variable TRACE must have type SEQ(VECTOR(2,INT))
2) The free variable I must have type INT
3) The free variable PAIR must have type MODE and value VECTOR(2,INT)
4) The actual parameters for LEFT and RIGHT must be shared under the condition SP*
5) No other sharing relations are permitted

Side-effects in pseudo-locations:

```
LEFT : initial value: L* , final value: R*
RIGHT: initial value: R*
       final value: cond(SP*, R*, L*)
I    : initial value: I*
       final value: cond(SP* OR R*=L*, I*, I*+1)
TRACE: initial value: T*
       final value: cond(SP* OR R*=L*, T*,
                    store(T*, I*, vector(R*, L*)))
```

Result: NOTHING

Entry-condition to be verified:

SP* OR R*=L* OR I* GT 0 AND I* LE DT1*,
where DT1* is the extent of the free variable TRACE in its first dimension.

END OF TEMPLATE

To encourage the hierarchical development of programs, we permit the user to specify procedures with bodies omitted. In these settings, we expect the user to provide a complete list of affected free variables. A call on a defaulted procedure is simulated by the introduction of tokens for the values in all affected locations of the call-environment. If the user wishes his program to be verified, he has to supply specifications about the result and side-effects of the defaulted procedure sufficient for deriving the correctness proof in all call-environments. The proof of these specifications is performed when a full definition replaces the defaulted procedure body, and is symbolically evaluated.

For example, a defaulted version of the SWAP-procedure could be

```
EXPR(LEFT:INT SHARED, RIGHT:INT SHARED)
  DEFAULT
    CHANGES(TRACE, I);
    USES(PAIR)
  END
```

The symbolic evaluation of the defaulted procedure body adds pseudo-locations for the specified free variables to the pseudo-environment. The final values in locations for parameters and for affected free variables are represented by new tokens unrelated to the tokens for the corresponding initial values.

The defaulted template for SWAP produced by the procedure analysis, is the previously displayed template without the derived entry-condition, and with the final values in pseudo-locations replaced by new tokens.

II.3.2. Template Retrieval and Call Analysis

The retrieval of a template for a call-analysis has to ensure the validity of its application:

Forcibly instantiated values, modes and the existence of sharing relations, on which the template is based, have to match the current call-environment, unless it can be shown that certain discrepancies do not influence the validity of the template. A typical irrelevant mismatch we detect arises from differences in sharing relations among pseudo-locations that are referenced but never assigned to in the procedure body. For tokens, arbitrary symbolic expressions can be substituted.

If a match is successful, the symbolic expressions representing the actual values for tokens in the template are determined. Some tokens may have no corresponding value, e.g., tokens for final values in defaulted templates, or tokens for the results of input operations within the procedure body; their actual value is a new token. The template tokens are then instantiated in the descriptions of the result of the procedure and of the final values in pseudo-locations. The Call Analyzer takes these instantiated values to model a concurrent assignment to the affected (components of) locations in the call-environment, and to describe the result returned by the call.

If no matching template exists, a new template is created by performing a procedure analysis, unless we are dealing with a recursive sub-call. In the latter situation the application of a defaulted template is simulated.

The tokens introduced by the application of a defaulted template for a recursive subcall, combined with the respective tokens generated for the still pending procedure analysis, the final values in the corresponding pseudo-locations, the values at the time of the subcall and the enabling predicate of the subcall provide the basis for reasoning about the results of recursive procedures. At the present time we have no general solvers for recursive procedures; however, for regular recursion many of the loop solving techniques are applicable.

While our method of analyzing procedures has been motivated by the desire to have no restrictions on storage sharing and to proceed with minimal a-priori specifications about the program, it allows us to model such language features as generic modes, procedure variables, parameters of type procedure, a simulated call-by-name parameter mechanism and a user-accessible evaluating function.

In addition, the presented techniques are immediately transferable to the analysis of rewriting rules++ (Conrad[11]) as provided by the PDS. This would not be the case, if we had sharing restrictions and specification requirements.

