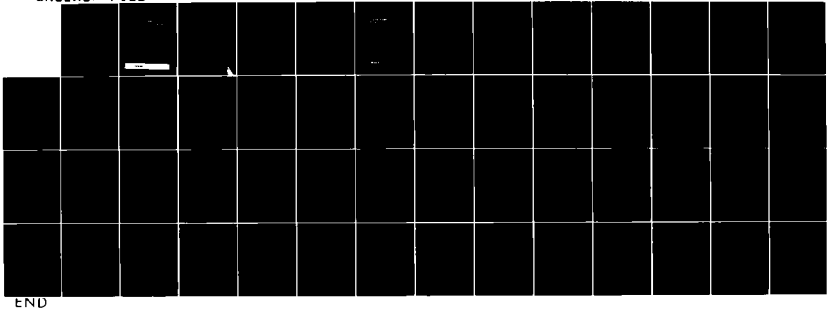
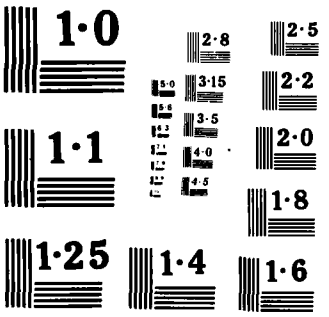
AD-A159 821 THE RAND-ABEL PROGRAMMING LANGUAGE: HISTORY RATIONALE 1/1
AND DESIGN(U) RAND CORP SANTA MONICA CA
N SHAPIRO ET AL. AUG 85 RAND/R-3274-NA MDA903-85-C-0030
UNCLASSIFIED F/G 9/2 NL



END
DATE FILMED
12-85
DTIC

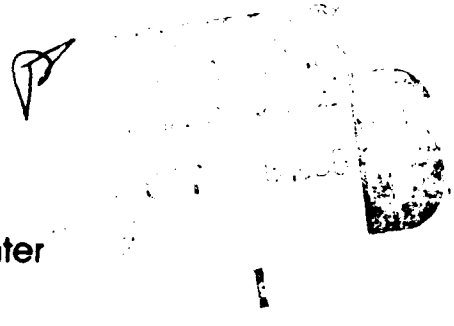

1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
5·0
1·4
2·5
2·2
2·0
1·8
1·6

(12)

The RAND-ABEL™ Programming Language

History, Rationale, and Design

Norman Z. Shapiro, H. Edward Hall,
Robert H. Anderson, Mark LaCasse



A Report from
The Rand Strategy Assessment Center

Rand

The research described in this report was sponsored by the Director of Net Assessment, Office of the Secretary of Defense under Contract No. MDA903-85-C-0030.

Library of Congress Cataloging in Publication Data

Main entry under title:

The RAND-ABEL programming language.

"Prepared for the Director of Net Assessment, Office of the Secretary of Defense."
"August 1985."
Bibliography: p.
"R-3274-NA."
1. ABEL (Computer program language) I. Shapiro, Norman Zalmon, 1932- . II. Rand Corporation. III. United States. Dept. of Defense. Director of Net Assessment.
QA76.73.A14R36 1985 005.13'3 85-19187
ISBN 0-8330-0674-6

The Rand Publication Series: The Report is the principal publication documenting and transmitting Rand's major research findings and final research results. The Rand Note reports other outputs of sponsored research for general distribution. Publications of The Rand Corporation do not necessarily reflect the opinions or policies of the sponsors of Rand research.

Published by The Rand Corporation

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER R-3274-NA	2. GOVT ACCESSION NO. AD-A159821	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The RAND-ABEL Programming Language History, Rationale, and Design		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Norman Shapiro, H. Edward Hall, Robert Anderson, Mark LaCasse		8. CONTRACT OR GRANT NUMBER(s) MDA903-85-C-0030
9. PERFORMING ORGANIZATION NAME AND ADDRESS The Rand Corporation 1700 Main Street Santa Monica, CA 90406		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Director of Net Assessment Office of the Secretary of Defense Washington, DC 20301		12. REPORT DATE August 1985 13. NUMBER OF PAGES 43
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

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

No restrictions

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming Languages
War Games

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

see reverse side

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

This report describes the motivations behind the development of the RAND-ABEL programming language and some of its novel features. RAND-ABEL was designed to meet the needs of the Rand Strategy Assessment Center, which is building a large system for automated war gaming in which separate rule-based models represent U.S., Soviet, and third-country behavior. To satisfy the requirements for speed and transparency, the language was designed to be: (1) rapidly compilable and executable; (2) self-documenting; (3) understandable by nonprogrammer domain experts after modest instruction; (4) reasonably easy to learn and use, especially for modifying or incrementally extending existing code; (5) portable across different computers; and (6) well suited to development of large and complex rule-based simulations. Certain of its features are unique: the ability to express directly in RAND-ABEL source code such natural structures as decision tables (isomorphic with decision trees) and order tables, which lay out orders to be executed sequentially, and its novel declaration-by-example feature, which is useful for rule-based programs with enumerated variables and many distinct data types. RAND-ABEL has built-in support for a data dictionary for communication between separate modules.

R-3274-NA

The RAND-ABEL™ Programming Language

History, Rationale, and Design

Norman Z. Shapiro, H. Edward Hall,
Robert H. Anderson, Mark LaCasse

August 1985

Prepared for the
Director of Net Assessment
Office of the Secretary of Defense

A Report from
The Rand Strategy Assessment Center

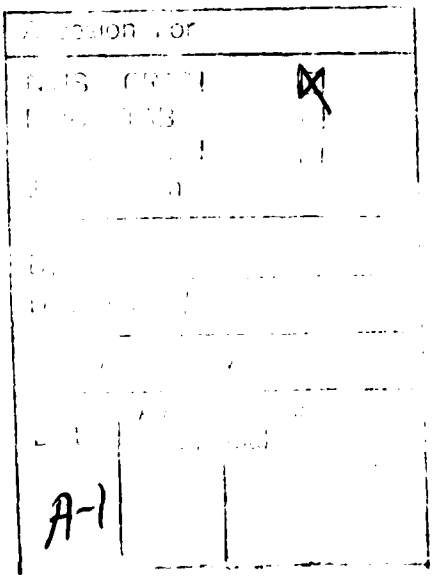
Rand
1700 MAIN STREET
P.O. BOX 2138
SANTA MONICA, CA 90406-2138

PREFACE

This report describes the motivations behind the development of the RAND-ABEL™ programming language and some of its novel features. The RAND-ABEL language was developed at The Rand Corporation for use in writing complex rule-based models as part of a system for automated war gaming. The language was designed and implemented for use by the Rand Strategy Assessment Center (RSAC), which is supported by the Director of Net Assessment, Office of the Secretary of Defense.

RAND-ABEL is an evolving operational language. This report describes some of RAND-ABEL's features as of July 1985. It is intended for military analysts, modelers, and software engineers interested in new programming languages suitable for large rule-based simulations, including those using expert system techniques. It also serves as an introduction to the principles underlying the design of RAND-ABEL for programmers using the language. A complete description of the RAND-ABEL language is contained in the authors' forthcoming Rand Note, *The RAND-ABEL Programming Language: Reference Manual.*

Inquiries and comments on this report are welcome. They may be made directly to the authors or to Paul K. Davis, director of the Rand Strategy Assessment Center.



SUMMARY

This report is one of a series describing a new programming language called RAND-ABEL. The principal objective here is to explain in some detail the rationale underlying RAND-ABEL's design and to describe certain RAND-ABEL features that are unique or unusual. A secondary objective is to describe RAND-ABEL's development history, which may be of interest to computer scientists because there are so few discussions in the literature of how such developments actually proceed.

RAND-ABEL was designed to meet the needs of the Rand Strategy Assessment Center (RSAC), which is building a large system for automated war gaming in which separate rule-based models represent U.S., Soviet, and third-country behavior. The decisions from those models then become inputs to a simulation of large-scale crisis and conflict. Primarily because of requirements for speed and transparency, it was necessary to design a new language. Upon reflection, we determined that the language should be: (1) rapidly compilable and executable; (2) self-documenting; (3) understandable by nonprogrammer domain experts after modest instruction; (4) reasonably easy to learn and use—especially for modifying or incrementally extending existing code; (5) portable across different computers; and (6) well suited to development of large and complex rule-based simulations. Later, we concluded that it is highly desirable to add interpretive features and incremental compiling, especially if RAND-ABEL programs were to be written by analysts with only modest programming skills.

To a substantial degree, RAND-ABEL's design derived logically from these requirements coupled with certain beliefs we hold concerning languages for RSAC-like applications. Throughout RAND-ABEL's development we have emphasized: (1) language features thought natural to the intended users, with particular emphasis on two-dimensional structures, such as decision tables, that are cognitively efficient and logically transparent; (2) readability (even at the expense of efficient initial coding); (3) strong type checking; (4) syntactic conventions minimizing the scan-ahead needed to resolve ambiguity; and (5) orthogonality in the sense of not using an operator such as "+" to represent two different concepts.

Certain features resulting from this approach are unique and, we believe, a contribution to the state of the art. In particular, the ability to express directly in RAND-ABEL source code such natural structures

as *decision tables* (isomorphic with decision trees) and *order tables*, which lay out orders to be executed sequentially, have proved in practice to be enormously valuable—to both writers and reviewers of code. RAND-ABEL also uses a novel *declaration by example* feature, which is especially useful for rule-based programs with enumerated variables and many distinct data types that otherwise would require separate names. Associated with RAND-ABEL is built-in support for a data dictionary for communication between separate modules.

RAND-ABEL is currently implemented as a preprocessor for the C programming language under the UNIX operating system, which makes it quite portable across different computers. In its present environment at Rand, RAND-ABEL is used with a data dictionary, a data editor, and a special capability permitting co-routines. This has been essential for representing naturally in computer code the various objects of a hierarchically structured system (i.e., various military and national command levels). It has also been essential in permitting RAND-ABEL code to accommodate scripts (time sequences of commands or actions to be performed), which are important in many expert systems.

