META-EVALUATION AS A TOOL FOR PROGRAM UNDERSTANDING

Robert Balzer, Neil Goldman and David Wile
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina Del Rey, California 90291

ABSTRACT

Formal program specifications are difficult to write. They are always constructed from an informal precursor. We are exploring the technology required to aid in the construction of the formal specification from the informal version.

An informal specification differs from a formal one in that much information which the writer believes the reader can infer from the context has been supressed from the specification. Resolution of the supressed information depends upon information contained in other parts of the specification and upon Knowledge of what makes a specification well-formed and the ability to model the parts of the specification interacting with one another.

This paper describes the technology used in a running system which embodies theories of program well-formedness and informality resolution within the context established by symbolically executing the program to systematically discover the intended meaning of each informal construct within an informal specification.

KEYWORDS: Meta-Evaluation, Symbolic Execution, Informal Specification, Program Specification, Understanding Systems, Informality Resolution, Program Well-Formedness

INTRODUCTION

Producing a good specification has been recognized as a critical precursor to producing *an* acceptable software implementation. Considerable effort has been expended to produce better formalisms for software specification. We believe, however, that the difficulty lies in the formalisms themselves and that an aid in creating such formalisms, rather than a better formalism, is required.

Since software specifications are always first created in an informal language and then converted, external to any computer system, to some formalism, a system which aided this conversion process, from informal to formal, would significantly aid the specifier.

NOTE: This research was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DAHC15 72 C 0308, ARPA Order No. 2223, Program Codes 3D30 and 3P10.

ACKNOWLEDGEMENT: We are deeDly indebted to Professor Herbert Simon for his comments on this work which have deepened our understanding and sharpened our perception of its relation to his pioneering work with Professor Newell in understanding ill-formed problems.

We are constructing such a system, called SAFE [1], which accepts an informal software specification as input and produces a formal operational equivalent (see [1] for example). Most of the transformation is accomplished automatically via the techniques described in this paper, but some interaction with the specifier is also required to resolve particular informal constructs for which insufficient context exists.

This system consists of three phases: (D a Linguistic Phase which acquires a model of the domain [2] and identifies the individual actions to be performed, (2) the Planning Phase which creates a control structure for these actions, and (3) the Meta-Evaluation Phase which is the focus of this paper.

The purpose of the Meta-Evaluation process is to simulate the run-time environment of a program to provide the context for disambiguating informal constructs contained in the program description. It thus must provide three separate capabilities: (1) the ability to simulate the state of a program as it is being executed, (2) the ability to form an ordered set of hypotheses for the intended meaning of an informal construct, and (3) the ability to test these hypotheses against some criteria. The second of these capabilities represents a theory of informality resolution for program specification; the third, provides an operational theory of well-formed programs which eliminates hypothesis which do not satisfy the rules of this theory; while the first provides the data for testing these well-formedness rules.

The combination of these three capabilities provides a mechanism for effectively applying our theories of informality resolution of program specifications and of program well-formedness to the task of understanding informal program specifications. The following sections describe the major features of each of these capabilities and is followed by an example illustrating the interaction between them as an informal program specification is Meta-Evaluated.

However, before describing the capabilities, we must first consider the language in which the program to be disambiguated is expressed and the types of informality which are allowed.

THE PROGRAM MODEL

As we mentioned, the Meta-Evaluation process is the third and final phase of a larger system [1] which deals with a wide range of informal constructs in program specifications and starts from a parsed version of a natural language program specification. This system acquires (or augments) a description of the relevant domain in which the specified program will operate. In this regard, it is very similar to Simon's UNDERSTAND [3] system as it determines what objects exist in the domain, how they relate to other objects, what constraints they must satisfy, and how they are to be manipulated by the program being specified.

This work has been described elsewhere [4} Here we are concerned with how the acquired domain is represented, how the specified program is expressed, and what informal constructs remain unresolved.

We begin with our model of what a program should be, which we feel is central to the success of our system. This model is derived from the desire to minimize the translation from the informal natural language specification, to avoid issues of representation and optimization (which have colored

many other program models), and to keep the semantics of the programs as simple as possible so that programs could be understood and composed by our system.

