INTERPRETATION IN A TOOL-FRAGMENT ENVIRONMENT

Steven J. Zeil
Edward C. Epp

COINS Technical Report 89–41
April 1988

Software Development Laboratory
Department of Computer & Information Science
University of Massachusetts
Amherst, Massachusetts 01003

This paper appeared in
The Proceedings of the 10th International Conference on Software Engineering

This work was supported in part by the National Science Foundation under grant CCR-87-04478, with cooperation with the Defense Advanced Research Projects Agency (ARPA Order No.6104), by the National Science Foundation under grant DCR-8404217, by Control Data Corporation under grant 84M103, and by RADC under grant F30602-86-C-0006.

Interpretation in a Tool-Fragment Environment

Steven J. Zeil and Edward C. Epp

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

The philosophy of composition of new software tools from previously created tool fragments is a useful approach to facilitating the development of software systems. This paper examines the extension of this philosophy to the design of program interpreters, demonstrating how the separation of interpretation into a core algorithm, value kind definitions, and computation model allows the capture of conventional execution models, symbolic execution models, dynamic data flow tracking, and other useful forms of program interpretation. An interpretation system based upon this separation, called ARIES, is currently under development.

1. Introduction

This paper discusses the role and design of program interpretation tools in a software development environment. We will examine the prospects for providing interpreters that satisfy two important characteristics of powerful development environments:

- support for a wide variety of activities occurring throughout the development life cycle, and
- support for the straightforward construction of new software tools via the composition of previously constructed tool fragments.

It is not the purpose of this paper to defend the utility of these characteristics, nor to discuss general environment architectures for their support. Instead, we propose to examine a useful class of tools that have not traditionally been designed with these characteristics in mind and to show how these characteristics affect those tools.

1.1 Varieties of Interpretation

A survey of activities that can take place within a comprehensive software development environment reveals an impressive range of activities where some form of program execution is required. In this section, we will examine some of these activities, with the goal of illustrating both the variety of computation models that must be encompassed and the common threads that run through those models.

Although it is not uncommon for a software development environment to center upon the use of some single language [9,13,11,14], there is certainly no requirement that all environments should do so. Indeed, as the scope of our environments widen to include direct support for more of the life-cycle, we can reasonably expect that environments will include not only a variety of implementation languages, but also distinct languages for specification, design, etc.

What may be less apparent, however, is that even for a given language there may still be a requirement for many different forms of interpretation depending upon the development activity being performed. Consider, for example, the following scenarios for the execution of the statement `A := B + C`, where A, B, and C are integer variables:

Actual Interpretation: One possibility is to request interpretation in a form that mimics, as closely as possible, the actions that would be taken by compiled code for that statement. This form of interpretation is the most familiar, and can serve as the basis for a wide variety of tools and activities that require more detailed monitoring/control of intermediate states of execution than are conventionally provided by compiled, machine-native code.

In this case, we might choose, for example, to represent integer variables as 32-bit binary numbers in 2's complement form. The semantics associated with '+' would be the familiar 2's complement addition, so that, if the initial values of B and C were the binary strings corresponding to 2 amd 3, respectively, then following interpretation of this statement we would expect the value of A to be the binary representation of 5.

Symbolic Interpretation: Another option is to represent the values taken on by variables as algebraic expressions that denote the computational history of those variables. Symbolic Interpretation has a variety of applications in software testing, verification, and analysis [2,7].

The semantics associated with '+' could then be to form a new algebraic expression with + as its root operator and with the symbolic values of B and C as the operands of that +. Thus, if the initial values of B and C had been x and $2 * x + y$, respectively, then after interpretation the value of A might be $x + (2 * x + y)$. We can achieve different varieties of symbolic interpretation by altering these semantics somewhat. For example, we might choose to say that the semantics of '+' involve forming a new algebraic expression as above, but then simplifying that expression to yield, in this example, $3 * x + y$ as the final value for A. Alteration of the semantics associated with conditional statements can yield the variants known as *path-dependent symbolic evaluation* [2] or *global symbolic execution* [1,2]. Symbolic and Actual interpretation can also be combined to yield *dynamic symbolic evaluation* [2].

Dynamic Data Flow Interpretation: An even less conventional form of interpretation can be achieved by letting the

This work was supported by grants DCR-8404217 and CCR-8704478 from the National Science Foundation, 84M103 from Control Data Corporation, and RADC grant F30602-86-C-0006.

value of a variable or expression be a variable name or a null indicator. This form of interpretation can serve to monitor many of the data-flow based testing metrics surveyed in [3].