Although it is too early for a definitive assessment, RAND-ABEL appears to be successful in achieving most of its goals. In particular, it can execute a rule-based program of approximately 4000 lines in about a second or less, corresponding to approximately 1 millisecond per rule, on an unloaded VAX 11/780. It produces succinct readable programs, and has been used successfully by a multi-team development project. Its principal shortcoming so far has been that it has proved more difficult to learn and write than we had hoped. We are quite confident that this shortcoming will be greatly reduced with the imminent addition of interpretive features and incremental compiling of the data dictionary. Even then, writing complex programs will be more difficult than we had expected, but the difficulties we observe are increasingly related to fundamentals such as the inherent complexity of the phenomena we are simulating. Other improvements planned for RAND-ABEL in the very near term (during summer 1985) include: sets, structures (i.e., records), pointers, transcendental functions, more succinct declarations, and many more applications of the two-dimensional Table statement.

Based on our experience, we expect that RAND-ABEL will be of value for other applications requiring a highly readable language, fast performance, and early discovery of errors. It is especially suitable, we believe, for complex models in which these requirements are especially stringent.

ACKNOWLEDGMENTS

The design and implementation of RAND-ABEL to date have benefited from comments, usage, and suggestions by Paul Davis, Herbert Shukiar, Steven Bankes, William Jones, Jean LaCasse, Arthur Bullock, David Shlapak, and William Schwabe. Bill Jones "walked on grass" (see the subsection "Build the Sidewalks Where the People Walk"); his use of tables of information in particular inspired our development of the Table statement. Jean LaCasse's study of ROSIE™ use by the Rand Strategy Assessment Center (RSAC) was particularly helpful, and she wrote the first RAND-ABEL program with content.

We especially wish to acknowledge strong and continuing support for the development of RAND-ABEL by Paul Davis and Herb Shukiar of RSAC. Paul's stubborn refusal to be satisfied with anything less than our best effort at meeting RSAC's real needs, Herb's effective and knowledgeable guidance, and their allocation of resources to RAND-ABEL when its future was uncharted are responsible for RAND-ABEL's success to date.

CONTENTS

I. INTRODUCTION

This report discusses RAND-ABEL,[1] a new programming language developed during the past 18 months at The Rand Corporation. One question must be asked and answered immediately upon making that statement: Why does Rand, or the world for that matter, need another programming language? In essence, this whole report is a response to that question; it presents the background leading up to the decision to develop RAND-ABEL, the requirements that could not be met by an existing language, and the novel features of RAND-ABEL that resulted in meeting these requirements.

For now, the short answer to "Why RAND-ABEL?" is: A team of people developing a complex rule-based[2] simulation system, who were experts in the subject matter of the simulation but not in programming, needed a highly readable, fast-executing simulation development system that was portable across computing machines. Existing languages were not adequate. Also, as will become clear below, the time and cost for developing a new language were relatively low, because we stood on the shoulders of a giant: the C language and its supporting UNIX[3] system.

Beyond answering the "Why RAND-ABEL?" question, we believe it is important to present a case study of the development of a new programming language, to show the real-world considerations involved, the compromises inevitably made, and the extensions and insights that occur in mid-process that enrich the final result. These considerations do not traditionally appear in a language reference manual (and, following tradition, do not appear in the RAND-ABEL Reference Manual [Shapiro et al., 1985a], but affect the design and applicability of a language and should therefore be understood by its users.

This report also discusses the novel features of the RAND-ABEL language, which we feel are a contribution to programming language design. These features include: use of two-dimensional tables of information as a programming language construct; declaration of identifiers by giving examples of their usage; translation of RAND-ABEL code

[1] The name RAND-ABEL stands for nothing in particular.

[2] The term "rule-based" is sometimes reserved for programming systems having a built-in inference mechanism interpreting a set of rules having a standardized form. In this report, we use the term more generally to mean the specification of logic that is primarily in an "If . . . then" form or else is described by decision tables.

[3] UNIX is a trademark of AT&T Bell Laboratories.

Last, as the prototype version of RAND ABEL became used for the construction of agents, analysts themselves suggested new facilities, or sometimes new facilities suggested themselves through the patterns we observed in their usage. Since RAND ABEL was meant from the start as a language for strategic analysts, analysts were taken as an important touchstone in all design considerations. The point is well made by a story related during design discussions:

> One day, Robert Hutchins, then President of the University of Chicago, was talking with the campus Director of Buildings and Grounds. The Director handed Hutchins an aerial photograph of the campus showing where the students had worn brown paths through the beautiful lawns by taking shortcuts among the sidewalks. Hutchins exclaimed, "Then, that's where you should build the sidewalks!"

Perhaps the most important "sidewalk" in the RAND ABEL language is the concept of a statement that directly incorporates a two-dimensional table of information. A senior Rand analyst, Bill Jones, was observed early in RAND-ABEL's usage to regularly rely on tables of information as a concise yet complete description of the options to be considered in a particular situation, or the set of orders to be processed in some consecutive manner. He then spent time translating this information into more verbose and less obvious rules or statements in RAND-ABEL. Members of the design team decided that these tables were a natural construct that should be directly incorporated, although their two-dimensional structure is complex and cannot be handled by normal left-to-right parsing algorithms.

In our first attempt, tabular information was handled quite differently from the current mechanism: a sequence of one-line "tabular function calls" consisting of a function name followed by a space-separated list of parameters. This allowed function calls to be organized into a tabular form if desired, but did not make tabular notation a part of the language syntax. We then observed that the most common use of this feature was to create a block of calls, all using the same function name, but with varying parameter values. In our continuing attempt to make these "proto-tables" as terse as possible, we then replaced the multiple occurrences of the function name within the table body with a single occurrence in the table header.

We originally used "Begin Table" and "End Table" brackets. Within the table's header, the order in which parameter names were listed as column headings determined the matching of columns to function parameters. (That is, the first—leftmost—column was assumed to contain values for the first-mentioned parameter, etc.) The next step in the table statement's evolution was the Macro Table, in which a

even by these early steps. (Currently, compiled RAND-ABEL rule based programs can execute approximately 4000 lines in about a second or less, or about a millisecond per rule, on an unloaded VAX 11/780.)

During this time, to get some capability quickly, the C language itself was used to declare data items and perform other housekeeping chores. Modeling logic was written in the higher-level pseudo-ROSIE language, linked with C programs. The translator extracted these higher-level statements and replaced them with C language equivalents. Within one or two months from the decision to create an intermediate pseudo-ROSIE language, the logic of the Scenario Agent was successfully translated into an intermediate language that could be automatically translated into C. From this point on, extensions and modifications to the Scenario Agent could be performed directly in the intermediate language. Pseudo-ROSIE took on a life of its own, freed from its ROSIE ancestor. This new language was called ABEL. (The prefix RAND- was later added to distinguish the language from other systems and products with similar names.)

RAND-ABEL AS A LANGUAGE

The evolution of the RAND-ABEL language, once RAND-ABEL was perceived as a separate entity, was driven by four different themes, although not necessarily all at once, and with varying priorities. First, design discussions began shifting from "How do we handle ROSIE constructions?" to "What should a modeling language really contain?" Second, the initial version of RAND-ABEL relied on the C language to provide such facilities as iteration and declaration of local variables (since the initial goal of quickly creating a pseudo-ROSIE did not require or justify re-inventing some common programming constructs already available). This, however, required a programmer's intimate knowledge of the C language, violating the goals of readability and writability of RAND-ABEL code. So a second theme of RAND-ABEL language development was to make RAND-ABEL a self-contained, consistent, and high-level language that could be understood as a single conceptual entity.

Third, the expanding concept of a data dictionary linking all modules together, and the need for co-routines, required new supporting facilities in the RAND-ABEL language.

a load of between 10 and 20 users. Regrettably, we never bothered either to run *Decide Policy* 100 times for a more precise wall clock timing, or to separate its execution from its interactive aspects so that the UNIX *time* command could be run on it.

- Even if ROSIE-in-C was developed successfully, on time, the resulting speed increase might still not be adequate for RSAC needs.
- We felt that the requirements of the RSAC might best be served by some new language features; experimentation with new language facilities might well be easier while continuing to pursue a specially designed language, not trying to retrofit these features into the existing language ROSIE.
- Finally, the RSAC team dealing with these language issues had "a head of steam up" toward developing a language tailored for RSAC. (At this time, using *yacc* and *lex*, the design team already had a kernel of the new language running as a test.) Giving a go-ahead for a new language development would maintain the interest and morale of the RSAC system design team.

After reviewing the situation, the RSAC program director concluded that we should go ahead on language development because the resulting system had the potential to be an order of magnitude faster even than ROSIE-C, which could be critical for the RSAC's unusual objectives (gaming and simulation); also, we were far enough along by that time to be confident of being able to develop an operational language reasonably soon, while considerable uncertainties still existed regarding the difficulties in reprogramming the ROSIE language into C. Finally, we had begun to discuss a number of new language features that seemed especially attractive, features that were not likely to exist in ROSIE for some time (e.g., tables).

Development of the new language continued, with continued reliance on the UNIX tools *yacc* and *lex*. By using these general-purpose tools, we were able to achieve both speed and generality. (Generality was important because we had some new features in mind for the Mark III version of RSAC, and wanted to create a system into which they could be gracefully introduced.)

Results of initial tests of the new language were dramatic: Although precise timings were not done, it was clear that the resulting compiled C code executed thousands of times faster than the equivalent ROSIE code. Run time was similarly dramatically improved. The new code ran on a time-shared VAX 11/780, which we estimate to be about 10 times slower than the dedicated DEC-20 formerly used. In spite of this machine disadvantage, the new Scenario Agent ran on the VAX in under a tenth of a second.[2] At least the efficiency goal was being met,

[2]On a DEC-2060 with a single user, the Scenario Agent's main function, *Decide Policy*, took ten minutes of clock time. The same function, recoded into the first version of RAND-ABEL, took less than 0.5 seconds of clock time on a VAX 11/780 with

IV. THE EVOLUTION OF RAND-ABEL

PSEUDO-ROSIE

The Scenario Agent within the Mark II version of RSAC, which described the actions of other parts of the world except for Blue (United States) and Red (USSR), was written in the ROSIE language. ROSIE is a rich, complex language whose parsing is necessarily complex. But recall that only a subset of ROSIE was in practice being used. Therefore, a tactic suggested itself:

1. Hand-translate existing ROSIE code for the Scenario Agent into a much simpler "pseudo-ROSIE"[1]
2. Write an automatic translator that could turn pseudo-ROSIE statements into valid C language statements

The hand translation step did not translate ROSIE directly into C because of the goal of having statements readable by analysts. This two-step strategy was thus adopted. A first version of a pseudo-ROSIE to C translator was written, relying on the UNIX *yacc* program for syntactic analysis and on a lexical analysis program written earlier.