Although our program model was largely derived from concerns of simplifying our system's task of resolving informal program specifications, we strongly believe that this program model (with suitable syntactic sugar) is also appropriate for people to express formal unambiguous operational program specifications.

To avoid issues of data representation, the most uniform representation known, and one which closely mirrors the original parsed natural language specification, was selected. This representation is a fully associative relational data base and is used to hold all data manipulated by the program. An object in this data base can be thought of as a named point in space whose meaning is defined totally by the other objects (points) and it is connected to by relations (lines).

The only actions (changes) allowed in this data base are the creation and destruction of named objects and the making and breaking of relations between them. In addition, information can be extracted from the data base in a side-effect free manner (i.e., the extraction mechanism does not change the data base) via a pattern-match language. This language enables the full associativity of the data base to be used to access any object connected to a named object via the appropriate relation. Any object so accessed may be bound to a placemarker which may then be used to access further objects, and so on. Placemarkers once bound by a pattern-match are never rebound. They are merely an indirect reference to the named object to which they are bound.

Placemarkers have completely replaced variables in our programming model (which contains neither variables nor assignment statements) and their semantics are particularly simple. They are bound only via a pattern-match to a named object in the data base, and once bound, they are not rebound. Thus, they provide the means for focusing attention on some portion of the data base and of accessing further information associated with the referenced named object.

There is one exception to the rebinding rule. Inside of a loop (which takes the form of TOR ALL <pattern> DO <statement>) all placemarkers bound in the iteration pattern are rebound on each successive iteration so that a different named object (or named objects if more than one unbound placemarker appears in the iteration pattern) can be accessed and manipulated by the loop body.

The only data manipulated by the programming model are patterns composed of relations and the operations AND, OR, and NOT. Each relation has arguments which must be a named object, a function which evaluates to a named object, or a placemarker. The placemarker must either be bound to a named object or be unbound. If an unbound placemarker occurs in a pattern being retrieved from the data base, then if the pattern is successfully matched with some portion of the data base, the placemarker is bound to the corresponding named object. If the match is unsuccessful, then the placemarker remains unbound.

The control statements available are a subroutine call, a sequence of statements, a conditional statement, an iterative statement, and a demonic statement. The conditional statement OF <pattern> THEN statement-1 ELSE statement-2") causes statement-1 to be executed if the pattern is matched and statement-2 to be executed otherwise. The iterative statement (TOR ALL <pattern> DO statement-1") causes statement-1 to be repeatedly executed for each portion of the data base which matches the pattern with the placemarkers in the pattern bound to the named objects in the matched portion of the data base. The demonic statement ("WHENEVER <PATTERN> DO statement-D causes statement-1 to be executed whenever a relation is added to the data base which enables the patten to be matched.

Finally, to prevent the intrusion of representation considerations, the associative relational data base supports inference so that the distinction between explicit and implicit (computed) data can be ignored.

Thus, to first order our programming model represents the integration of the data handling of a fully associative relational data base and the control aspects of a conventional programming language. We believe that this combination provides a particularly simple basis for stating and analyzing unoptimized operational program specifications, and hence, provides a solid foundation for our work on informality resolution.

PROGRAM SIMULATOR

The purpose of the program simulator is to simulate the run-time environment which will exist at each step in the execution of a program to provide the data to resolve informalities in the program. The complexity of this capability arises from our desire to simulate the run-time environment for a "typical" execution rather than for some particular set of input data. In essence, we wish to represent the run-time environment as a function of some prototypal state.

The technique of Symbolic Execution [5-12] was developed to symbolically express the output as a function of the inputs. This technique has generally been applied to numeric problems where well known simplifications and theorems exist which prevent the resulting expression from becoming overly complex. However, even with these simplifications the complexity of the output expression is such that individual paths through the program are normally explored one at a time.

In non-numeric problems the simplification techniques are much less developed and the expressions describing the state of the computation become very complex. Particularly difficult are loops and conditional statements. Loops require the use of universal quantification over the loop predicate as the condition which controls application of the loop body. Conditional statements require a splitting of the computation state into cases controlling which branch of the conditional will be executed.

