ARPA ORDER NO. 2223

12

ISI/RR-77-63
October 1977

ADA048153

Robert Balzer
Neil Goldman
David Wile

On the Use of Programming Knowledge to Understand Informal Process Description

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA ISI

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

14 1. REPORT NUMBER
ISI/RR-77-63 ✓

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

6 4. TITLE *(and Subtitle)*
On the Use of Programming Knowledge to Understand Informal Process Description.

5. TYPE OF REPORT & PERIOD COVERED
9 Research rept.

6. PERFORMING ORG. REPORT NUMBER

10 7. AUTHOR(s)
Robert Balzer, Neil Goldman, David Wile

8. CONTRACT OR GRANT NUMBER(s)
15 DAHC 15-72-C-0308

9. PERFORMING ORGANIZATION NAME AND ADDRESS
USC/Information Sciences Institute ✓
4676 Admiralty Way
Marina del Rey, CA 90291

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
ARPA Order 2223

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

11 12. REPORT DATE
Oct[redacted]77

13. NUMBER OF PAGES
12

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*
12 15p.

15. SECURITY CLASS. *(of this report)*
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

-----

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*
formal specification language, informal languages, informal software specification, meta-evaluation, natural language understanding, software specification, symbolic-execution

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The goal of improving and simplifying communication with computers has been pursued largely through the creation and use of better formal languages. This report investigates an alternative approach by exploring the variety and extent of informal constructs which can be introduced into a formal language without impairing communication. These informal constructs rep-
(continued)

next page

407 952

mt

20. (continued)

resent the suppression of certain explicit information which must be inferred from the surrounding context. In general, each informal construct has several possible interpretations, only one on which was intended by the speaker. The system's task is to use the existing context to focus attention on a small ordered subset of the most probable alternatives and to further reduce it by applying any constraints or well-formedness rules. The most probable remaining alternative is selected as the intended one. Program descriptions were chosen as the example task domain to test this approach because its rules of context and well-formedness are fairly well developed, and because we, as computer scientists, are our own domain experts.

ARPA ORDER NO. 2223

ISI/RR-77-63
October 1977

Robert Balzer
Neil Goldman
David Wile

On the Use of Programming Knowledge to Understand Informal Process Description

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
ISI
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO DAHC15 72 C 0308. ARPA ORDER NO 2223.

CONTENTS

ABSTRACT

The goal of improving and simplifying communication with computers has been pursued largely through the creation and use of better formal languages. This report investigates an alternative approach by exploring the variety and extent of informal constructs which can be introduced into a formal language without impairing communication. These informal constructs represent the suppression of certain explicit information which must be inferred from the surrounding context.

In general, each informal construct has several possible interpretations, only one of which was intended by the speaker. The system's task is to use the existing context to focus attention on a small ordered subset of the most probable alternatives and to further reduce it by applying any constraints or well-formedness rules. The most probable remaining alternative is selected as the intended one.

Program descriptions were chosen as the example task domain to test this approach because its rules of context and well-formedness are fairly well developed and because we, as computer scientists, are our own domain experts.

1. *INTRODUCTION*

We believe that current communication between people and computers suffers greatly from an artificial rigidity imposed by formal input languages. While it would be more convenient if natural language could be used, that goal has proved most elusive. Instead, we wish to investigate the use of and justify the need for informal languages for communication with computer systems.

By *informal language* we mean a language with a formal syntax and semantics that guarantees an unambiguous parse of any input sentence. The semantics are also constructed so that any input sentence composed entirely of complete constructs has a unique interpretation. The informality of the language arises from the use of partial constructs for complete ones; each informal language will have its own set of rules for the kinds of partial constructs allowed.

For each partial construct appearing in the input, the syntax of the language will define the set of possible completions of that construct; the system's task is to select the correct one.

Generally the selection is based on two factors: an ordering of the possibilities based on the context in which the partial construct appears, and a set of well-formedness rules based on the properties of the objects in the construct, the operations being performed on them, and the environment of these operations.