++ Rewriting rules provide a very flexible system for textual macros.

III. An Application: Program Verification

The principles of symbolic evaluation have far-reaching consequences for the methods of program verification.

We have implemented a small program verifier based on the results provided by the Symbolic Evaluator. In this section we discuss the differences between our approach and other, more conventional verifiers (e.g., Elspas[14], Good[15], Igarashi[20], Suzuki[27]).

Asserted predicates are symbolically evaluated by special handlers in the Symbolic Interpreter. The resulting symbolic expressions are derived from the program data base at the point where the assertion is given, but once established, they are invariant with respect to assignments in the program, since assignments affect only the location-value binding, but never symbolic expressions.

Hence the asserted symbolic expressions can be propagated forward and backward across program expressions without being changed.

The propagation out of and into branches of conditional expressions has to establish the usual logical connection with the branch-enabling predicate as provided by the context graph.

The tokens appearing in symbolic expressions asserted within loops represent program values at the beginning of an arbitrary but fixed cycle of the loop. A replacement of these tokens by the general solution-set of the loop will provide additional information. The forward or backward propagation out of the loop is accomplished by a replacement of the cycle-token k in the solution-set by the derived symbolic expression for the total number of cycles taken or by the constant "1", respectively. The propagation to the previous cycle, as required for the proof of invariance, is made by a replacement of the token k by the symbolic expression "k+1".

Likewise the entry- and exit-assertions of procedures have to be instantiated by substituting call-environment values for tokens of the template used in analyzing the call.

The ease with which knowledge can be propagated permits us to perform a backward directed generation of verification conditions with minimal effort. It allows us to search for additional premises from beyond delimiting assertions without transformations of the existing verification condition. Hence the frame problem is solved up to the heuristic decision of determining the relevancy of the readily available knowledge.

This has led us to a new approach towards VC-generation. We have abandoned the idea of single path verification. Instead we use incremental program segment verification. A program segment is the set of all paths leading from one assertion to the assertion currently under investigation; an EP-segment is a program segment delimited by the assertion to be proved and any unconditional ("essential") predecessor assertion.

Our method starts with the smallest EP-segment and derives premises for the verification condition from assertions within this segment. If these premises are not strong enough to carry the proof, the next larger EP-segment is consulted, and so on until no more premises are available or the search is terminated. Within EP-segments a sub-division in program segments lends structure to the retrieval of knowledge.

A rich set of assertive functions is provided in the specification language to allow a-priori guidance of these search algorithms. In addition we intend to create a user-interface permitting the user to interact with the search for relevant assertions.

Hence we produce and gradually strengthen only one verification condition per assertion. The proving component consisting of the Simplifier and a theorem prover (Townley[29]), lends itself to this incremental approach.

Unsuccessful attempts to prove assertions within loops based on their EP-segments inside the loop, cause an inductive proof. The respective theorems are obtained from the original verification condition by the cycle-index substitutions described earlier. For the induction step additional premises are collected from all assertions in the loop.

The generated verification conditions are usually more complicated than those produced by path verification, since all conditionalities are absorbed in the symbolic expressions. However, the proving component tries to obtain immediate success by resolving unconditional unit-clauses; only if this fails, it distributes the conditionalites in remaining clauses and tries to prove each case in turn. The conditions relevant to the assertion are rederived, while irrelevant conditionalities in the program do not cause any proving effort.

A further benefit of the solving capabilities of the Symbolic Evaluator is the fact that for those loops whose recurrence relations have been solved in closed form, trivial inductive assertions are not required.

The assertion language we currently support consists of the EL1-functions accepted by the Symbolic Evaluator, augmented by quantifiers and some convenient logical connectives (e.g., "implies") not in EL1. A facility for the non-recursive definition of predicates is provided by the rewriting rules and the use of user-defined procedures with boolean results. In the future we intend to allow the axiomatic introduction of user-defined predicates and symbolic functions. These axiomatic definitions will be subjected to symbolic evaluation, producing an algebraic characterization of transformations of the predicates by (groups of) program instructions. The resulting predicate behaviour patterns are independent of their external representations (choice of names for variables and of sequences of program instructions causing the same results); they will be applied not by pattern matching with the program text or with symbolic expressions, but rather as algebraic axioms in the proofs of theorems. Thus, this approach may provide the basis for "expert" predicate libraries very much like today's program libraries.