The alternatives for dealing with this complexity are quite clear; either it must be mastered, or it must be avoided. The majority of researchers in the field have pursued the first alternative and are working on theorem provers and simplification systems better able to cope with these complexities. Compiler writers, on the other hand, have avoided this complexity in such techniques as data flow analysis by recognizing that for their purposes, it is not important to know the exact circumstances under which some particular data will be accessed, but only that there exist some

(unknown) circumstances under which it can be accessed. Their particular needs allow a much weaker form of analysis than symbolic execution to be applied to the program, avoiding the complexity.

In a similar way, our use of the "analysis" of the program is not to describe the outputs as a function of the input, but rather to resolve informalities in the program itself. For this reason, a weaker form of program interpretation, which we call Meta-Evaluation, is adequate. This technique avoids complexity by only executing each loop once (the informalities within the loop must make sense during the first execution) and by picking an arbitrary branch of conditional statements for execution (informalities following a conditional statement must make sense no matter which branch was executed).

In addition, rather than representing the state of the computation as a simple compound expression, we represent it as the running program (in our program model) would, as a set of relations in the associative data base. As Meta-Evaluation proceeds and control passes from statement to statement in the program, this data base is altered to reflect the additions and deletions specified in the program. Thus, the data base will reflect the state of the run-time data base for the program as control reaches each statement in the program. This simulation of the run-time data base enables each statement to be Meta-Evaluated in an appropriate environment which provides the context to resolve any informalities in the statement and to test the program for well-formedness.

Simulating this data base as execution proceeds through the program would be quite simple if some particular set of input data were selected. However, this data base must represent the program's behavior on arbitrary input data. Therefore, symbolic data must be created and the data base expressed in terms of this symbolic data.

Once we recognize that the input data to any program expressed in our program model consists of those relations in the data base which it accesses without having previously created, the representation of symbolic data in the data base becomes quite simple. A program simulation is started with an empty data base. Whenever the program attempts to access the data base (except in the predicate of a conditional statement) the following rules are applied. If the accessed pattern already matches data existing in the data base, then the pattern match proceeds normally binding any placemarkers in the pattern to the corresponding named objects in the data base. If$_t$ on the other hand, the pattern does not match existing data, then new symbolic data is created (and assumed to be part of the input data to the program) so that the pattern match can succeed.

The rationale for creating new data to match the accessed pattern is that the program has assumed that this data already exists because it is unconditionally accessing it. Hence, unless that data does exist, the program will not operate correctly. Therefore, to enable the program simulation to proceed, suitable data is created to satisfy the accessed pattern. However, only the existence of named objects rather than their particular identity can be inferred for arguments in the pattern specified by unbound placemarkers. Therefore, new "symbolic" instances of the appropriate type of object are created as part of the assumed relation.

As Meta-Evaluation proceeds, more and more of the input data for the program is created because it is accessed by the program and does not already exist. Although the named objects in this data base are "symbolic" in that their identity is unknown, they are manipulated by the program just like actual data. As data is accessed by the program, placemarkers are bound to these "symbolic" data, and the program creates new relations involving these objects and/or deletes old ones.

Occasionally constraints on the data base, such as a particular relation being single-valued, will enable the identity of a "symbolic" object or the equivalence of two different "symbolic" objects to be determined. When this occurs, the Meta-Evaluation process and the state of the data base are restored to the point at which the "symbolic" object was first used and the process is resumed using the discovered identity.

With these rules for data base access during Meta-Evaluation and the update of the data base caused by ASSERT and DELETE statements, the remainder of the Meta-Evaluation process pertains to individual types of program statements:

A. Subroutine call. The actual parameters are substituted for the formals and the subroutine is simulated. If it is a routine in the informal specification then the Meta-Evaluation process is recursively applied to it. Otherwise, the routine is simulated by assuming all of its pre-conditions and by asserting its post-conditions. *Pre-* and post-conditions provide a way of summarizing the requirements and results of a routine without actually executing it (and must be provided for the library routines which the program invokes so that they can be simulated during Meta-Evaluation).