The power of an informal language hence depends upon the use of context and well-formedness rules to select among the possible completions of a partial construct. The stronger the rules for rejecting possibilities and the better the ordering in suggesting acceptable completions, the more information can be suppressed from the input and still remain understandable. This suppression is very important because it focuses attention on the important components of the input, makes it more concise and understandable, and provides an automatic mechanism for maintaining consistency as the environment changes. These features (as well as syntactic variability) are precisely the reasons why natural language is so comfortable to use.

As a simple example of an informal specification consider the following: "Search for an invoice for Bill. If one is found...." Understanding this specification requires knowledge beyond the text itself. Let's begin with "an invoice for Bill". How are Bill and invoice related? Bill is a customer and customers are identified by a number which appears as the customer number on an invoice. Thus, "an invoice for Bill" means an invoice where customer number is the number which identifies Bill. In the next sentence the word "one" must be understood. To do so, we must recognize that searching can result in finding something and that the thing found will be the thing searched for. Thus "one" in the second sentence refers to "an invoice for Bill".

This small example illustrates how information can be suppressed from a communication if it exists elsewhere and is available to the viewer of the communication to complete the partial constructs. The price we pay for these advantages is an increased processing by the receiver of the communication and possible misinterpretations. Some of the partial constructs can be completed straightforwardly, and their processing is so ingrained in us that we are normally unaware that we are completing a partial construct; others we find truly ambiguous or misinterpret because our context was different from that of the originator of the communication.

For suitable environments in which enough semantic support is offered by the ordering and well-formedness rules, we believe that a comparable level of comprehension can be provided by computer systems and that such a capability will be of enormous benefit to users in interacting with those systems and in formulating coherent, consistent bodies of information. A further benefit would be that the information could be precisely restated in a more complete form by the system.

2. *INFORMAL SOFTWARE SPECIFICATIONS*

The area of software specifications represents, we believe, an important field in which such an opportunity exists. Current specification languages display none of the *informal features* we described; in fact, informality is rigorously avoided. Because this makes such specifications hard to construct, understand, and modify, they are normally accompanied by a natural language description. Suppose we had a computer system which accepted an informal software specification, interacted with the specifier to clarify points of ambiguity, to point out inconsistencies, and to request additional information, and then *automatically* reformulated the input into a precise formal specification. Such a system would certainly help users construct, understand, and modify specifications by relieving them of the need to attend to myriad details and consistencies.

This is precisely the task we have undertaken in the Specification Acquisition From Experts (SAFE) project, the results of which are described in more detail elsewhere [1]. Here we wish to concentrate on the types of informality allowed and briefly indicate how they are resolved through the ordering criteria and well-formedness rules.

The basic assumption of the SAFE system is that, since a program is being specified, there are a set of objects to be manipulated by a series of actions. These objects are related to each other by relations, and the only primitive actions that can be performed are to create or destroy these relations between the objects or to create new objects. This model has greatly simplified the semantics of programs, provided a uniform method of dealing with all data, suppressed representation issues, enabled the program to model the task domain more directly, and reduced the translation required from input to running program. It is therefore a key factor in the success of the system.

3. FORMAL TARGET SPECIFICATION LANGUAGE

With this general model, the formal specification language into which all the informal constructs will eventually be translated by the system is quite simple. It is a programming language whose control statements consist of procedure-calls, if-statements, loops, and sequences of these statements. The predicate of an if-statement is a pattern which, if matched in the data-base of asserted relations, causes the then-clause to be executed; otherwise the else-clause is executed. The loop statement causes repeated execution of the loop-body for all instances of the loop-pattern matched in the data-base. Notice that there are no assignment statements; instead, whenever a pattern is matched in the data-base the variables of that pattern are bound to the objects of the relation matched.