Some initial experiments were run using this translator and the results were very encouraging. At this point, a major management decision was required: whether to (1) risk the RSAC on the development of a new language, (2) continue using ROSIE (which existed, and was much liked by users), or (3) await a scheduled reprogramming of ROSIE in the C language and thereby ameliorate the speed problem. This decision was complicated by several factors, among them:

- It was not clear how long the reprogramming of ROSIE would take; the problem was hard and the opportunities for delays legion.
- The code for the Scenario Agent in ROSIE was not as readable and transparent to analysts as desired; it contained excessive prose, making it difficult to tell whether all cases were really covered, and what the essential decision tree lurking within the rules really was.

[1]The definition of a simple pseudo-ROSIE language gradually emerged from a series of meetings attended, in addition to most of the authors, by Jean LaCasse, Jim Gillogly, and (sporadically but importantly) Bill Jones.

standability of the translator, since they force a table-driven coding style that separates the unique features of the language being translated from the details of lexical analysis and parsing algorithms.

As a result of the above design considerations, a translator was to be developed that could take certain statements describing a model, considered to be understandable by policy analysts, and turn them into statements in the C programming language. This leaves open the question of what form those statements should take, and whether those statements should form a complete modeling language or else be an extension to the C language. In the next section, we describe how a language now called RAND-ABEL emerged from the process of developing such a translator.

executed in a computerized political-military war game between Red and Blue (the Soviet Union and the United States). In this case, each line requires that a simulated diplomatic message be sent to the country shown requesting that that country take sides, cooperate, participate in the European war, and participate in the Southwest Asian war as indicated. The order table is a part of a simulated Red war plan. The second line, for example, corresponds to Red sending Austria a message saying "Stay neutral and do not cooperate with Blue!"

```
Define RGCL3-intl-communication-action

   Table Red-to-3rd-countries

                                          european-      swa-
country        side      cooperation      involvement    involvement
=======        ======    ===========      ===========    ===========
Afghanistan    Red       Reinforcement    --             --
Austria        White     Uncooperative    Normal         Normal
Bahrain        White     Uncooperative    --             --

[...]

Poland         Red       Reinforcement    Combatant      --

[...]

Yugoslavia     Red       Reinforcement    Combatant      --

End.
```

The above table meets our criterion for being a self-documenting statement. It is also a valid statement in the RAND-ABEL language; Table statements are discussed in more detail in later sections.

To handle the needs for modularity and team development of complex systems, the system should have a *data dictionary* that defines all global data, so that communication between separately compiled modules always uses common terminology and assumptions. It should also allow co-routines,[2] permitting processes to become dormant and reawaken while retaining context.

The following additional design consideration is not really a goal, but rather a practical strategy: Use the UNIX tools *lex* (a lexical analyzer) and *yacc* (Yet Another Compiler-Compiler) to aid in developing this preprocessor for the C language. Use of these tools speeds up the development process, and aids portability because of the widespread availability of the UNIX system on a variety of computing machines. They also contribute to a certain cleanliness and under-

[2]Co-routines are discussed further in a later section of this report.

The C language does not meet the goals of readability and easy writability. Therefore, there should be some higher-level language tailored to creating complex simulations; processing of this new language should result in C code, which can then be compiled in the normal manner.

The goals of readability and writability should be met by a language that is "self-documenting." We use the term self-documenting to mean a language whose meaning can be understood by someone not intimately familiar with the details of the syntax of the language. We deliberately do *not* use the term "English-like" here, because that term has been misused enough to create complications—a situation that is exacerbated by ambiguity in the term: Do we mean using English words, syntax, and grammar, semantics, pragmatics, or all of the above? We mean none of the above, but rather: An English-speaking person that is familiar with the subject matter being modeled should be able to read the programming language in which the model is expressed and understand what is meant.

Another reason that "English-like" is inappropriate to describe a language like RAND-ABEL is that the *writer* of RAND-ABEL programs does not rely on his knowledge of English in writing those programs (unlike such programming languages as ROSIE). This is deliberate, so that there is no confusing assumption made by the writer of a RAND-ABEL program that some rules of English apply as valid RAND-ABEL syntax. It is interesting that we explicitly do not want to confuse writing of RAND-ABEL with English, yet we expect to rely on the reader's knowledge of English to help interpret RAND-ABEL statements—even to the extent that someone totally unfamiliar with the RAND-ABEL language, but familiar with the subject matter of a RAND-ABEL program, should be able to interpret that program. In fact, precisely this has occurred in RSAC demonstrations: military officers have stopped the demonstration, looked at the RAND-ABEL logic defining an agent, understood it, and recommended changes in the logic as an interesting exercise.

We note that a "self-documenting" language might differ in other important ways from an "English-like" language. The syntax of the English language (or any other natural language, for that matter) is not commonly considered to include many constructions that are understandable to every educated English speaker; examples are simple mathematical notation, tables, and charts. The following excerpt does not follow any English language syntax that we are aware of, yet should be immediately understandable to the reader (who, having gotten this far, qualifies as an English reader). This is an example of an "order table," each line of which corresponds to an instruction to be

or some equivalent. Yet an important goal was that, even for these complex programs, any interested person familiar with the subject matter of an agent should be able to read the (source) code for an agent and understand the assumptions and logic underlying its behavior. It would also be highly desirable for them to be able to make straightforward changes to the code, once the underlying structure for an agent had been developed.

- **Easy to write with little training**. Programmers or strategic analysts knowing one high-level programming language fairly well should be able to learn this new language easily, and be able to write agents after some study and practice (especially when using existing agents as a guideline and example) albeit with consultations with professional programmers.
- **Allow creation of complex simulations by groups of developers**. The language must be rich enough to allow complexity and subtlety in models, and to handle large models. The RSAC system consists of several agents acting together—ideally overlapping each other in processing. Therefore, it should be possible to create modular systems in which the various modules operate in a multiprogramming mode on a single CPU, or else execute in parallel on separate CPUs with some intercommunication between agents.
- **Solution required quickly**. If an existing language is to be modified or a new programming language developed, it should not require a major diversion of project resources. A quick, efficient solution is needed.

The RSAC system design team concluded that no existing programming language adequately met all these requirements. Therefore, the design team now concentrated its attention on options for developing a new language. From the above set of requirements, the following design goals were synthesized.

C is a good language for satisfying the requirements for portability, execution speed, and compilation efficiency. C, and its associated operating system, UNIX, were well known to system developers at Rand, since UNIX has been operational within Rand for over 12 years. Therefore, the new language should be related to C in some manner.[1]

[1]Other languages that might be considered are: BASIC (portable, but not powerful enough), FORTRAN and COBOL (not sufficiently structured), LISP (full version not available on the VAX when we needed it), PL/I (restricts portability primarily to IBM mainframes), Pascal (similar advantages and disadvantages to C, with C better known at Rand), and Ada (no compiler had yet passed the DoD standard, and there were insufficient program development tools available when the development effort began).

III. REQUIREMENTS FOR A NEW PROGRAMMING LANGUAGE

The previous section outlined a set of problems, constraints, and needs for a programming language arising from the RSAC project. Project discussions about these needs led to a first conclusion: moving the entire ROSIE system onto a VAX minicomputer would not meet all project needs.

Project members therefore distilled the earlier discussions into a set of requirements for a programming language:

- **Speed in execution**. The new language should execute at least hundreds of times faster than did the same rule-based models in ROSIE.
- **Speed in recompilation**. RSAC "agents" are developed and enhanced over time. Writing a new agent requires perhaps thousands of compilations of the code at various stages of development to achieve a stable, useful agent. (We are planning an interpretive version of RAND-ABEL for developmental purposes, but it is meant to supplement, not replace, compiled RAND-ABEL.)
- **Portable**. The inability to move existing code easily from a DEC-20 to a VAX computer helped precipitate a remedy. Project members did not want to be in the same situation again, and the RSAC had a requirement that its programs be transferable to government agencies. Thus, the language chosen to represent RSAC agents should be available on a number of popular computers, and easily portable to new ones.
- **Readable by "real people."** Real people are defined here as civilian or military strategic analysts with knowledge of a subject domain (e.g., NATO defense forces, Soviet command and control systems) or the RSAC project's Department of Defense sponsors interested in studying models developed by Rand analysts. Almost all such people are not professional programmers. Yet major parts of the programs being developed for the RSAC were large, complex systems similar to "expert systems" in artificial intelligence, containing hundreds or thousands of rules. This type of program was written with heavy use of enumerated variables (so that lists of choices and other finite lists of elements could be highly readable) and If . . . then rules,

In considering how to move, or recode, the Scenario Agent's ROSIE code, our colleague Jean LaCasse studied the actual ROSIE code representing the agent, and found that it did not use all of the special features of ROSIE, such as the built-in "inference engine" that automatically applies rules to a situation represented by a data base. It might therefore be possible to reprogram the agent(s) in a simpler language, so it would not be necessary to move all of the ROSIE system onto the VAX. Thus began a set of language design and discussion meetings regarding new programming options.