B. Sequence of statements. Each statement in the sequence is Meta-Evaluated in turn.

C. Loops. If the loop predicate matches existing relations in the Meta-Evaluation data base, then the loop body is Meta-Evaluated for each such match with the placemarkers bound to the matched named objects. If no match exists, then symbolic data is created so that a single match of the loop predicate will succeed, and then the loop body is Meta-Evaluated for the (newly created) matched pattern. Thus, whether or not the pattern is initially matched (and normally it won't be, so that a single new symbolic relation satisfying the pattern will be created), the loop body will be executed for each known relation satisfying the loop predicate. Thus, even though we have no way of representing universal quantification, such quantification has been operationally applied to the data base so that the resulting state is consistent with universal quantification.

D. Conditional statement. The predicate of the IF statement is assumed to be false (i.e., is deleted from the data base) and the ELSE clause is Meta-Evaluated. Then the data base is restored to its state before Meta-Evaluating the IF statement, the predicate is assumed to be true (i.e., is asserted in the data base), and the THEN clause is

Meta-Evaluated.. Our present implementation is incapable of simultaneously representing the effects of the THEN and ELSE clauses as separate alternatives, and one branch—the THEN clause—is chosen as the one whose effects will be reflected in the data base for Meta-Evaluation of succeeding statements. This choice is based on the fact that the THEN clause is usually more fully developed than the ELSE clause and because it is normally the expected case—the normal path through the program.

THEORY OF INFORMALITY RESOLUTION

The previous section described how a program's behavior could be simulated statement by statement on symbolic data. The purpose of this simulation is to provide the context for resolving informalities in the program. This resolution is composed of two parts: (1) The hypothesizing of one particular interpretation for the informality from a set of possible interpretations and (2) the testing of hypotheses.

There are many types of informalities which can occur in a program specification (See [13]). These informalities correspond in one way or another to the suppression of explicit information. Each informality is expressed by use of a partial construct in place of some intended complete construct. For each partial construct we have algorithms which generate an ordered set of possible completions. The alternatives are tested by the well-formedness criteria explained in the next section. The generation algorithms represents our theory of informality resolution.

Although there are many types of informality handled by the SAFE system, we will consider only those which are resolved during the Meta-Evaluation process.

These informalities arise because in natural communication the first usage of an object is not labeled and then reused for later references to that object. Instead, references tend to include as little detail as required to reference objects from the current context. This might simply be a pronoun ("it" or "one"), a type name ("the message"), a partial description ("the red one"), or a completely omitted reference when the desired object is already part of the context. Otherwise, either a full reference sufficient to unambiguously select the desired object from the data base, or simply a type name if the desired object is associated with *an* object already in context, must be used. Any references in a description may themselves be incomplete. All these ambiguities are resolved in the context established by the running program rather than the context of the input description. This context is the set of objects already bound and accessible in the program block. This includes the parameters of the program, embedding iteration placemarkers and placemarkers bound in preceding statements.

Descriptive references are resolved by pattern matching them with the simulated run-time data base. If the pattern match succeeds then the reference placemarker is bound to the matched object which must either be a literal in an asserted relation previously produced by the program or a previously created symbolic object (because those are the only categories of objects which exist in the simulated data base). If a literal was matched, then the placemarker is replaced in the program by that literal. Otherwise (a previously created symbolic object was matched), the placemarker is replaced in the program by the placemarker previously bound to the symbolic object thus equating the two references in different parts of the program. If the pattern match for the descriptive reference fails, then new symbolic objects are created so that the match will succeed and the reference placemarker is bound to the appropriate symbolic object and is left unaltered in the program. It is treated as a separate placemarker which must be bound to an actual named object at run-time rather than as a reference to other placemarkers or literals in the program.

Pronouns are replaced by a reference of the type required for that argument. For both these typed references and those which explicitly occur in the input, (e.g. "the message") an ordered set of possibilities is constructed. These possibilities are all drawn from the current context by their degree of closeness to the typed reference according to the following categories relating the type (X) of the reference to the type (Y) of a placemarker in the context: X equals Y, X is a subtype of Y, X is a part of Y, Y is a part of X, X is connected via a path of single valued relations to Y, and X is a supertype of Y. Within a category the placemarkers are ordered by their use in the program as: scope placemarkers (placemarkers bound in an IF statement predicate or a loop predicate), *parame\ers, and* the remaining previously bound placemarkers.