These variables, which always reference an object in the data-base, are merely a shorthand for the description used to bind the variable. A pattern is simply a relation followed by its arguments. Each argument can be a literal, a variable, a description (the X such that <pattern>), or a function whose evaluation produces a literal. When the variables, descriptions, and functions in a pattern have been replaced by their literal values, the pattern becomes a tuple which can be retrieved from, added to, or removed from the data-base.

The language also supports demons so that event-driven processing can be specified, constraints so that checking can be automatically performed (thus suppressed from the program), and inference rules so that information can be automatically converted between equivalent forms as needed. These facilities are designed to provide the maximum flexibility in precisely describing the logical behavior of a program at the expense of drastically reducing the efficiency of the logical program. Since the specification is intended only to define the desired logical behavior, we feel that the suppression of optimization issues and the resulting simplification of the specification are decided advantages. In [2] we have addressed the issue of how such programs expressed in the formal specification language could be optimized.

4. TYPES OF INFORMALITY

There are three categories of informality based upon when and how the informality is resolved; they are obviously dependent on the basic approach adopted by the SAFE system--i.e., that the process of understanding an informal specification depends upon determining the structure of the domain in which the program will operate (what objects exist, how are they interrelated, what constraints must they satisfy), collecting unconnected fragments of processing to be applied to the objects of the domain, synthesizing these fragments into a coordinated outline (or plan) of processing, and supplying details to this outline to produce a well-formed program.

These activities have been implemented in three phases: the Linguistics Phase comprises both the structuring of the domain and fragment-collecting activities; it is followed by the Planning Phase, which builds the program outline, and the Meta-Evaluation Phase, which supplies details to the outline. This approach--extracting individual fragments from the input, assembling them into a plan, and detailing the plan--is the same as that used by Simon in the Understand [3] system. To simplify implementation, both the SAFE and Understand systems have chosen to omit any feedback path from a phase to a previous one, which means that an ambiguity in a phase must either be resolved correctly (via a well-formedness rule or by asking the user) or passed to the next phase.

We can now present the informalities allowed in SAFE category by category and discuss the information and processes used to resolve them. Examples of informal specifications which are processed by the three phases to produce a complete formal specification together with a description of the processing involved are given in [1].

5. *STRUCTURAL INFORMALITIES*

These informalities are handled by the Linguistic Phase, which uses both grammatical and dictionary linguistic information to resolve ambiguity, then queries the user if this knowledge is insufficient. No structural informalities are passed to other phases.

A. *Domain Acquisition* - We have found that a great deal of the structural domain-specific knowledge needed to understand a specification is implicitly contained in the specification itself and can be automatically extracted. Without such a capability the user would have to laboriously construct a formal structural model for the domain; with it, non-inferrable aspects of the domain can simply be included as part of the specification as is normally done in human communication. This informality represents a major effort within the SAFE project and is more fully covered in [4].

B. *Implicit Association* - Several constructs which indicate that one object is associated with another without specifying the association itself (such as "the X of Y", "Y's X, "X for Y", etc.) are allowed in the input. The system attempts to determine a unique path between objects of these types. If one can be found, it is used to resolve the ambiguity. If multiple paths are found, the user is asked which was intended;, if none are found, a single relation between the two types with an unknown name is assumed. Subsequent associations between these types are assumed to use this same relation; if it is identified explicitly, that name becomes the relation name.

C. *Passive/Active Recognition* - The use of the passive voice in the input and the fact that certain verbs can refer to either an action or a result of that action cause ambiguities as to whether the construct should be treated as an action to be performed, a pattern to be retrieved, or a statement of the way things are expected to be at some point. These possibilities are resolved either linguistically or by the user.

D. *Plurals* - The use of plurals in the input can indicate loops, specification of a "generic element," or groupings of objects into a set.

E. *Verb Definition* - Many natural language verbs have more than one meaning. When multiple meanings are found in the dictionary and they cannot be linguistically resolved, the user is asked to resolve the ambiguity. Repeated references to the same verb are assumed to refer to the same meaning.