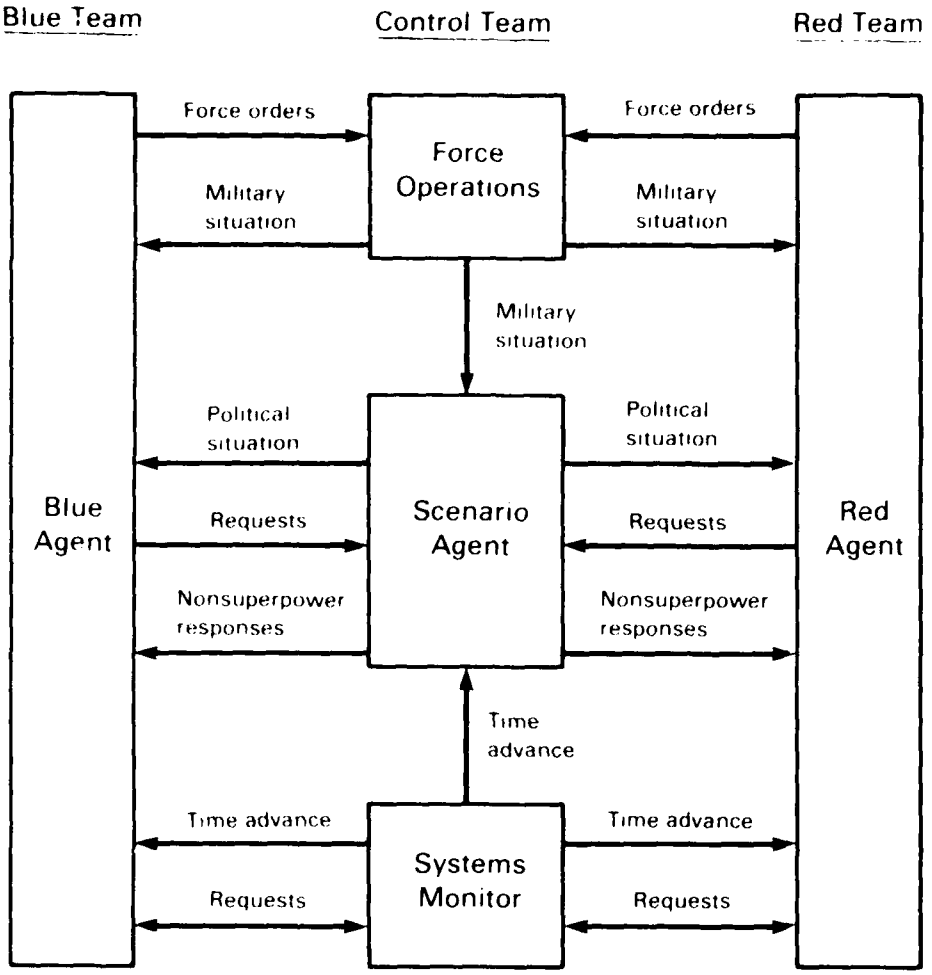


Fig. 1—Structure of RSAC automated political-military gaming

At about this same time, we began studying how agents representing the Red and Blue forces could be programmed within a next-generation Mark III version of RSAC. The notions suggested in a conceptual model design [Steeb and Gillogly, 1983] required many rules, games within games, and effective communication between agents. Also required were explanatory facilities for debugging and to aid analysis of a model. All this led to the need for some new programming features that were not then available, or that could be constructed only with difficulty.

II. RSAC DEVELOPMENT PRIOR TO RAND-ABEL

The Rand Strategy Assessment Center is, at this writing, operating the Mark III version of a war-game modeling and simulation system [Davis, forthcoming]. The Mark I version was demonstrated in 1981. The Mark II version was begun in January 1982. It consisted of a Blue Agent, a Red Agent, a Scenario Agent, a Force Operations module, and a Systems Monitor [Davis and Winnefeld, 1983]. Figure 1 shows the overall architecture of the Mark II RSAC system.

By late 1982, the Mark II system was operating as a collection of computers and programs, with people acting as manual links between the various system components. The Scenario Agent, containing logic for the behavior of countries other than the Soviet Union and the United States, was written in the ROSIE[1] language [Fain et al., 1981; Hayes-Roth et al., 1981]. ROSIE is a general-purpose artificial intelligence language developed at Rand for the building of expert systems and other AI applications. It is interactive and interpretive with a built-in inference mechanism for applying the rules comprising a program. The rules are written in a fixed grammar that resembles a stylized kind of prose English; the rules therefore appear quite readable, although the meaning a human reader extracts from a ROSIE rule and the logic a computer extracts from it might differ in subtle but important ways. A typical ROSIE rule used in the Mark II version of RSAC is the following:

> *If Strait-of-Hormuz is blocked and the actor is economically dependent on Strait-of-Hormuz,*
> *let the actor's threat be indirectly serious*
> *and record serious [threat] as "economic losses from Hormuz blockage".*[2]

The [] brackets in the above example delimit an unexecuted comment.

The ROSIE language satisfied some of the RSAC's goals (namely, flexibility and readability) but it had some shortcomings: (1) Most importantly, it ran too slowly for this application, requiring minutes for each cycle of "play" of the scenario model within the simulation; (2) it was written in the LISP language, operating on a DEC-20 computer, and the DEC-20 was being replaced by a slower VAX computer.

[1]ROSIE is a trademark of The Rand Corporation.

[2]From Schwabe and Jamison, 1982, pg. 43.

The function is delimited by the Define and End statements. The Record statement generates an execution-time trace file. The table, a simplified example, is executable code. It orders 10 percent of U.S. tactical aircraft to Southwest Asia. It also orders deployment of one division of U.S. airborne troop

Although RAND-ABEL should be more generally applicable, the reasons for its character are best understood in the context of the RSAC project, as it existed just prior to RAND-ABEL's development. The next section describes this RAND-ABEL prehistory. With this background, we then describe the resulting requirements for a new programming language, our attempts to meet those goals, and our assessment of our success to date in doing so.

into C language source code, rather than complete compilation of it; and very strict type checking.

The RAND-ABEL language cannot be understood without some understanding of the Rand research project whose requirements led to its development and test: the Rand Strategy Assessment Center (RSAC). The RSAC is an ambitious effort to develop a war-game modeling and simulation system composed of programmed "agents" representing the decisionmaking of the hierarchical command structures of the United States and the USSR, as well as third parties involved in the scenario. (The purpose and design of RSAC are described in Davis and Winnefeld, 1983; Davis, 1984.) The logic governing the agents' behavior should be describable in RAND-ABEL by policy analysts and be modifiable and extendable by them. The resulting code should be readable by persons knowledgeable in the subject matter but unfamiliar with programming in general and RAND-ABEL in particular. Professional programmers or computer scientists should, however, be involved in the implementation of complex simulations to create a sound, modular foundation that permits graceful modification and extension.

The reader can assess whether some of these goals have been achieved by reading examples of RAND-ABEL code. Here is a typical example of a RAND-ABEL statement:[4]

```
If Deployment-authorization of SWA [Southwest Asia] is Yes
then perform Order-SWA-deployment.
```

Deployment-authorization is an array indexed by military theater, here the Southwest Asian theater. Brackets enclose comments, used here to indicate the meaning of the acronym SWA. "Perform" is a call to the function Order-SWA-deployment, which is operationally defined by the following RAND-ABEL function:

```
Define Order-SWA-deployment:

    Record "Deploying forces to Southwest Asia.".

    Table Deploy

    qty    #-% unit-type unit-owner     to-area
    ═══    ═══ ═══════ ═══════════     ═══════.
     10     %  Tacair    US            SWA
      1     #  Airborne  US            SWA

End.
```

[4]Reserved words in RAND-ABEL are shown in bold type in this report as an aid to the reader's understanding the structure of the language.

compound RAND-ABEL statement, containing both declarations and executable statements, could be used in place of a function call in the table header; the order of the declarations of identifiers determined their matching with the columns of data values. Finally, the link between identifiers and table columns was made much more flexible by creating column headers that were readable by both user and machine. These "text islands" (later extended to allow abbreviations and hyphens) name RAND-ABEL identifiers, and can therefore be matched by name, independent of their ordering, with declared variables or function parameters.

Throughout this process, a major contributing factor to RAND-ABEL tables derived from Paul Davis' continuing interest in decision trees as a natural means of logic representation. Although direct representations of tree structures have yet not been made available in RAND-ABEL, decision tables are providing a useful compromise as a natural and relevant means of communication. In discussions with the RSAC system design team, it became clear that the Table statement had become encompassing enough to include decision tables as a special case. The form and content of table statements in RAND-ABEL are discussed further in a subsequent section of this report.

V. SOME PHILOSOPHICAL UNDERPINNINGS GUIDING RAND-ABEL DEVELOPMENT

RAND-ABEL, or any other language, would quickly evolve into a hodgepodge if new features were tacked on as needed or as conceived. From the beginning, our development of RAND-ABEL has been strongly guided by a set of philosophical principles we have come to believe in through combined decades of the use of programming languages and the observation of others' use of them. We give here the main principles we believe have affected the design of RAND-ABEL. Some of these principles should be of general interest; others are more specific to formal language design.

BUILD THE SIDEWALKS WHERE THE PEOPLE WALK

This principle was discussed earlier, and is perhaps the most important. The RAND-ABEL language has been conceived first as a programming language for a particular class of users (in our case, strategic analysts, both within Rand and within the government), and secondarily for other simulation modelers. How these users conceive of their models, how they express them, and how it is natural for them to read and write code expressing these models are the primary criteria to be used in deciding what facilities RAND-ABEL has, and how those facilities are called upon. (An interesting but different approach from ours to designing user interfaces in response to user requirements is described in Good et al., 1984.)

LANGUAGES SHOULD BE SELF-DOCUMENTING

We have discussed our distinction between "self-documenting" and "English-like" languages. Programming languages should be self-documenting because the code rarely stays put; it is inevitably modified, extended, and rewritten—usually by people other than the original authors. This is especially true for the complex models worked on by teams of analysts and programmers in concert, as in the case of the RSAC. Effective use of tables, matrices, charts, and mathematical (or chemical, etc.) notation can often condense pages of verbose English-like code into succinct but eminently readable instructions for both man and machine.

MINIMIZE THE SCAN-AHEAD NEEDED TO UNDERSTAND A RAND-ABEL STATEMENT (*)[1]

A growing body of literature indicates that people understand sentences by absorbing k words, making a hypothesis about the meaning, then reading the next word. If the successive words confirm the hypothesis, they go on; otherwise they go back and rehypothesize. In earlier RAND-ABEL statements, such as the assignment statement:

```
Let country be choice using readiness as criterion .
```

it was difficult to make an early, correct hypothesis about a RAND-ABEL statement's meaning. (In the above example, it becomes clear that "choice" is a function only after reading ahead and detecting the keywords "**using**" and "**as**".) We therefore undertook to redesign parts of the language to minimize the scan-ahead that was needed to correctly parse a RAND-ABEL statement. To accomplish this, we have started all RAND-ABEL statement forms with a keyword to indicate the type of statement. For example, the above assignment statement now is written:

```
Let country be report from choice using readiness as criterion .
```

in which the phrase **report from** provides prior notice that a function call is involved. It turns out that minimizing scan-ahead for readers of RAND-ABEL has a serendipitous effect: it also makes parsing of RAND-ABEL easy for computers. Almost all of RAND-ABEL is LR(1), meaning that the interpretation of a statement can proceed from left to right, never requiring a scan more than one lexical token ahead to resolve any ambiguity. (The big exception to this is the table header, which is handled by a specially written program, not an LR(1) parser.)