For this purpose, the semantics of '+' is to check each of its operands and, if that operand's value is a variable name, to mark that variable as having last been *referenced* at this statement. (Similarly, the semantics of ':=' would mark it's right-hand-side operand, if not null, as having been referenced, but to mark it's left-hand-side operand as having been *defined* at this statement.) Many variations of dynamic data flow interpretation can be realized by minor changes in these semantics, including monitoring of other testing metrics or, when combined with symbolic interpretation, the generation of program slices [15].

1.2 Tool Fragments

There is little chance that any single interpretation tool could encompass the range of interpretation activities outlined in the previous section. Futhermore, the above descriptions were merely a sampling of the kinds of activities requiring interpretation that might arise within a development environment. One can reasonably expect the range and character of interpretation activities in an environment to evolve over time and to vary somewhat from one project to another.

It is worth noting that, in many instances, interpretation is not itself an end goal but is instead a part of some larger activity. The examples above may serve to suggest the existence of a variety of software tools within the environment, tools of which the interpreter is simply a component.

Software development environments must have the construction of new software tools as one of their major concerns, whether those tools represent the end product of some project or an intermediate tool to aid in the construction of that end product. One philosophy that has proven useful is an emphasis upon the composition of new tools from smaller, well-defined tool fragments [10,9,12]. The question that we wish to raise in this paper is, "Can we define a fragment architecture for interpretation that would encompass the varieties of interpretation outlined above and that would facilitate the construction of new software tools having interpretation as one component?"

As diverse as the above examples of interpretation activities may seem, there are clearly common threads that suggest that such an interpreter architecture should be possible. In each instance, the interpretation of `A := B + C` consisted of the following sequence of operations:

1. Fetch the current values of `B` and `C`.
2. Apply the + operator to those values.
3. Determine the location known as "`A`".
4. Apply the := operator to that location and to the value returned by the + operation.

What changed from one interpretation variant to another was:

1. the representation of *values* of variables and expressions, and
2. the semantics associated with the operators + and :=.

This suggests that we may be able to define an interpreter architecture comprised of a common core interpretation algorithm that determines the order in which values are fetched and operators invoked, a variant part providing the representation of values, and variant parts defining the semantics of the language operators.

2. Interpretation Semantics

2.1 Value Kinds

Conventionally, the values manipulated by programs are classified by domain and operation sets into different data types. The previous Section, however, suggests another classification, the *value kind*. Our examples have served to informally introduce three value kinds, which we call *actual*, *symbolic*, and *data flow*. Other value kinds are possible as well, but these three will suffice as examples.

The need for this new classification stems from a fundamental and sometimes confusing property of interpreters and interpreted code, namely the existence of two distinct levels at which we must view the objects and operations being manipulated during interpretation. At the level of the interpreted code, we have various objects subject to the typing rules of the language being interpreted. At the level of the interpreter, we have objects subject to the typing rules of the language in which the interpreter is written. The connection between these two sets of types can be tenuous, even if the language being interpreted and the language in which the interpreter is written are the same. It is possible that an integer object in the interpreted code has an integer-typed counterpart in the interpreter, but requiring that, for every object of type T in the interpreted code, there should be an object of identical type T in the interpreter would raise severe problems in most implementation languages, since such a correspondence would limit the ability to interpret code in which new types are constructed.

At one level, then, a value kind is simply a data type used by the interpreter to hold values during the course of an interpretation. Because these values also have types at the lower level of the interpreted code, we here reserve the use of the word *type* to refer to the lower-level classification and use *kind* to refer to the higher-level classification. There is a connotation, however, that may be safely drawn from the designation of some class of objects as a value kind – that we intend that set of objects and their associated operations to support one or more computation models.

2.2 Computation Models

Our interpreter architecture is based upon an expression-oriented view of programs. A language is considered to be a set of *syntactic operators* $\{f_i\}$ and a set of constraints on the legal ways in which expressions may be built using those operators. Typical operators would include operators for manipulating data (e.g., `addition`, `subtraction`, `array-component-selection`) and also operators for composing statements and groups of statements (e.g., `sequence`, `while-loop`). Constraints on the expressions built from these operators would include type rules such as "`addition` takes two integers and returns a single integer" or "`while-loop` takes a boolean-typed expression (the loop condition) and a statement list (the loop body) and returns a (compound) statement". A program is a legal expression built using those operators. The responsibility for determining that a given expression is a legal program lies with the tool that produces the expression (usually a compiler that converts source code into this expression-oriented form). The set of legal expressions and subexpressions of legal expressions in a language L will be denoted by E_L.