Completely omitted references are treated exactly like the pronoun case except that literal instances of the required type *are* added as possibilities before any supertype ones. Furthermore, if a literal instance is selected as the accepted binding, and all other literal instances are also acceptable, then the omitted reference is treated as a don't-care situation.

One remaining kind of informal reference remains--a reference of inappropriate type. Either a descriptive reference or explicit type reference was specified but its type was not compatible with the type required by the action or relation in which the reference occurred. This difficulty is resolved by creating a new placemarker of the required type and determining an ordered set of possible conversions from the specified type (X) to the required type (Y) from the following list: X is a subtype of Y, X is a part of Y, Y is a part of X, X is connected via a path of single valued relations to Y, Y is a subtype of X.

Thus, for each kind of informality, an explicit ordered set of possible interpretations has been created. These possibilities are explored by a simple backtracking search process integrated with the Meta-Evaluation of the program so that whenever an informal construct is encountered during Meta-Evaluation the first possible interpretation is selected and Meta-Evaluation continues until the *program* has been completely Meta-Evaluated or the program is found to be ill-formed (as described in the next section). In the latter case, the Meta-Evaluation process and the state of the simulated program is restored to its state at the point of the most recent informality interpretation selection for which remaining, untried possibilities exist. The next untried possible interpretation for that informal construct is selected and the Meta-Evaluation process resumed.

This process will terminate either by finding a set of interpretations which, within the documentation capabilities of the system, yields a well-formed formal program, or by determining that the informal specification was unintelligible because no well-formed program could be discovered for it.

PROGRAM WELL-FORMEDNESS RULES

In this section we describe some of the rules which provide the basis for rejecting the current selected set of interpretations as producing an ill-formed program. Programs *are* highly constrained objects (one reason they are hard to construct) and these constraints provide the means of rejecting interpretations of informality which don't make sense.

These rules are divided into two categories: (1) general ones which *are* resolved by backtracking through the current set of selected interpretations and (2) specific ones for which particular fixes to the program *are* known. The general ones pertain to incorrect interpretations of informalities which explicitly appear in the program and for which a set of alternative interpretations has been generated as explained in the previous section. The specific ones, on the other hand, pertain to implicit informalities in the program, which until the specific well-formedness rule was violated, were not known to exist and for which unknowingly one particular interpretation was chosen without considering the other alternatives. The chosen alternative caused the specific well formedness rule to be violated and, hence, the other alternatives must now be tried.

General Rules--resolved by backtracking through the explicit informalities:

1. An error cannot occur during Meta-Evaluation~-in our program model errors can only occur by violating constraints on the data base. These constraints are particular to a domain and are discovered during the domain acquisition process. They may involve only a single relation (such as requiring it to be single valued) or combinations of relations (such as, "the boss of a person must work for the same company as that person").

2. The predicate of conditional statements must not be determined during Meta-Evaluation—if it is, then the predicate is independent of the input data and the same branch of the conditional will always be executed. Thus, the program is ill-formed.

3. Each demon and procedure specified must be invoked somewhere -if not, why bother to describe it.

A. At least one placcmarkcr in the loop predicate mu^t be referenced within the loop body--otherwr.e, the loop' body is independent of the loop predicate (we are explicitly ruling out "counting loops" which simply determine the number of objects which satisfy some criteria).

5. An action should not be invoked which only produces redundant results (i.e., doesn't chance the data base)—the invocation produced no effect. Lither it should not be invoked or invoked with different arguments or some previous action should not have been invoked or invoked with different arguments.

6. All produced relations in the data base mu.t be consumed (read-accessed) either by the program or as part of the output—otherwise, its existence in the data base has no effect.

7. All expectations must be fulfilled. Informal specifications normally include descriptions of why certain actions *are* being performed to help create a context for people to understand the process being cJescribed. Such statements create an expectation about how the process will behave and can be used as a constraint on the process' behavior.

Specific Rules—uncovers an implicit informality and specifies how to resolve it:

1. Each typed reference must have a non-empty set of possible interpretations—if not, then the reference cannot be resolved within the current context. Solution: Assume (and verify) that it can be resolved by the caller of the current routine. Make it a parameter of the current routine and add it as an omitted reference to all calls of this routine.