It seems quite clear that people are adept at scanning and interpreting a linear language, such as normal English sentences. But they use a different mechanism to understand tabular displays—a mechanism much more akin to picture or pattern interpretation. By creating formal languages rich in two-dimensional patterns (now made possible by ubiquitous CRT displays, tablets, mice, and graphics printers) we can tap onto this other nonlinear information-absorbing mechanism people possess, and thereby transfer more information, faster and more succinctly, between man and machine. Candidates for such two-

[1]Design principles marked with an asterisk (*) are technically oriented and may be skipped by readers without a background in programming or language design.

dimensional information representation include: tables and matrices, two-dimensional mathematical notation, and networks and graphs. One of the authors [Anderson, 1977] has discussed an extension of normal computer parsing methods to handle the interpretation of such two-dimensional constructions.

TWO-DIMENSIONAL LANGUAGE STRUCTURES ARE GOOD FOR PEOPLE AND THEREFORE GOOD

Most computer languages do not use two-dimensional structure; for example, if they are indented to show nesting of statements, that indentation is ignored by the compiler. In this sense, the languages are not self-documenting, because the structure of the language itself does not aid in the interpretation. We believe that information conveyed by such two-dimensional forms as indenting should be interpretable equally well by man and machine. However, in spite of our enthusiasm for two-dimensional constructs in programming languages, RAND-ABEL at present pays no attention to indenting of programs; RAND-ABEL's compiler (called the RAND-ABEL Translator) therefore would *not* prevent misunderstandings such as in the following program:

```
If condition_1 then
     {
       For alpha:
           {
             Let r be report from function_1
                  using alpha as argument_1.
             If r > 2.5 then
                   { Perform function_2.
                     Let s be report from function_3
                          using alpha as argument_3.
             Let p be report from function_4
                          using r as argument_4.
           }
       Perform function_5 using n as argument_5.}
     }
```

In this example, it appears at first glance that three statements are executed in the iterative loop headed For alpha:

```
Let r ...
If r > ...
Let p ...
```

In fact, the third statement, "**Let** p . . .", is buried within the "**If** r . . ." statement, and is not on the same level as the other two state-

ments. Similarly, the last statement, "**Perform** function 5 . . .", is really within the scope of the **For** loop, not the next sequential statement after it.

For twenty years, programming language designers have made languages easy for machines to parse, because machines were expensive and slow. Also, language users were turned off by such formatting restrictions as FORTRAN statements that had to start in column 7 of a punched card, and vowed to make all future languages just a string of text with no artificial boundaries or restrictions; in addition, in an era of line-oriented text editors (punched cards being the lowest form of such), it was difficult to create two-dimensional structures. All of these constraints have been greatly reduced, if not eliminated.

Finally, we are reaching a stage where machines are fast enough and cheap enough that we can begin to optimize for people, not the machines. Two-dimensional text editors allow the cursor to be moved to any spot on a page, so that tabular arrays of data can be created in a natural manner. We have observed that people are very good at absorbing the meaning of a two-dimensional construct.[2] (The widespread popularity of interactive spreadsheet programs, especially with nonprogrammers, is one indication of the naturalness of two-dimensional information presentation.) By employing these constructs in our programming languages, we often can make the languages more concise and more readable. RAND-ABEL's Table statement is a first step. We believe much more can be done in this area, as our emphasis shifts away from the machines and toward people.

ORTHOGONALITY

The word *orthogonal*, in its simplest definition, means *mutually perpendicular*. In mathematics, two functions that define orthogonal planes are independent of each other. In the computer and other sciences, the term has come to mean that two concepts are independent of each other within one or more categories of interest.

We have strived for orthogonality among the various concepts and constructs of RAND-ABEL. More precisely, we believe that semantically independent constructions should have syntactically independent representations. For example, the "+" sign is used to represent addition of both integers and real numbers, because the underlying concept of mathematical addition is the same for both. The "+" sign is *not*, however, used in RAND-ABEL for string concatenation as it is in some languages because that is a semantically independent concept.

[2]Some interesting observations on the human predilection for two-dimensional rectilinear constructs is contained in Hayes, 1985.

In the many discussions during the design and implementation of RAND ABEL, our desire to achieve and preserve orthogonality of language concepts was a continual touchstone of the quality of our design.

EMPHASIZE READABILITY OVER WRITABILITY

Programming languages are both written and read. The attributes of readability and writability are not synonymous. A language such as APL might be quite writable by someone familiar with the language, because of its tremendous economy of style; however, APL is famous for allowing the production of "one-line programs" that are hard to read, because of this same extreme terseness of style.

Whenever a choice had to be made, we emphasized readability of RAND ABEL over writability for the simple reason that programs are read much more often than written. We note that in addition to nonprogrammers wanting to understand their logic, existing programs are also often read by programmers attempting to recall their logic and to modify or extend the programs. We know of no statistics on this, but it is likely that once a line of code is written, it is read ten to one hundred times during its existence, often by people with less knowledge of the programming language than the writer.

STRONG TYPE CHECKING IS GOOD (*)

A strongly typed language is one in which the data type of each identifier must be declared, with the usual restriction that it must be declared before the first use of that identifier. (Multipass compilers can relax this second restriction, but such relaxation leads to sloppy programming practices, so it is not encouraged. In RAND ABEL, it is not allowed.)

Strong type checking allows the types of all identifiers being used in operations to be checked, usually at compile time, to be sure that they are consistent. For example, a string cannot be added to an integer; only logical data types can be ANDed and ORed together; an enumerated data type cannot be assigned to a real (number) variable.

Some languages (most notoriously, PL/I) allow automatic coercion of data types, so that if an inappropriate data type is used, the compiler tries to "make it right" (e.g., by coercing a string of digits into a number, or an integer into a floating point number). We believe *impli-*

cut[3] coercion of data types in a programming language is not good practice, because (1) it leads to mistakes in interpretation between the writer and the reader (and due to the self-documenting nature of RAND-ABEL, each might assume they understood the meaning, although each understands a different meaning); (2) it leads to mistakes in interpretation between the writer and the compiler; and (3) representation of information is important (for example, whether a number is represented internally as an integer or floating point), and coercions change representation without informing anyone. RAND-ABEL does no coercion, with one exception: in certain cases, if an integer is used where a real number is required, that integer is interpreted as if it were a real number. RAND-ABEL does *not* coerce a real number into an integer, because it could well silently lose precision in the process.

DEFAULT DECLARATIONS SHOULD NOT BE USED

RAND-ABEL does not permit default declarations of data values; that is, there are no implicit declarations of identifiers, by their spelling or first usage, or whatever. We believe it is good programming practice to clearly define all terms in advance, and we require this discipline in RAND-ABEL programs.

[3]We use the term *implicit* coercion here in contrast to *explicit* coercion in which the programmer clearly signals which change of data type is to take place, as in the C language statement: "int var = (int) char_ptr;". This type of explicit coercion of data types is occasionally useful, although at times overused.

VI. IMPORTANT FEATURES IN THE RAND-ABEL LANGUAGE

There are features of the RAND-ABEL language that should be of interest to persons interested in modeling and simulation, to programmers, and to programming language designers. In approximate decreasing order of novelty, they are: two-dimensional constructs like order tables and decision tables; declaration by example; RAND-ABEL's translation into another high-level language; integral use of a data dictionary for linkage between modules; and facilities for co-routines. In addition, our use of very strict type checking, mentioned in the previous section, distinguishes RAND-ABEL from many current languages.

THE TABLE STATEMENT

General Comments

As mentioned earlier, one of the most important features of the RAND-ABEL language is its use of two-dimensional table statements. In fact, we currently (July 1985) have three types of RAND-ABEL tables: (a) function tables, (b) macro tables, and (c) decision tables. In each case the table includes a key word (**Table** or **Decision Table**), header names for the various table columns, and a series of lines within the table, each of which defines a separate RAND-ABEL statement. The table types differ only in the way they convey to the computer how to "read" each line of the table.

The Function Table

An example of a function table was shown above, on p. 10. It is a function table accomplishing the job of issuing a series of orders to be performed by a simulation model. The function Red-to-3rd-countries results in a series of messages being sent from Red to various third countries with particular requests (e.g., Red asks Afghanistan to take the Red side and to cooperate by allowing reinforcement. It makes no requests pertaining to Afghanistan participating in the war itself, either in Europe or Southwest Asia).

The Macro Table

By contrast with function tables, macro tables are self-contained, i.e., the instructions for how the computer is to read each of the table's lines are contained within the table header. In this particular example, we have a *decision table.* The material following the key word **Table** includes: declarations of local variables needed for the table construction; the macro itself (the inner If . . . then statement); and the column headers. In this example, only the last variable is the result of a decision and the other If variables are all connected by "and"s. More generally, however, the macro could use "and"s and "or"s (or even < s and - s) and could have several dependent variables (decisions). The **Break** appearing in the macro means that as soon as the computer reaches a line for which the conditionals prove to be true, the decision is made *and* control shifts to the first line of code following the table. This had advantages in writing certain types of succinct decision tables—advantages similar to the If-then-else construction.

```
If Current-situation is Eur-demo-tac-nuc

   ["Eur-demo-tac-nuc" represents the situation that one or both
    superpowers have used some tactical nuclear weapons in Europe,
    but have done so primarily for demonstrative purposes --i.e.,
    to coerce the opponent into terminating]

Then
{
  Table
  {
     Declare Basic-status# by example: Let Basic-status# be Basic-status.
     Declare Other-status# by example: Let Other-status# be Other-status.
     Declare Prospects#    by example: Let Prospect#     be Prospects.
     Declare Risks#        by example: Let Risks#        be Risks.
     Declare Escalation-guidance# by example: Let Escalation-guidance#
       be Escalation-guidance.

       If (Basic-status# is Basic-status or
            Basic-status# is Unspecified) and

          (Other-status# is Other-status or
            Other-status# is Unspecified) and

          (Prospects# is Prospects or Prospects# is Unspecified) and
          (Risks# is Risks or Risks# is Unspecified)
       Then
       {
            Let Escalation-guidance be Escalation-guidance#.
            Break.
       }
  }
[-------------------------------------------------------------------]
```

```
Basic-status# Other-status# Prospects# Risks#  /  Escalation-
                                                   guidance#
============= ============= ========== ======= / ================.

goals-met     --            --         --          Eur-term
progress      good          good       low         Eur-demo-tac-nuc
progress      marginal      good       low         Eur-demo-tac-nuc
progress      good          good       low         Eur-gen-tac-nuc
progress      marginal      good       low         Eur-gen-tac-nuc
progress      good          good       marginal    Eur-demo-tac-nuc
progress      marginal      good       marginal    Eur-demo-tac-nuc
progress      good          good       marginal    Eur-gen-tac-nuc
progress      marginal      good       marginal    Eur-gen-tac-nuc .

)
```

The Decision Table

Our next example is representative of a great deal of actual RAND-ABEL code in current Rand work. It accomplishes precisely the same thing as the macro table given earlier, but without the complex header. In this case, the key word **Decision Table** coupled with the separator / (separating the independent variables from the dependent variables) in the header line are sufficient to define the logic. Note, however, that *all* tables headed by **Decision Table** have the logic of If . . . and . . . and . . . Then . . . and . . . and . . . Break, whereas other combinations are possible with macro tables.