In a normal discussion of programming language semantics, one would take an expression from E_L and ask directly, "what is the meaning of this expression?" If an interpreter is intended to support multiple value kinds, this question may not have a direct answer. This is in part due to the different underlying representations employed by different value kinds. Were this not the case, however, it is still far from clear that the question has a direct answer. Representation-independent notations for expressing semantics seem to offer little meaningful information in this situation. For example, one might try to argue on the basis of the axiomatic semantics of real arithmetic that the result of executing `Z:=X+Y; Z:=Z-Y` is to leave `Z` and `X` equal, because "-" is the inverse of "+". This is not true in general for actual value kinds, however, because of possible round-off and overflow. It is not even approximately true for many data-flow value kinds, where `Z` would have a value indicating that `Z` and `Y` had been used in its calculation, which may or may not be true of `X` depending on the rest of the program's code. On the other hand, it is exactly true for most symbolic execution systems. Thus even very simple properties of the data and operations employed in a program may vary from one value kind to another. It is customary for any application making use of an unusual value kind to acknowledge these potential differences from the results that would be obtained using any given machine's actual representations [1,2]. Designers and users of tools must determine whether those differences are significant to the task at hand, whatever that might be.

We take the view that the meaning of a program expression cannot be determined in isolation from the value kinds that will be employed during the interpretation of that expression. Each value kind has associated with it a set of **semantic functions** $\{u_j : E_L^{n_j} \times s \rightarrow s\}$, where s is an interpretation state[1]. These semantic functions produce a new state from an old state and (potentially) from a collection of program expressions.

A *computation model* is a relation $\{(f_{i_1}, u_{j_1}), (f_{i_2}, u_{j_2}), \ldots\}$ that describes a binding of semantic functions to syntactic operators. The interpretation (denotation) of an expression $P = f_{i_k}(g_1, \ldots, g_n)$ under such a computation model is taken to be

$$I[P](s) := u_{j_k}(g_1, \ldots, g_n, s). \qquad (1)$$

For example, to accomplish actual interpretation of $P = E_1 + E_2$, where E_1 and E_2 are integer expressions, we would use a sematic function $u_+(E_1, E_2, s)$ that evaluates its operand expressions, ($s' := I[E_1](s); s'' := I[E_2](s')$) and then adds the results (the two "top" results in s''). By binding this u_+ to the + operator, we in effect declare it as the semantics of + in this model.

There may be many different useful computation models possible for a given value kind, if that value kind provides a sufficiently broad set of semantic functions that more than one meaningful set of bindings is possible. For example, for symbolic value kinds the semantic function associated with the syntactic operator "apply user-defined function `F`" might be to interpret the code associated with the function `F`, or might be to simply form a new symbolic value with `F` as the root operator (i.e. "$\mathtt{F}(\bar{x})$"), where $\bar{x}$ is the vector of current symbolic values of the actual parameters to this call of `F`).

A value kind may also provide operations that are not semantic functions but are useful to tools employing interpretation over that kind. For example, most symbolic kinds would provide some sort of algebraic simplification routine, which would probably not by itself constitute a semantic function that could be meaningfully bound to any syntactic operator, but which might be profitably employed by a tool (or within some semantic functions). Note that this operation has no meaningful counterpart in many other value kinds (e.g., actual or data flow), but this is not a problem since this operation would not be directly invoked by the interpreter in any case and would only be invoked by semantic functions or tools specifically designed for use with the symbolic kind.

2.3 Trace-Equivalent Models

Combinations of different value kinds are often valuable, requiring simultaneous support for more than one computation model. Our earlier examples noted applications requiring combinations of actual and symbolic and combinations of symbolic and data flow models. Applications requiring other combinations can be anticipated. We could, of course, require tool implementors to provide a new value kind and model that simply merges the information and operations of the appropriate combination of separate kinds, but concern for the reusability of code for implementing value kinds and for avoiding a combinatorial explosion in the number of value kinds has moved us to instead design our interpreter with the expectation of simultaneously supporting multiple models from one or more value kinds.