IV. Conclusions

An often cited disadvantage of symbolic evaluation is the potential combinatorial explosion of symbolic expressions caused by unresolved conditionalities in large programs.

Therefore special care is taken within the Symbolic Interpreter to resolve or reduce conditionalities as often as possible by context tag comparison rather than at the boolean level of path predicates.

In addition the Symbolic Evaluator gives up gracefully producing only partial analysis with weak results, when given a computation that is too intricate to solve in closed form within a reasonable amount of time, or when specifically instructed to do so.

The success of our approach depends on the fact that most computations in most programs are not pathologically complicated and that even partial analysis is often valuable (Cheatham[8]).

Furthermore we expect large programs to be divided into modules containing hierarchically structured control entities (procedures, rewriting rules) of relatively low complexity, each of which can be subjected to symbolic evaluation and analysis in as much isolation as feasible.

If, in spite of this division of concerns, a combinatorial explosion occurs, the analyzed control entity may be dealing with an abnormally intricate algorithm or the program may be badly structured.

In this case the user has to be content with a partial analysis. He has to provide sufficient assertive specifications to justify his program; these assertions should be validated by a program verifier. Should undisciplined programming be the major cause for the combinatorial explosion, the user is implicitly penalized for his programming style; hopefully he will rewrite his program to be more amenable to symbolic evaluation and its peripheral programming aids.

Thus the objective of understandable, well structured programs with good specifications in critical areas is attained without resorting to inflexible syntactic restrictions. At the same time, symbolic evaluation permits a wide range of language features and relieves the programmer of the tedious necessity to provide trivial program specifications. The results of the program analysis can be used by a number of tools assisting the process of program generation. These facts convince us that Symbolic Evaluation is well worthwhile, although it may be more expensive in its unit operations than conventional program analysis methods for more restricted languages.

With the fully implemented program design and analysis tools of the Program Development System we hope to achieve a large step towards more reliable and easily maintainable software.

Acknowledgements

The basic Symbolic Evaluator was originally designed and implemented by Prof. Thomas E. Cheatham, Jr. and Dr. Judy Townley, with assistance from Glenn Holloway. The author contributed subsequent minor modifications, the Procedure Analysis Package and the Verifier.

The author wishes to express his gratitude to Prof. Thomas E. Cheatham, Jr., Dr. Judy Townley and Glenn Holloway for numerous discussions related to these subjects and for commenting on this paper.

REFERENCES

(1) Balzer, Robert and Neil Goldman, "Principles of good software specification and their implications for specification languages", USC/Information Sciences Institute, undated.

(2) Bauer, F.L., "Programming as an evolutionary process", Proc. of the Second Conference on Software Engineering, San Francisco, 1976.

(3) Bauer, F.L., "'Variables considered harmful' und andere Bemerkungen zur Programmierung", Technische Universitaet Muenchen, Report Nr. 7519, Munich, undated.

(4) Boyer, R.S., B. Elspas, and K.N. Levitt, "SELECT - A formal system for testing and debugging programs by symbolic execution", Proc. of the Int. Conference on Reliable Software, Los Angeles, Calif., April 1975.

(5) Cartwright, Robert, and Derek Oppen, "Unrestricted procedure calls in Hoare's logic", Proc. of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, January 1978.

(6) Cheatham, T.E., Jr., and Judy A. Townley, "Symbolic evaluation of programs -- A look at loop analysis", Proc. of the ACM Symposium on Symbolic and Algebraic Computation, August 1976.

(7) Cheatham, T.E., Jr., and Deborah B. Washington, "Program loop analysis by solving first order recurrence relations", Center for Research in Computing Technology, Harvard University, TR-13-78, May 1978.