The decision table structure is very powerful, for both programmers and subject area specialists developing program logic. Note, for example, that the decision table is isomorphic with a decision tree, and that decision trees have long been an especially effective way to work because they break problems down into pieces and allow one to keep track of whether all the cases are being considered. Furthermore, decision trees (and to a similar degree, after practice, decision tables) are a highly effective way to *communicate* logic. Indeed, the motivation for decision tables in our work arose when the program director found himself very unhappy trying to review the logic of models written in long series of If-then-else statments: the individual statements were clear enough (either in ROSIE or RAND-ABEL), but comprehending the whole was quite another matter. Moreover, in practice, his review demonstrated that important cases *were* being excluded—which is a notorious problem in If-then-else code.

The following examples illustrate the differences among expressions of decision logic as a typical computer program, as a decision tree, and as a RAND-ABEL decision table:

(1) Programming language form (e.g., in RAND-ABEL without use of the Table statement):

```
        If X = x1
           Then
           {
             If Y = y1
             Then
             {

             If Z = z1
                  Then Let D be d1.
             Else [Z = z2]
                  Let D be d2.
        }
        Else [Y = y2]
        {
             If Z = z1
                  Then Let D be d3.
             Else
                  Let D be d4.
        }
Else [X = x2]
   {
     If Y = y1
     Then
     {
          If Z = z1
               Then Let D be d5.
          Else [Z=z2]
               Let D be d6.
     }
     Else [Y = y2]
          If Z = z1
               Then Let D be d7.
          Else [Z=z2]
               Let D be d8.
   }
```

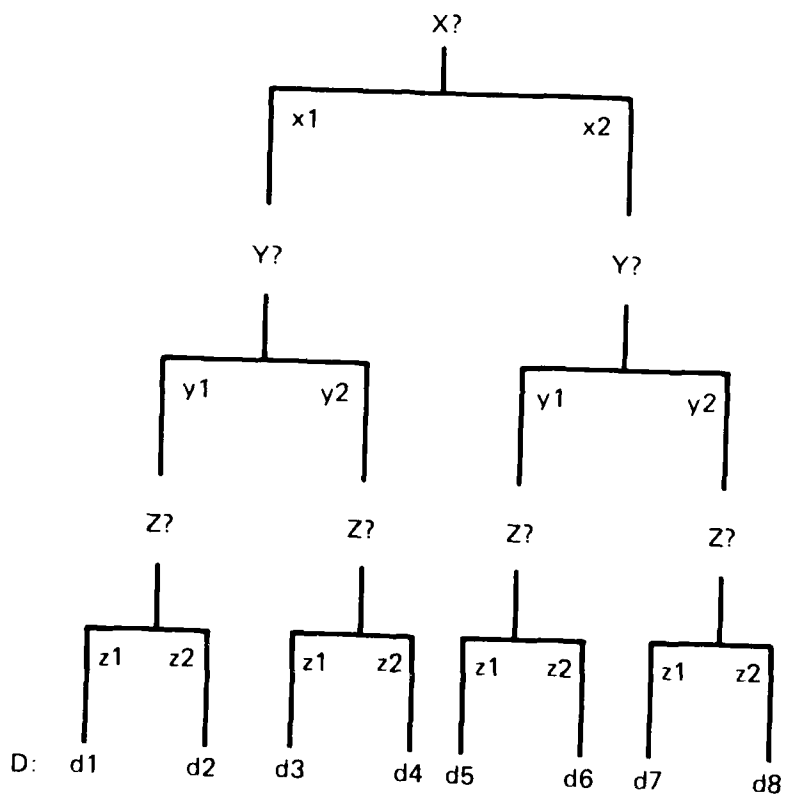
(2) Expressing the logic as a decision tree. (Note that this is *not* a programming language statement, merely a display of the logic that humans find useful in understanding the logic of a decision.)

Independent variables:

$$X = \{x1,x2\};\ Y = \{y1,y2\};\ Z = \{z1,z2\}$$

Dependent variable (the decision):

$$D = \{d1,d2,d3,d4,d5,d6,d7,d8\}$$



(3) RAND-ABEL code representing the same logic, using the Decision Table statement:

```
Decision Table
    X      Y      Z    /    D
   ------ ------ ------ / ------.
   x1     y1     z1          d1
   x1     y1     z2          d2
   x1     y2     z1          d3
   x1     y2     z2          d4
   x2     y1     z1          d5
   x2     y1     z2          d6
   x2     y2     z1          d7
   x2     y2     z2          d8.
```

Although we shall not discuss the matter here, we should also note that the decision table construction makes it especially easy to produce "explanation logs" that are rigorous statements of what logic path the computer followed.

As one additional example of a decision table, the following is equivalent to the macro table illustrated in the previous subsection:

```
Decision Table
                                                        Escalation-
Basic-status  Other-status  Prospects  Risks    /       guidance
------------  ------------  ---------  -------  / ----------------.

goals-met     --            --         --               Eur-term
progress      good          good       low              Eur-demo-tac-nuc
progress      marginal      good       low              Eur-demo-tac-nuc
progress      good          good       low              Eur-gen-tac-nuc
progress      marginal      good       low              Eur-gen-tac-nuc
progress      good          good       marginal         Eur-demo-tac-nuc
progress      marginal      good       marginal         Eur-demo-tac-nuc
progress      good          good       marginal         Eur-gen-tac-nuc
progress      marginal      good       marginal         Eur-gen-tac-nuc .
```

Complications and Fine Points

In practice, there are many complications in using tables—complications that could prove lethal if not resolved. Because of this, we spent considerable time developing techniques for handling them rather easily. The next example shows, for instance, how we can deal with long variable names heading up table columns. This particular table happens to be an unusual function table that calls a function, Initialize, that makes a decision (it establishes how long it takes the vari-

ous named countries to make decisions in the simulation, with this time being a function of some 12 variables). This table does *not* include all the cases needed to cover the complete space of possibilities.

Table Initialize

Region	Country-set [is it a country?, not a region/sea]	Superpower-set [is it one?]	Player-status [should the model simulate it?]	Borders-WP	Assertive-country [always fight on attack]	Nuclear-capable	Leader	Member-ship	Orien-tation	Temperament	Red-presence	Blue-presence	Decision-delay [1-366] days
Afghanistan	Y	N	Y	Y	N	N	USSR	--	Red	Captive	Major	None	1
Afghanistan	Y	N	Y	Y	N	N	USSR	--	Red	Captive	Token	None	6
Afghanistan	Y	N	Y	Y	N	N	USSR	--	Red	Captive	None	None	10
Arabian-Sea	N	N	N	N	N	N	--	--	--	--	--	--	--
Australia	Y	N	Y	N	N	N	UK	ANZS	Blue	Moderate	None	Token	2
Austria	Y	N	Y	Y	N	N	--	--	White	Reluctant	None	None	4
Belgium	Y	N	Y	N	N	N	US	NATO	Blue	Reliable	None	TripW	1

(In the above table as in other RAND-ABEL statements, unexecuted comments are delineated by square brackets. The three separate table rows dealing with Afghanistan are deliberate, showing as an example handling special cases of differing Red-presence. As special situations arise requiring "tuning" of such decision tables, additional rows can easily be added to describe the desired response to these situations.)

Our final example shows how long variable values within the table itself can be accommodated by a "wraparound" feature. Although it is uglier than our basic examples, the wraparound feature can be very useful.

Table Function-of-6

First-parameter	Second-parameter	Third-parameter	Fourth-parameter	Fifth-parameter	Sixth-parameter
12.5	Green	512	"String 1"	[1002]	(A + 10)
6.7	Blue	221	"String 2"	[994]	(A - 24)

REFERENCES

Anderson, Robert H., "Two-Dimensional Mathematical Notation," in K. S. Fu (ed.), *Syntactic Pattern Recognition, Applications*, Springer-Verlag, New York, 1977, pp. 147–177.

Davis, Paul K., and J. A. Winnefeld, *The Rand Strategy Assessment Center: An Overview and Interim Conclusions about Utility and Development Options*, The Rand Corporation, R-2945-DNA, March 1983.

Davis, Paul K., "Concepts and a Prototype System for Game-Structured Strategic Analysis," The Rand Corporation (forthcoming).

Davis, Paul K., *Experience in Applying Artificial Intelligence Techniques to Strategic-Level Military-Political War Gaming*, The Rand Corporation, P-6977, April 1984.

Fain, J. E., D. M. Gorlin, F. A. Hayes-Roth, S. J. Rosenschein, H. A. Sowizral, and D. A. Waterman, *The ROSIE Language Reference Manual*, The Rand Corporation, N-1647-ARPA, December 1981.

Good, Michael D., John A. Whiteside, Dennis R. Wixon, and Sandra J. Jones, "Building a User-Derived Interface," *Communications of the ACM*, Vol. 27, No. 10, October 1984, pp. 1032–1043.

Hayes, Brian, "Rank-and-File Thinking," *Lotus*, Vol. 1, No. 2, June 1985, pp. 73–77.

Hayes-Roth, F. A., D. M. Gorlin, S. J. Rosenschein, H. A. Sowizral, and D. A. Waterman, *Rationale and Motivation for ROSIE*, The Rand Corporation, N-1648-ARPA, November 1981.

Schwabe, William, and Lewis M. Jamison, *A Rule-Based Policy-Level Model of Nonsuperpower Behavior in Strategic Conflicts*, The Rand Corporation, R-2962-DNA, December 1982.