This support is not completely general. Some models are so incompatible that it is not even clear that "simultaneous interpretation" would have any useful meaning. For example, under actual execution and some forms of symbolic execution, the interpretation of an IF statement involves choosing either the `then` or `else` branch and requesting interpretation of that branch alone. Under *global symbolic* execution, the interpretation of the same IF involves interpreting both branches and combining the resulting states to form something like ($c \wedge$ *then-state*) $\vee$ ($\bar{c} \wedge$ *else-state*), where c is the symbolic value of the IF condition [1,2]. The term "simultaneous" would seem to have little meaning when applied to global symbolic and actual execution, although it may make perfect sense when used with other model combinations.

To characterize the models that can be supported in combination, we return to the definition of interpretation in equation (1). There we noted that many semantic functions would request interpretation of the subexpressions passed to them as parameters. Define a *trace* $T(P, s)$ of expression $P = f(\ldots)$ on state s under a given computation model as a tree of nodes labelled with syntactic functions as follows:

- The root of $T(P, s)$ is labelled by the syntactic function f.
- If the semantic function bound to f in this model, when executed in state s, would not request interpretation of other expressions, then the root of $T(P, s)$ is a leaf node.

[1]Because the interpretation state must include bindings of current values to variables, the structure of the state must clearly vary from one value kind to another. This level of detail, however, is not crucial to an understanding of the notion of a computational model.

- If the semantic function bound to f in this model, when executed in state s, would request interpretation of expressions $e_1 \ldots e_k$ in states $s_1 \ldots s_k$, then the root of $T(P,s)$ has children $T_1(e_1,s_1)\ldots T_k(e_k,s_k)$. The children are ordered according to the order of the interpretation requests when significant.

We will say that two traces are equivalent if every existing pair of nodes in corresponding position (relative to the root) have identical labels.

We allow simultaneous use of models that yield equivalent traces on the programs and states being interpreted. This means that most models resulting in the same flow of control through a program can be interpreted simultaneously. An important additional case satisfying trace equivalence arises when one trace includes nodes for which no corresponding nodes occur in the other. An example of this might arise during simultaneous actual and symbolic interpretation of ABS(X), where actual interpretation might require subsequent interpretation of the body of the ABS routine, but the symbolic interpretation might be content to immediately return "ABS(x)", where x is the current symbolic value of X, without actually conducting a symbolic execution of the code for ABS.

3. The ARIES Interpretation System

The ideas presented here are being used in the development of an interpretation system called ARIES, for ARcadia Interpretive Execution System, which will be a part of the Arcadia-1 environment [12].[3] In this Section, we sketch some of the key features of the ARIES design.

3.1 IRIS

Programs in the Arcadia-1 environment will be compiled into an internal representation called IRIS [5], and it is this internal representation that will be provided to the interpreter. IRIS, which stands for Internal Representation Including Semantics, is an abstract syntax graph that represents a program in terms of expressions. In essence, IRIS encodes everything as literals or as operators applied to a set of operand expressions. Thus the expression 2 + 3 is encoded as the application of an addition operator to the literals 2 and 3. A while-loop would be encoded as the application of a while operator applied to two operands, the first being an IRIS structure encoding a condition and the second an IRIS structure encoding a statement list.

A key feature of IRIS that separates it from other syntax graph representations (e.g., DIANA [6]) is that none of the operators in IRIS are predefined. Instead, the operator in each graph node is represented by a pointer to an IRIS structure representing the declaration of that operator, including such information as the operator's name, the number of operands it takes, the data types of those operands and of the returned value (if any), the in/out mode of the parameters, and whether each parameter is to be evaluated prior to invoking the operator's semantics (for example, '+' operators assume that their operands have been evaluated prior to performing the semantic action we call addition; the while operator does not expect its operands to have been previously evaluated but instead will, as part of its semantics, determine when and how often to evaluate them). There is essentially no difference between the declarations for the "predefined" operators for the programming language and for user-defined procedures and functions that have been compiled into IRIS. IRIS is therefore a general purpose structure for representing programs. To represent a given language in IRIS, one must provide the IRIS-structured declarations of the syntactic operators for that language. Different sets of primitive operators would yield different languages.

[3]The Arcadia project is a consortium effort that includes researchers from the University of California at Irvine, University of Colorado, University of Massachusetts, Stanford University, Aerospace Corp., Incremental Systems Corp., and TRW.