6. *GROUPING INFORMALITIES*

The Planning Phase uses program structural knowledge (the semantics of the various control statements), program well-formedness criteria (e.g., instances of objects and relations must be produced before they can be consumed), some linguistic knowledge (e.g., that people tend to explain the normal case first and then provide refinements and/or exceptions), and an important assumption (that all explained actions must be invoked somewhere--otherwise, why bother to define them) to assemble the individual processing fragments into a program outline.

A. *Relative Sequencing* - Whenever an explicit statement of the relative sequencing of two or more actions is omitted, an attempt is made to determine their relative sequencing though a producer/consumer analysis. If one produces an object or relation consumed by another, then it must precede that action. This rule is the major determinant of the sequencing of the program outline. If two or more actions which are not explicitly sequenced do not consume either directly or indirectly the results of any of the others, then the relative sequencing of these actions is irrelevant and they are placed in a parallel execution block.

B. *Omitted Action* - If a consumed object or relation is not produced anywhere and there is a known action which produces it, then the action is added to the program outline so that it precedes the consumption. If more than one way of producing the result is known, they are all placed in an alternative execution block which precedes the consumption. This remaining ambiguity is passed to the next phase for resolution.

C. *Refinements* - Several processing fragments may refer to the same action rather than different actions which must be sequenced. Refinements are recognized by their reference to the common action and/or use of a refinement statement (e.g., "during X...") and are then merged into a separate program plan for the refined action.

7. *DETAIL INFORMALITIES*

The final category of informalities is handled by the Meta-Evaluation Phase. It deals with two types of informalities: explicit and implicit. The former are already-identified partial constructs in the program plan which must be completed (primarily object reference constructs which must be uniquely bound to a parameter of an action, an iteration variable of a loop, an object retrieved from the data base, a previously referenced object, or an object associated with one of these). In general, a large number of such possiblities exist for each reference ambiguity, and a program plan contains many of these.

It is clear that these ambiguities cannot be resolved in isolation from one another but are highly interdependent. These interdependencies are all related to how the program behaves dynamically and correspond to a set of well-formedness rules. The most effective way of testing that these rules are satisfied is to check them during the execution of the program. We therefore built a special program interpreter which checks these well-formedness rules as it executes a program. Since these rules must be satisfied for all executions of the program, it is executed on symbolic rather than actual data.

These well-formedness rules are based on the total dynamic state of the computation and, hence, are not associated with individual decisions of which possibility to select for an informality. Thus, when one of these rules is violated, the cause of the violation cannot be associated with a particular decision. Instead, the decisions are made one at a time as they are encountered during the Meta-Evaluation of the program; when a well-formedness rule is violated, the system backtracks through the decisions and resumes Meta-Evaluation of the program at the point of the revised decision.

This sequential decisionmaking and backtracking, together with a reasonable initial ordering of the possibilities, reduces the large space of possible bindings to a manageable size. In addition, these rules can uncover other hidden problems in the program for which specific remedies are known. The discovery of these problems and their resolution creates a set of additional implicit informalities.

A. *Incomplete Reference* - This explicit informality arises 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"), or a partial description ("the red one") 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 variables, and bindings established in preceding statements.

Descriptive references are resolved by pattern matching them with the run-time data base, which may bind them either to stored objects or to objects already in context. Pronouns are replaced by a type name reference of the type required for that argument. For both these typed references and those which explicitly occur in the input, 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 an object 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 to Y, and X is a supertype of Y. Within a category the objects are ordered by their use in the program as: scope objects (iteration variables and objects bound in an if-statement predicate), parameters, and other (the remaining objects in the context). These possibilities are examined as necessary during the backtrack Meta-Evaluation of the program.

B. *Omitted Operand* - This explicit informality is an extreme case of an incomplete reference and is 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 accepted, then the omitted operand is treated as a don't-care situation.

C. *Alternative Block* - This explicit informality indicates that one of the alternatives was intended. This construct is created by the Planning Phase for omitted action informalities that it is unable to resolve. The possibilities are examined, as necessary, during the backtrack Meta-Evaluation of the program.