(8) Cheatham, T.E., Jr., G.H. Holloway, and Judy A. Townley, "Symbolic evaluation and the analysis of programs", Center for Research in Computing Technology, Harvard University, TR-19-78, November 1978; to appear in IEEE Transactions on Software Engineering.

(9) Cheatham, T.E., Jr., Glenn H. Holloway and Judy A. Townley, "A system for program refinement", Proc. of the Forth International Conference on Software Engineering, Munich, Germany, September 17-19, 1979.

(10) 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.

(11) Conrad, W.R., "Rewrite user's guide", Center for Research in Computing Technology, Harvard University, Memo, August 1976.

(12) Dahl, O.J., E. Dijkstra, and C.A.R. Hoare, "Structured Programming", Academic Press, New York, 1972.

(13) Dijkstra, E.W., "A Discipline of Programming", Prentice Hall, Englewood Cliffs, 1976.

(14) Elspas, B., "The semiautomatic generation of inductive assertions for proving program correctness", Research Report, SRI, Menlo Park, California, July 1974.

(15) Good, D.I., R.L. London, and W.W. Bledsoe, "An interactive program verification system", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

(16) Guttag, J.V., E. Horowitz, and D.R. Musser, "Abstract data types and software validation", CACM, Vol. 21, No. 12, December 1978.

(17) Hantler, S.L., and King, J.C., "An introduction to proving the correctness of programs", Computing Surveys, Vol. 8, No. 3, September 1976.

(18) Horning, J.J., "A case study in language design: Euclid", International Summer School on Program Construction, Munich-Marktoberdorf Germany, July 26 - August 6, 1978.

(19) Howden, W.E., "Symbolic testing and the DISSECT symbolic evaluation system", IEEE Transactions on Software Engineering, Vol. SE-3, No. 4, July 1977.

(20) Igarashi, S., R.L. London, and D.C. Luckham, "Automatic program verification I: A logical basis and its implementation", Acta Informatica, Vol. 4, No. 2, 1975.

(21) King, J.C., "Symbolic execution and program testing", CACM, Vol. 19, No. 7, July 1976.

(22) Liskov, B. and S. Zilles, "Programming with abstract data types", Proc. of the ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, April 1974.

(23) "ECL programmer's manual", Center for Research in Computing Technology, Harvard University, TR-23-74, December 1974.

(24) Parnas, D.L., "A technique for software module specification with examples", CACM, Vol. 15, No. 5, May 1972.

(25) Ploedereder, Erhard O.J., "Symbolic evaluation of user-defined procedures in EL1", Center for Research in Computing Technology, Harvard University, TR-01-79, February 1979.

(26) Shaw, M., and Wulf, W.A., "Global variables considered harmful", SIGPLAN Notices, Vol. 8, No. 2, February 1973.

(27) Suzuki, N. , "Automatic verification of programs with complex data structures", Ph.D. thesis, STAN-CS-76-552, Computer Science Department, Stanford University, February 1976.

(28) Townley, Judy A., "A symbolic interpreter for EL1", Center for Research in Computing Technology, Harvard University, Memo, November 1976.

(29) Townley, Judy A., "An incremental approach to resolution-based theorem proving", Center for Research in Computing Technology, Harvard University, TR-15-78, August 1978.

(30) Townley, Judy A., "Program analysis techniques for software reliability", Proc. of the ACM Workshop on Reliable Software, Bonn University, September 1978.

(31) Townley, Judy A., "The analysis of pointers in programs", Center for Research in Computing Technology, Harvard University, Memo, in preparation.

(32) Wirth, N., "Systematisches Programmieren", Teubner, Stuttgart, 1975.

(33) Wirth, N., "Modula: A language for modular multiprogramming", Software -- Practice and Engineering, 7, 1977.

(34) Wulf, W.A., "ALPHARD: Toward a language to support structured programs", Computer Science Department, Carnegie-Mellon University, April 1974.