3.2 Dynamic Manipulation of Computation Models

The set of operators defined in a given IRIS graph constitute the set of syntactic operators during interpretation of that graph. In practice this set includes both language-primitive operators and operators representing user-defined functions and procedures. Semantic functions must be bound to each operator for each model supported by a given instance of ARIES.

In ARIES we have chosen to allow these bindings to be altered dynamically. This means that a tool is actually able to change computation models during the course of an interpretation. We do require that the value kinds and the number of models to be based upon each value kind be declared when the tool's version of ARIES is instantiated. Also, current environmental limitations require that the semantic functions be compiled as part of the tool. Thus the pool of semantic functions from which models are constructed must also be determined at compile time. Within that pool, however, one still has considerable flexibility in the construction/alteration of a model. The advantages of allowing dynamic alteration of the computation models are:

- It provides a simple mechanism by which tools can exert control over the interpretation process. For example, during interpretation of a program that reads a large amount of input, a tool might wish to begin taking inputs from a file but then allow the tool user to add additional items from the keyboard. This can be accomplished by altering the binding to various I/O operators at the appropriate time. Similarly, a debugger might wish to dynamically choose whether an operator representing a user-defined routine should be interpreted by simply invoking a previously compiled-into-native-code version of that routine or instead by interpreting the code for that routine, depending on whether the internals of that particular routine were of interest to the person employing the debugger. Similar choices have earlier been described for symbolic execution of functions and procedures like ABS. These choices are easily achieved under ARIES by changing the semantic functions bound to the relevant syntactic operators.
- Dynamic manipulation of these bindings also provides a natural treatment for dynamic creation of new procedures and functions to be interpreted. Many tools slated for implementation under Arcadia-1 involve incremental changes to the code coupled with immediate interpretation (e.g., [17]).
- Tools may also easily adjust the response of the interpreter to exceptions and run-time errors by changing the bindings of exception handlers, etc., in response to changing tool requirements.

We have found it convenient to define one special model, called the traversal model, whose values consist of IRIS expressions and whose activation stack is used to summarise the control flow information for each process. The use of this value kind has the advantage of providing a mechanism by which trace equivalence can be readily enforced, of providing a standard means by which tools may gather control flow information no matter what other value kinds and models are employed, and of providing a basis for a set of standard semantic functions for describing control-related semantics that are common to many models. For example, the meaning of an IF statement can change from model to model (as in global versus non-global symbolic execution) but the "standard" definition is sufficiently common that it is worth providing a commonly accessible semantic function defining it, which we do using the traversal kind. This does not prevent a tool from binding a new function to the IF operator; it merely provides a useful default.

3.3 ARIES Components

ARIES has been partitioned into four conceptual levels, as illustrated in Figure 1:

1. The *Tool* is the highest conceptual level. Operations occurring at this level will vary considerably from tool to tool.

2. The second level is the *Tool Interpreter Interface* (TII). This level serves as an interface between the particular requirements of a tool and the details of the interpreter. It is at the TII level that the interpreter becomes language and tool specific via the importation of the appropriate value kinds and of the appropriate semantic functions.

3. The third level is the language and tool independent *Atomic Interpreter* level. Its primary functions are to traverse the IRIS structure and select the semantic procedures that have been bound to the syntactic operators.

4. The lowest conceptual level consists of the set of semantic functions, which implement particular operators for particular value kinds.

These four levels are discussed in more detail below. Because the TII serves primarily as the interface between levels, we reserve its discussion until after the functions of the other levels have been made clear.

3.3.1 The Tool

Although the set of tools that will employ interpretation is not fixed, we do know that tools will need to set up interpretations of arbitrary IRIS structures, to run the interpreter, to halt interpretation at appropriate times, and to then gather information before resuming interpretation. A typical scenario for tools is therefore:

```
         :
State := initial_state(Code_To_Be_Interpreted);
while not done(State) loop
     run_interpreter (State);
     collect information from the state;
end loop;
         :
```

The operations initial_state, done, and run_interpreter may vary from tool to tool. Their precise definition occurs in the Tool-Interpreter Interface. Other high level functions in which a tool might be interested include altering operator bindings, altering the interpreter state, and comparing the results of different states.

3.3.2 The Atomic Interpreter

The atomic interpreter knows nothing about specific languages, tools, and value kinds. It treats the IRIS structure as an expression to be evaluated. Most of its work is therefore concerned with determining what IRIS node is currently being executed, whether that node has operands that require evaluation, and what semantic functions should be invoked to execute that node. IRIS contains information about the expected evaluation patterns of many operators, in that it allows indications as to whether an individual operand is to be evaluated prior to invoking the operator's semantic function. Thus the atomic interpreter can follow a default evaluation order for operators such as "+", "<", or ":=", but allow other operators such as "IF" or "LOOP" to leave all decisions on interpretation of their operands to the semantic functions.