Shapiro, Norman, H. Edward Hall, Robert H. Anderson, and Mark LaCasse, *The RAND-ABEL Programming Language: Reference Manual*, The Rand Corporation, 1985a (forthcoming).

Shapiro, Norman, H. Edward Hall, Robert H. Anderson, and Mark LaCasse, *The RAND-ABEL Programming Language: An Interactive Tutorial*, The Rand Corporation, 1985b (forthcoming).

Steeb, Randall, and James Gillogly, *Design for an Advanced Red Agent for the Rand Strategy Assessment Center*, The Rand Corporation, R-2977-DNA, May 1983.

VIII. CONCLUSION

The Rand Strategy Assessment Center required a programming language for the development of complex strategic simulations that was fast, portable, easy to read and write, and permitted modular development of a large system by separate teams of developers. No existing language was found to have all the needed qualities. By creating a very-high-level language that is translated into C source code, it was possible quickly to tailor a language to the needs of these analysts, while retaining some major advantages inherent in the C language and its supporting UNIX system.

The RAND-ABEL language contains some novel features that we believe are important to its success. Perhaps most important is the Table statement, having a fully two-dimensional syntax that is difficult to describe as a traditional programming language, yet is understandable to any educated person. Also of interest are declarations using only examples of usage of identifiers, very strict type checking, a data dictionary facility for intermodule communication, facilities for coroutines, and processes as a RAND-ABEL data type.

The RAND-ABEL language is important for its understandability by specialists in the subject matter being modeled, combined with execution efficiency and facilities for handling complex models that are primarily qualitative, not quantitative. RAND-ABEL is a young language, still evolving. However, we feel it has been very successful in its intended application.

Explanatory facilities are critical in a language designed for building large, symbolic models. This is especially true when they involve complex control structures with co-routines. In models of this complexity, a "dump" or "trace" of all system behavior (even if selectively turned on and off) is inappropriate; it is too hard for the model developer to find key data at the right level of abstraction. RAND-ABEL allows reporting of system actions controlled by the user, in conjunction with a "stack" mechanism whereby the stored trace of entire lines of logic that do not prove fruitful can be discarded before they become part of the final explanatory output. The current explanatory facilities in RAND-ABEL are possibly adequate, but new approaches will be explored in the future that provide better selectivity and control over the level of abstraction desired.

Programming in current RAND-ABEL sometimes requires some knowledge of the C language (for example, to set wakeup rules for co-routines that are dormant). By completing the support environment for RAND-ABEL as part of the RAND-ABEL language itself, it should be possible to allow strategic analysts to perform their work within one consistent language, thereby reducing the entry barrier for persons developing RAND-ABEL programs.

- Incremental compilation of RAND-ABEL programs
- Better explanatory features
- Use of sets and their operations within RAND-ABEL
- A major generalization of the Table statement to succinctly perform groups of declarations
- Incorporation of structures (i.e., records)
- Extension of pointer facilities
- Transcendental functions
- Completion of a stand-alone environment for RAND-ABEL, so that it becomes a complete programming system not requiring the application programmer to know or use its underlying C language.

Sets are viewed as a useful supplement to the current enumerated variables of RAND-ABEL, in particular permitting a more easily understood form of "For" statement than currently exists. They will also allow writing decision tables succinctly by permitting constructions such as "<good" or ">marginal" to be values in the table—meaning that any value in the ordered set occurring before the value "good" or after the value "marginal" is acceptable in that position.

By using the Table statement to declare identifiers, the same succinctness that has been achieved in iteration and selection can be applied to the declaration of a sequence of identifiers. An example of a tabular declaration is contained in Sec. VI; that particular form, however, is unlikely to be the final version of this feature. We are beginning to recognize in the Table statement a great expressive power that we intend to exploit. Indeed, if we are successful, entire programs might be recoded as a series of tables in a fraction of their current size and seeming complexity.

Structures are groups of data composed of differing data types. In some languages, they are known as records, in which each field may have a distinct data type. Structures have been found useful in the C language, and we are planning an implementation that mirrors their use in C.

Pointers will be able to point at essentially all RAND-ABEL data types; they are based on the existing pointer facility of the C language.

Interpretive RAND-ABEL will be an extremely useful supplement to the current compiled version, permitting program changes and testing without lengthy intermediate compilations. In a separate but related development, we will be providing an interactive version of RAND-ABEL. We expect that the RAND-ABEL tutorial document will rely heavily on interactive RAND-ABEL to provide the reader with a set of exercises introducing language features in a "hands-on" manner.

For all the above reasons and others of less criticality, we believe RAND-ABEL is a success. It is being used at Rand by a broad range of strategic analysts and programmers on a daily basis for the development of complex models.

However, not all of our goals have been completely met. Although translation of RAND-ABEL into C and the subsequent compilation of that C code was initially very fast, it has become somewhat slower due to features being added to the language; on the other hand, with the continuing evolution of the language we have not concentrated on translation efficiency. It is therefore possible that considerably greater efficiency can be gained when we turn our attention to that area.

From our experience to date, it is now clear that an interpretive RAND-ABEL should be available for program development. In addition, the data dictionary should not require total recompilation every time some data item is changed; incremental compilation of the dictionary must be added so that delays in compiling do not impede the development process.

The most difficult area to assess is the readability and writability of RAND-ABEL programs. The Table statement, in particular, makes RAND-ABEL programs easy to understand by casual users and programmers alike. Important RAND-ABEL programs are being written by strategic analysts with some FORTRAN or BASIC familiarity who are not professional programmers, based on general control structures and data organizations established by programmers. In this sense, we have succeeded. However, we frankly had goals beyond these, which may well have been too idealistic; we hoped that RAND-ABEL programs would be *transparent*, not just readable. Such totally self-documenting transparency has not been achieved; portions of RAND-ABEL agents still look too much like computer programs, with nested If . . . thens, "For" loops, pages of declarations, and all the other baggage of programming languages. However, as analysts become more familiar with the power of the Table statement, major portions of their programs are being encoded in highly readable and succinct tables. There is some chance, however, that the level of complexity of these agents simply cannot be expressed much more simply than RAND-ABEL already permits.

FUTURE DIRECTIONS

Extensions of RAND-ABEL are planned in the following areas:

- An interpretive version of RAND-ABEL and an interactive RAND-ABEL that will aid in learning the language and in program development

VII. CURRENT STATUS AND FUTURE DIRECTIONS

PRELIMINARY ASSESSMENT OF RAND-ABEL

The RAND-ABEL language has been in serious use as a developmental tool for only about 18 months. It is therefore too early for a definitive statement regarding our successes and failures in achieving our goals. However, enough has been learned to provide an initial assessment. An overview is provided by the following table:

Success	Less Than Success
1. Runs fast	1. Incrementally compilable data dictionary required
2. People like it	2. Compilation too slow
3. Succinct programs that have proven in practice to be readable by nonprogrammers	3. Not as easy to learn and use as hoped
4. Portable among UNIX systems	4. C code unreadable
5. Allows creation of complex simulations by groups of developers	5. Good programming still requires a sense of style
6. Initial RAND-ABEL language developed quickly	

Certainly the greatest success of RAND-ABEL is its ability to run complex simulation agents many times faster than their equivalents in ROSIE. This alone has made RSAC goals achievable that otherwise would not have been. Perhaps the next most important success has been the Table statement. It compresses into a succinct, easily readable form logic that previously took pages of verbose text. More and more of the logic defining RSAC agents is being put into tabular form (for both iterative calculations and decision tables) as analysts become more familiar with its power. Third, the data dictionary is successful in recording the necessary information to allow teams of developers, working at different times and places, to communicate and coordinate in developing their interrelated processes.

same structure. Each section of the reference manual will be mirrored by a section with the same number and title in the tutorial. Each section of the tutorial will present interactive exercises on the subject matter in the corresponding section of the reference manual. Also, importantly, we plan to include in the tutorial descriptions of how to access and use the information in the reference manual, so that learning to use the reference manual is in fact an integral part of learning the language.

- *Examples as templates.* The RAND-ABEL reference manual contains examples of the various language statements, as do similar manuals. What is somewhat unique about our approach is that the examples deliberately and explicitly are not designed to explain the subject matter, but rather are templates that can be used as examples to be filled in when one needs that form of statement in a program. We have observed that the most common use of examples in a manual is as examples to be copied in programming, and have tried to accommodate this "learning by example" process. We have set the examples apart from the rest of the reference material typographically, but have placed them consistently below each statement's syntax description, so that one can scan and quickly find the example of a RAND-ABEL statement when that is needed.

These experiments in language documentation are being published approximately coincidentally with this document, so it is too early to assess their success. However, they are indicative of our feeling that experiments with language documentation *should* be performed, in an attempt to increase the effectiveness of this important and often overlooked aspect of computer languages.

facility, so we created a co-routining facility in C to support RAND-ABEL co-routines. These C language co-routine functions are a general feature that could be used by other C programs. They were written in C itself, using essentially no assembly language coding, and correspond closely with the UNIX primitives that manipulate processes; for this reason, they can be quickly understood by someone knowledgeable of the UNIX system.

RAND-ABEL functions are available for the following actions involving co-routines:

- Spawn a new process
- Terminate a process
- Put a process to sleep
- Associate wakeup rules with a co-process
- Remove wakeup rules from a co-process

DOCUMENTATION AS AN INTEGRAL PART OF LANGUAGE DEVELOPMENT

Part of learning to use a language is learning to use its documentation. Therefore, we feel the form and content of documentation for a programming language are an integral part of that language. In thinking about our intended users and how we have used language documentation in the past, we decided on a set of strategies and innovations in RAND-ABEL language documentation. The following steps have been taken to date:

- *Parallel tutorial and reference manuals.* Many languages have quite decent tutorial manuals that lead new users through the features of a language. They also have separate reference manuals that give a terse but complete description of language features, in alphabetical or some other categorized order. The problem is that in using the tutorial manual, the new user becomes very familiar with its contents and organization; but as he or she outgrows the tutorial and needs to access the reference manual, all the learning about efficient information retrieval in the tutorial manual is wasted, even counterproductive, in finding information within the reference manual. A discontinuity is introduced into what should be a smooth learning process.