2. Parameters must be directly referenced within a routine—if they *are* only indirectly referenced, then those components of the parameter directly referenced should replace the unreferenced object as parameters of the routine.

3. Statements outside a conditional cannot unconditionally consume results produced in one branch of that conditional—either make the consuming statement part of the producing branch, or condition its execution with the predicate of the conditional. This corresponds to informality in natural language that the end of conditional statement is normally not explicitly signaled.

4. Non-produced goal (this is a specialization of the general expectation rule)--if a statement is invoked and is expected to produce some result but only produces a portion of the goal and the goal does not contain any unbound placemarkers outside of the portion produced, then assert the goal using the produced portion. This corresponds to the informality that a "passive" construct specifying the desired effect of some action actually indicates that the desired effect should be created from the results of that action.

CONCLUSION

The techniques described in this paper are only the beginning of a technology for understanding informal program specifications based on theories of informality resolution and program well-formedness acting in the context established by Meta-Evaluation of the program. Each of these areas requires further development and we have only started to experiment with their interactions and, yet, this prototype system has successfully transformed a few small (approximately one page) informal program specifications into their formal operational equivalents. These examples have been (carefully) extracted from actual functional specification manuals and the prototype system accommodated to the needs of the example by developing one or more of these areas. We expect that such example driven growth of the system will continue for some time until the theories and the Meta-Evaluation technology mature and become more complete. Unfortunately, we have been unable, so far, to represent the theories in other than a procedural manner so that growth and modification are ad-hoc and quite intertwined with the Meta-Evaluation process itself.

We do, however, believe that our approach is sound and the technology adequate. Composing a formal operational specification for a program is a difficult task and will remain so despite improvements in formal specification languages. The difficulty lies *m* the formalism itself. Thus, some *aid* must be provided in the composition process and we believe this can best be achieved by creating an interactive computer system which transforms an informal specification into the required formalism. This transformation can be accomplished by using the requirements of the formalism and a Knowledge of its operational characteristics to select the appropriate interpretation from the set of possible ones.

REFERENCES

Balzer, Robert, Neil Goldman and David Wile. Informality in program specification. *Fifth IJCAI Proceedings, 1977.* Also, USC Information Sciences Institute, ISI/RR-77-59, April 1977.

Goldman, Neil, Robert Balzer and David Wile. The use of a domain model in understanding informal process descriptions. *Fifth IJCAI Proceedings, 1977.*

Hayes, J. R. and Simon, H. Understanding written problem instructions. In Gregg (Ed.), *Knowledge and Cognition,* Lawerencc Erlbaum Associates, Potomac, Md., 1974.

4. Goldman, Neil, Robert Balzer and David Wile. The inference of domain structure from informal process descriptions. *Proceedings of Pattern Directed Inference Workshop in SIC ART Newsletter, *63, 1977.*

5. King, James C. "A new approach to program testing," *Proceedings of the International Conference on Reliable Software,* Los Angeles, April 1975.

6. Deutsch, L. P. *An interactive program verifier,* Ph.D. dissertation, University of California, Berkeley, May 1973.

7. Burstall, R. M. "Proving correctness as hand simulation with a little induction," *Proceedings of IFIPS 74,* North Holland Publishing Company, 1974.

8. Boyer, Robert S., Bernard Elspas and Kan N. Levitt. "Select-- a formal system for testing and debugging programs by symbolic execution, *"Proceedings of the Interational Conference on Reliable Software,* Los Angeles, April 1975.

9. Clarke, Lori A. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering,* September 1976.

10. Howden, William E. Experiments with a symbolic evaluation system. University of California at San Diego, La Jolla, California. *National Computer Conference, 1976.*

11. Yonczawa, Akinori. Symbolic-evaluation as an aid to program synthesis. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Working Paper 124, April 1976.

12. Beckman, Lennart, Anders Haraldson, Osten Oskarsson and Erik Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence, 1976,* pp. 319-357.

13. Balzer, Robert, Neil Goldman and David Wile. On the use of programming knowledge to understand informal process descriptions. *Proceedings of Pattern Directed Inference Workshop in SIC ART Newsletter,* #63, *1977.*