Information about which node in the IRIS structure is the next to be interpreted and which nodes have already been visited is kept on the activation stack for the traversal value kind. A rough sketch of the atomic interpreter's interaction with this traversal information is:

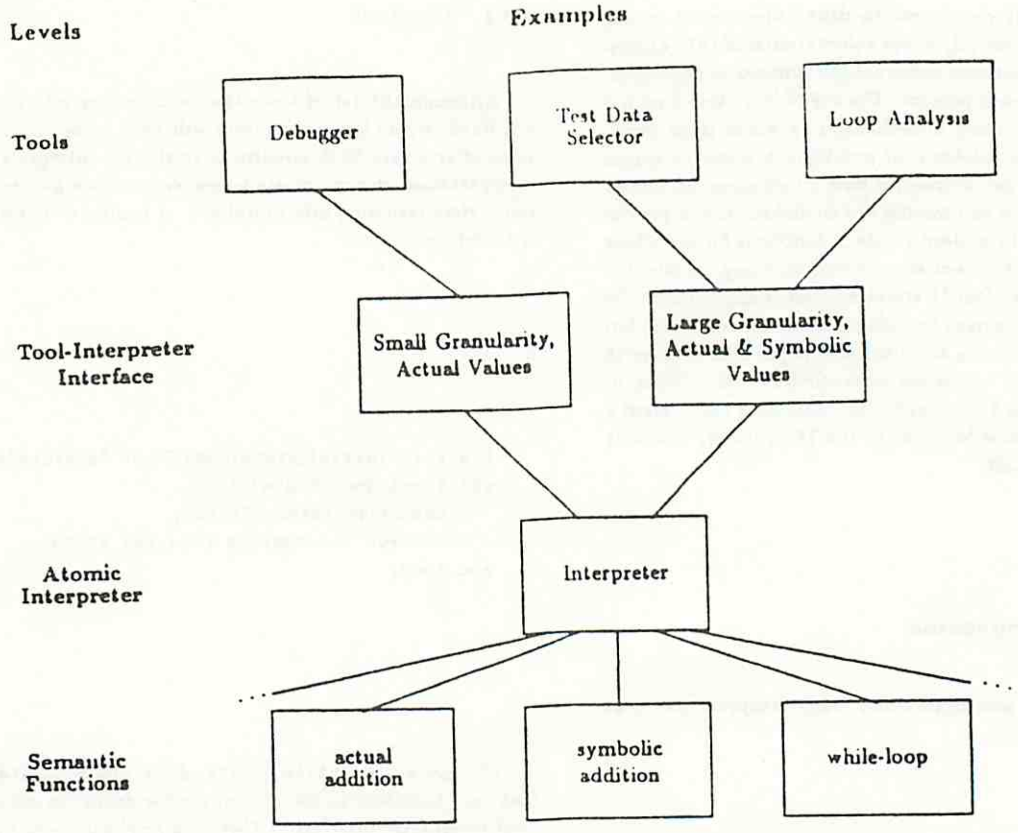

Figure 1: Levels of Abstraction in ARIES

```
Get the current node from the top of the
        traversal model stack;
Determine the current node's operator;
If all operands of the current node have not
        been processed then
    Get first unprocessed operand;
    If this operand is to be interpreted before
            invoking the operator's semantics then
        Push root node of operand onto stack;
            (Subsequent calls to the interpreter will
             result in the evaluation of this operand,
             eventually returning to the current node.)
    end if;
else
    invoke the semantic functions bound to the
            current operator for each
            computation model;
end if;
```

From this sketch, it can be seen that a single call to the atomic interpreter results in a change of state to force subsequent evaluation of an operand of the current node or in the invocation of the semantic functions for the current node's operator. Implicit in this simplified view of the atomic interpreter is the idea that the values returned by evaluated operands (e.g., any operands of the + operator) and the IRIS structures representing unevaluated operands (e.g., the statements in the "then" and "else" parts of an if operator) are all collected and eventually passed to the semantic functions.