D. *Scope of Conditionals* - This implicit informality arises because the end of the THEN or ELSE clause in a conditional is almost never explicitly signaled. Statements following a conditional are presumed by the Planning Phase to be outside the scope of the conditional. If such a statement unconditionally uses the results of a statement already in the conditional, then it is included in that branch of the conditional.

E. *Scope of Demons* - This implicit informality arises because the pattern specified to activate a demon may well be too general and cause it to be activated inappropriately. If the repeated activation of a demon causes an error, rather than backtrack, the user is first asked whether the demon should be activated in the current situation.

F. *Discovered Parameters* - Often actions are described without explicitly specifying their parameters. The description uses these implicit parameters as if they were already part of the established context. When an explicit typed reference occurs for which no antecedent can be found by the binding mechanism, an attempt is made to interpret the antecedent as being an implicit parameter of that type. The parameter is added to the definition of the routine, and all calls to the routine are updated to include a missing operand of that type.

G. *Implicit Type Conversion* - When a typed or descriptive reference of inappropriate type occurs as an operand, then it is processed normally to determine its actual reference. In addition, it is replaced by a typed reference of the required type which is bound to an object (X), associated with the original one (Y) by one of the following relations: X is a subtype of Y, X is a part of Y, Y is a part of X, X is connected via a path to Y, or X is a supertype of Y. Recognition of the need for type conversion may come either via a static analysis of the program or during execution and may depend upon the existing set of binding decisions.

8. CONCLUSION

As can be seen from the preceding sections, the SAFE system is prepared to accept a wide range of informalities. Some of these informalities are explicitly recognized in the input and replaced by an ordered set of possibilities. Others are only recognized by failure of a well-formedness rule during Meta-Evaluation. The explicit informalities are resolved via a backtracking search through the sets of possibilities while the implicit ones are resolved via special solutions associated with the particular well-formedness rule which discovered the implicit informality.

Whether or not a particular informal construct can be correctly resolved is critically dependent on the context in which it appears. The more highly developed this context is the more likely the correct completion will be selected by the system.

Speakers naturally supply additional context and/or limit their use of informal constructs in complex situations so that the required informality resolution processing does not overly tax the receiver of the communication. That is, informal constructs are used when the speaker perceives that the constructs can be easily and successfully resolved by the receiver. Thus, once a minimum performance threshold has been reached by a computer system resolving informalities, we should expect quite broad system applicability because of the self-regulating use of informalities which reduces the range of difficulty in resolving such informalities.

The current performance of the SAFE system on a few example program descriptions, though far from the minimum required threshold, indicates the basic feasibility of our approach and provides a step toward its realization. Progress will be made as we devise better rules defining program context and well-formedness, and make better use of such rules in rejecting possible completions of informal constructs.

REFERENCES

1. Balzer, R., N. Goldman and D. Wile, "Informality in program specification", *Proceedings of Fifth International Joint Conference on Artificial Intelligence,* August, 1977, and USC/Information Sciences Institute, ISI/RR-77-59.

2. Balzer, R., N. Goldman and D. Wile, "On the transformational implementation approach to programming," *2nd International Conference on Software Engineering,* October 1976, IEEE Catalog No. 76CH1125-4C, pp. 337-349.

3. Hayes, J. R. and H. Simon, "Understanding written problem instructions," In Gregg, *Knowledge and Cognition,* Lawrence Erlbaum Associates, Potomac, Md., 1974.

4. Goldman, N., R. Balzer and D. Wile "The inference of domain structure from informal process descriptions," *Proceedings of Workshop on Pattern Directed Inference Systems,* Hawaii, May 1977.

Robert Balzer   Neil Goldman   David Wile

USC / INFORMATION SCIENCES INSTITUTE
4676 Admiralty Way   Marina del Rey   California 90291

On the Use of Programming Knowledge to Understand Informal Process Descriptions

ISI/RR-77-63