  We are planning to address this problem by providing a RAND-ABEL reference manual [Shapiro et al., 1985a] and a tutorial manual [Shapiro et al., 1985b] *that have exactly the*

CO-ROUTINES

A strategic simulation, as developed within the RSAC, consists of a number of agents—for example Red, Blue, Scenario—all pursuing certain logic and reacting to events communicated to them via the World Situation Data Set. There are times in such a simulation when a process can proceed no further and must await later events caused by other processes. It becomes dormant and awakens at a later time. (We omit in this discussion how it might be awakened.) Upon awakening, a process might operate one of two ways:

- *Total reassessment.* It awakens with no knowledge of its, or any other processes', prior decisions, accesses a data base to determine the current status of the simulated "world," and determines on that basis what, if anything, to do.
- *Marginal changes to existing plans.* It awakens "remembering" the context in which it became dormant, discovers what has changed during its dormancy, and makes appropriate incremental changes to the plan(s) it had been pursuing.

In the first case, the normal mode of behavior for a simulated planning agent would be to totally "rethink" the situation upon awakening, perhaps embarking on a totally new plan that seemed to fit the current situation best. The new plan might in fact be best in some sense, but this sort of "optimize afresh at each stage" behavior is very nonhuman. People and their organizations are strongly biased toward continuing plans previously decided upon, making marginal changes to fit new circumstances.

Although RSAC agents are not necessarily programmed to mimic the behavior of humans or human decisionmaking organizations at each stage of their planning, these agents should nevertheless mirror some of the inertia inherent in the planning processes of people and their organizations.

If "total reassessment" were an acceptable mode of behavior, co-routines would not be necessary. Co-routines are, however, a very natural way to allow processes to wake up in context and thereby exhibit consistent, incrementally changing behavior.

The RSAC can be considered a series of co-routines, each running independently of the others, with the World Situation Data Set as the integrating link among them all. Indeed, the RSAC Red and Blue Agents are themselves a hierarchical series of co-routines, each written in RAND-ABEL.

As described earlier, the RAND-ABEL language is in one sense a preprocessor built upon the C language. C does not have a co-routine

DATA DICTIONARY

In the large simulation models being developed within the RSAC, individual modules are often revised, or new modules created, that must communicate with other established modules. Traditionally, it has been possible to compile separate components of a program and then to "link-edit" them together, resolving external references. There has also been the notion of a "common" data area into which all modules can read and write, for communication between separately developed modules.

These simple mechanisms are not nearly enough for the RSAC environment. A source of problems in the old FORTRAN common area is differing assumptions or perceptions by different programmers, possibly at various times during a complex program's development, regarding the size, type, and meaning of data in this common area. Since there was no type checking possible, or even a record of what assumptions were made by whom and when, subtle errors could arise that might be undetectable until unlikely combinations of events arose—and then the culprit module can be hard to trace, since many different modules could read and write into the common area.

To address these problems, a common data area for intercommunication among RAND-ABEL modules was established. In the RSAC, this data area is called the World Situation Data Set (WSDS). Entries in the WSDS are defined by a data dictionary, somewhat analogous to a data dictionary within a modern data base management system (DBMS). The data dictionary is established by means of data item declarations describing external data within individual RAND-ABEL programs.

The data item declarations in a RAND-ABEL program that establish entries in the data dictionary fall into three categories:

- *Defining attributes*—Information that actually affects the object code. Among the declarations possible in this category are method of access, owner, read/write permissions, read/write preferred formats for the data item, validation range or function to be used in checking for proper data, and prompt string to be used in requesting this data item.
- *Identifying attributes*—Information that is documentary but mandatory, such as author, date, and a prose description explaining the purpose and use of the data item.
- *Informative attributes*—Information that is optional. Examples are bibliographic references for further description of the data item, comments, and whether the form and content of the data item is under consideration and subject to change, or is confirmed as definite.

```
Table
  {
   Declare variable by example: Let variable be sample .
  }
          variable          sample
          ================  ========= .
          factor-1              1
          no-of-sides           1
          conf-level            1.0
          result-msg        "Success" .
```

This table is equivalent to writing:

```
Declare factor-1 by example: Let factor-1 be 1 .
Declare no-of-sides by example: Let no-of-sides be 1 .
Declare conf-level by example: Let conf-level be 1.0 .
Declare result-msg by example: Let result-msg be "Success" .
```

TRANSLATION OF RAND-ABEL INTO C SOURCE CODE

Goals for the development of RAND-ABEL included a high degree of portability, compilation speed, and efficient execution. Existing languages such as C met all these goals, although they did not meet others. However, a great deal of time and effort was saved by translating RAND-ABEL source code into C source code, so that the advantages of C could be enjoyed without reinventing a major, fast, portable compiler. By using the UNIX functions *lex* and *yacc* as the basis for our translator, still greater implementation efficiencies were made possible; the RAND-ABEL-unique portion of the translator in effect becomes the set of syntax rules for RAND-ABEL, plus a set of lexical functions to extract identifiers, operators, constants, and handle special constructs such as the table syntax.

We hoped at the start of the project that the C code resulting from RAND-ABEL translation would itself be easily readable by any C programmer. However, especially with the development of the data dictionary, this goal has not been met. The resulting C code is quite arcane. In practice, this has not been a significant problem.

of the declaration. RAND-ABEL's declaration is an extension of this concept, eliminating the need for a keyword like *int*.

The following are all valid RAND-ABEL declarations:

```
Declare confidence-level by example: Let confidence-level be 6.5 .
Declare Evaluate-situation by example:
        Let Success be report from Evaluate-situation
        using red as side and 5.5 [km] as distance .
```

This declaration method does require some care in use. The data type of the expression used in the example within the declaration-by-example must be determinable at the time the declaration is encountered. It must also be unambiguous. For example, one cannot write:

```
Declare confidence-level by example: Let confidence-level be 8 .
```

and then later assign the value 6.5 to confidence-level. If a number can take on non-integral values, it should be indicated as a decimal in the declaration, even though an integer is a special case of a decimal number. (Note that the "." at the end of the declaration is a statement delimiter, not a decimal point.)

Another advantage of declaration by example is that the reader need not learn any new special syntax for declaring a data item, once he or she knows how to use that item in a RAND-ABEL statement; giving an example of its use suffices, once the uniform "Declare *x* by example:" method of declaring is known. (This property is another example of orthogonality in the language, because each statement represents itself, and no other means is required.)

Our current form of the declaration tends to be quite verbose, especially when a number of variables are being declared, each requiring a separate RAND-ABEL statement. It is natural to ask: Why not offer declaration-by-example as an option, but allow more traditional forms of declaration as well? Our answer is two-fold: (1) We believe a language becomes harder to learn and use when there are several ways of accomplishing the same thing, and (2) we are currently designing extensions to the Table statement permitting its use as a succinct means of representing groups of declarations. For example, the following table (almost certainly not the final form to be chosen, but illustrative of the idea) declares four variables in what we feel is a succinct, readable manner:

In general, we have found that the Table statement is useful in situations where a number of statements need to be made, each having a parallel structure, but differing in details. Examples of such situations include a series of similar function calls (e.g., to generate orders to countries having varying characteristics) and decision tables that are equivalent to a series of rules, each having slightly varying conditions and resultant actions. Because of the success of the Table statement in representing these situations, we are planning to expand its use in other areas having similar parallel structure. We also recommend that programming language designers seriously consider the addition of table structures to other languages, even though this might require selective "breaking away" from the linear parsing algorithms in widespread use. We see no reason why table statements could not be integrated into commonly used programming languages, such as C, Pascal, or Ada.

DECLARATION BY EXAMPLE

In essentially all programming languages, identifiers can be declared to have a specific data type by explicitly listing the name of the data type. For example,

```
C:        int int_variable;
FORTRAN:  INTEGER INTVAR
Pascal:   VAR int_variable: integer;
```

This approach requires the programmer and the reader of such languages to learn a new set of keywords and syntax that represents syntactic concepts. The problem is greatly magnified in a language like RAND-ABEL, which allows enumerated values. Since each enumerated variable is itself a new data type, taking on a finite range of enumerated constants as its range, for consistency one should make up a new name for each of these user-defined new data types. For our community of users, who are not professional programmers, this is excess baggage. We have also observed that most people rely heavily on examples in understanding programming languages. Combining these observations, in a further attempt to make RAND-ABEL code self-documenting, even to the casual reader, all identifiers are declared by giving examples of their use; in fact, this is the only means currently available for declaring identifiers.

We got the idea of declaration-by-example from the C language. In C, to declare *f* to be a pointer to a function returning an integer value, one writes: *int (*f)()*. Note that one uses a call on the function as part

The mechanism for allowing such complications is one of the more novel features of the table facility. The table header contains a set of column headings used to match table columns to parameter names or variable names. In making this match, extra hyphens used to create a column heading are ignored, as are various other "spacing" characters. Also, an apostrophe (') can be substituted for one or more characters in a variable's name to create abbreviations within a column heading. Essentially, a column heading can be thought of as an island of text within a field of white space, where the island can have isthmuses and peninsulas as long as it retains its connectivity. In determining this connectivity, the following characters are treated as "white space": space, tab, newline, comments enclosed in square brackets, /, \, _, and =.

The syntax of RAND-ABEL tables is actually richer than these illustrations indicate. The interested reader is referred to the RAND-ABEL language reference manual [Shapiro et al., 1985a] for a more complete description of this statement.

It is important to note that we have introduced a table *statement* into RAND-ABEL, rather than treating tables as pure data to be read in. At first glance, data residing in a file would seem simpler: it could be changed without recompiling the program, it could be accessed by multiple programs, and a simple linear syntax could represent all RAND-ABEL statements.

We believe the data approach is not the correct one for our users' applications for several reasons: (1) Separating tabular data from the statements that read it violates our principle of self-documenting programs. The meaning of the table is buried in the program that reads it, with the data and the program usually residing in different places. (2) Our use of column headers meaningful to both user and computer is made possible by the intimate association of program and tabular data. (Headers are usually only comments, if present at all, in data files because they do not have the same form and content as the data they elucidate.) (3) As we extend the power of the Table statement to handle declaration of variables and other programming constructs, the distinction between program and data within the Table statement will become blurred to the point where separation would seem artificial and contrived. (4) Perhaps most importantly, having a uniform syntax for tables means that they can quickly be interpreted by the reader. Programs lacking this unifying principle read data in a variety of ways, each of which is uniquely crafted for the situation. Creating these special cases takes extra programming and debugging time, as well as complicating the task of reading and interpreting them.


FILMED
2-8