3.3.3 Semantic Functions

The purpose of the semantic functions has already been discussed at length. The iterative approach to ARIES means that some of these functions (those not employing the default operand evaluation order) must examine the interpreter state to determine which of the operands have been interpeted and adjust the state to force interpretation of the other operands, as necessary. For example, the usual semantic function for the while operator would alternately modify the interpreter state to switch interpretation between its "condition" and "statements" operands until the evaluated condition is false.

Some semantic functions actually cut across the value-kind boundaries. For example, user defined operators representing procedures and functions to be interpreted can be evaluated with the semantic procedure interpret-body. Interpret-body will do the appropriate state manipulation so that subsequent calls to the interpreter level execute the IRIS-encoded body of the user-defined operator. In essence all the semantic procedure interpret-body does is change state in the IRIS-encoded program from one IRIS operator to the IRIS-encoded subtree that defines the semantics for that operator. Any IRIS operator that has an IRIS representation of its body may be interpreted via interpret-body in any model.

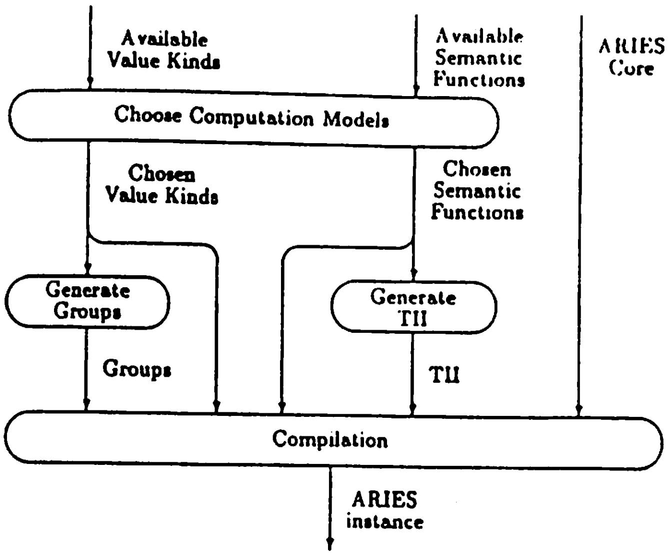

Figure 2: Creating an Instance of ARIES

3.3.4 The Tool-Interpreter Interface

All interactions between the tool and the interpreter pass through this interface. What those interactions are will depend on the tool. For example, a testing tool might view the program with large grain, being interested only in the final results and not in any intermediate values. On the other hand, an interactive debugger might require very fine-grained control over the interpretation, allowing execution to be halted at any node.

The TII is the customizing agent. It collects the objects and operations that might change from one tool to another. TII's may be unique to a particular tool, or tools with very similar interpretation requirements may share a common TII. The major components defined in the TII are:

- the value kinds to be employed during interpretation;
- the pool of semantic functions that may be bound to IRIS operators;
- the `RunInterpreter` procedure that repeatedly invokes the atomic interpreter until a state is reached in which the tool should regain control; and
- procedures to initialise the interpreter state and to establish default bindings of semantic functions to operators.

3.4 Generic Interpretation

ARIES is an interpreter skeleton that can be instantiated, much as one instantiates an Ada generic, to yield a particular interpreter. Thus each tool that requires interpretation to accomplish some task can have an instance of ARIES tailored for its own use. Of course, if the tool requires a value kind or a semantic function that has not been used in any previous tools, then that kind or function must be implemented before the instance of ARIES can be created.

We expect that the development of new value kinds will be quite rare, but that many tools will need a few customized semantic functions (e.g., different test path selection tools will each want to supply a new semantic functions for `if`, `loop`, etc., that choose branches in accordance with different path coverage metrics).

Figure 2 shows the process whereby an instance of an ARIES interpreter is created from a set of existing value kinds and semantic functions.

The tool builder must choose the appropriate computation models, meaning that the number of models, the value kind for each, and the semantic functions that may possibly be bound to operators during the interpretation must all be chosen.

The chosen value kinds are used to generate a set of Ada packages defining a *group* for each of the abstract data types defining the value kinds. A group is, intuitively, an array indexed by computation model of the corresponding data types at some level of the hierarchy. For example, a value group might contain one traversal value, one actual value, and one symbolic value, if those were the value kinds for the models employed in a particular tool. Currently, these group packages are created by hand, but there seems to be no fundamental barrier to having them created automatically.

The chosen semantic functions are grouped together, along with any tool-dependent code defining such interface behaviors as the conditions under which control is passed back to the tool, to form the TII. There is some opportunity for automatic aid in the formation of the TII, but the extent of possible automation of this step is still uncertain.

Finally, the chosen value kinds and semantic functions, the generated group packages, the TII, and the ARIES core can be compiled to yield a customized interpreter for the given tool, as shown in Figure 2.

4. Summary and Current Status

There is considerable variation in the forms of interpretation that may be required in a versatile software development environment. The decomposition of interpretation into an appropriate set of tool fragments can aid in the construction of the software tools that include some form of program execution as a part of their functioning.

We have suggested the separation of the core interpretation algorithm from the questions of representation of values (value kind) and of the binding of semantics to language operators (computation models) as one such decomposition that permits the capture of a wide variety of interpretation activities. This separation has been employed in the design of the ARIES interpretation system.

Work is now continuing on the development of ARIES-based interpreters. Our first instantiations of ARIES will be for the Ada language, with expansion into Ada-related languages such as the Anna specification language [8] and the PIC interface-control language [16] expected in the near future. As of this writing, we have successfully demonstrated actual interpretation of some simple Ada programs and are beginning to develop the semantic functions for symbolic interpretation.

The authors would like to acknowledge the work of Srinivasmurthy Acharya, Rahul Bose, and Mark Gisi on the implementation of ARIES, and the many comments and suggestions of the members of the Arcadia consortium that helped to shape the initial design.

REFERENCES

[1] T. E. Cheatham, G. H. Holloway, and J. A. Townley, "Symbolic Evaluation and the Analysis of Programs", *IEEE Transactions on Software Engineering*, SE-5, 4, July 1979, pp. 402-417

[2] L. A. Clarke and D. J. Richardson, "Symbolic Evaluation Methods - Implementations and Applications", *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (eds.), 1981, North-Holland Publishing Co.

[3] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A Comparison of Data Flow Path Selection Criteria", *Eighth International Conference on Software Engineering*, August 1985, IEEE, pp. 244-251

[4] E. C. Epp and S. J. Zeil, *ARIES: A Multi-Lingual Interpreter for a Tool-Fragment Environment*, COINS Technical Report 86-57, University of Massachusetts at Amherst, December 1986 (revised May 1987)

[5] D. A. Fisher, "IRIS Arcadia Presentation", *Arcadia Document INC-86-03*, Incremental Systems Corporation, Pittsburgh, April 1986

[6] G. Goos, W. A. Wulf, A. Evans, Jr. and K. J. Butler, *Diana: An Intermediate Language for Ada*, 1983, Springer-Verlag

[7] S. Hantler and J. King, "An Introduction to Proving the Correctness of Programs", *ACM Computing Surveys*, vol. 8, no. 3, September 1976, 331-353

[8] D. C. Luckham and F. W. von Henke, "An Overview of ANNA, a Specification Language for Ada", *IEEE Software*, vol. 2, 2, pp. 9-24, March 1985

[9] L. J. Osterweil, "Toolpack - An Experimental Software Development Environment Research Project," *IEEE Transactions on Software Engineering*, vol. SE-9, 6, pp. 673-685, November 1983

[10] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", *Bell System Technical Journal*, vol. 57, no. 6, July-August 1978, part 2

[11] R. N. Taylor and T. A. Standish, "Steps to an Advanced Ada Programming Environment", *IEEE Transactions on Software Engineering*, SE-11, no. 3, March 1985, 302-310

[12] R. N. Taylor, L. A. Clarke, L. J. Osterweil, J. C. Wileden, and M. Young, "ARCADIA: A Software Development Environment Research Project," *IEEE Computer Society Second International Conference on Ada Applications and Environments*, April 1986

[13] W. Teitelman and L. Masinter, "The InterLisp Programming Environment", *Computer*, vol. 14, no. 4, April 1981, 25-33

[14] W. Teitelman, "A Tour Through Cedar", *Proceedings of the Seventh International Conference on Software Engineering*, September 1982, IEEE, 58-67

[15] M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering*, SE-10, no. 4, July 1984, 352-357

[16] A. L. Wolf, L. A. Clarke, and J. C. Wileden, "Interface Control and Incremental Development in the PIC Environment", *Proceedings of the Eighth International Conference on Software Engineering*, London, England, August 1985

[17] S. J. Zeil, "The EQUATE Testing Strategy", *Proceedings of the Workshop on Software Testing*, July 1986, IEEE, pp. 142-151