AT&T Technical Journal
Vol. 64, No. 2, February 1985
Printed in U.S.A.

Inverted Decision Tables and Their Application: Automating the Translation of Specifications to Programs

By L. S. LEVY* and H. T. STUMP†

(Manuscript received August 24, 1983)

Code generation techniques are used to program an application characterized by complexity arising from many special cases, and rapid changes due to advances in the state of the art. A formal notation—an inverted decision table written in a propositional logic form—is developed as a means for allowing expert users to describe the application in a knowledge base that code generators then can use to create production code. The complete system described in the paper automatically transforms a one thousand-page specification into a running program. The development of this system is an example of the formalization of the specification of a complex application. In this case the application is a part of the Job Management Operations System, an operational support system to aid regional Bell Operating Company construction and engineering processes. The techniques described, however, can be generalized.

I. INTRODUCTION

A complex applications program that involved enumerating and analyzing many special cases was successfully developed using the notation, tools, and techniques described in this paper. Such enumerative complexity is characteristic of many applications. In such cases a detailed knowledge of the application is in the minds of experts who

*AT&T Bell Laboratories. †AT&T Bell Laboratories; present affiliation Bell Communications Research, Inc.

are best able to state the rules of the game, but are not able to program them. Computer scientists, on the other hand, can program but do not know the application in depth. Thus, the skills of these two groups are complementary. Formal notation and tools enable them to cooperate by allowing them to focus on their respective skills.

In addition many applications have a *knowledge base* or set of rules that changes with advances in the state of the art. Since the most current information about the application is in the hands of expert users, it is desirable for them, and not programmers, to maintain the knowledge base of the program. This is possible when an appropriate notation exists for stating the application information, and when programming tools are available for exploiting this repository of information.

In this paper we discuss the development of a module of the Job Management Operations System (JMOS) application software. This development made a deliberate attempt to incorporate both (1) notation to allow expert users to describe their knowledge base in a formal specification, and (2) programming tools to generate the application code directly from this specification. Updating the programs to reflect changes in the state of the art then consists primarily of modifications to the knowledge base driving the program, and these updates can be performed directly by the expert users.

In the software development method that evolved, the specification is developed using *UNIX*™ operating system shell and editing tools. This specification is in a form that can be used as a basis for a document for users, and also as a basis for mechanical translation into programs. From the specification, decision tables are derived by shell programs. Then these decision tables are translated into PL/1 programs by a decision table processor developed for this application. If the specification has been properly composed, it then becomes a directly compilable program and is the *primary program description.* Although we have not achieved 100 percent of this goal, we feel that it is a realizable objective.

In Section II of this paper we describe the application and those characteristics motivating the present work. In Section III, the evolution of our approach is presented as it proceeded through several iterations. Section IV details the current approach to the specification-design process. The last section is an assessment of this methodology.

Our intended audience is programmers, specifiers (analysts and systems engineers), and managers involved in applications development.

1.1 Related work

Decision tables are well known and have an extensive literature.

Their use in specifications as a methodology relevant to correctness is described in Gerhart and Goodenough.[1] Our inverted decision tables resemble production systems.[2,3] *Computer* contains a survey of recent work in specification languages.[4] Our own efforts in this area are somewhat more special purpose. The emphasis on productivity of programmers is part of the broader concern for productivity gains in the general economy. Jones is a collection of articles describing work in this area.[5]

Code generation, long a part of compiler technology,[6] rises to a higher level in application code generators. Reference 7 is a broad survey of the state of the art in 1977, and specifically discusses decision tables. The work of one of the authors in an earlier application has been reported in Levinson, Levy, and Salisbury.[8] Other recent work is described in Refs. 9 through 14.

II. PROBLEM DESCRIPTION

2.1 The application

The JMOS system is a predominantly on-line operations support system that tracks and manages construction work in the outside telephone plant.* Its major functions are to

- Analyze data from engineering drawings and compute the work content of the construction job;
- Schedule such jobs to meet due dates, taking into account material availability, job priorities, and construction resources;
- Use daily reports of work progress to track the status of jobs;
- Analyze and report the ongoing performance of the construction forces;
- Interface with accounting, budgeting, and inventory systems to maintain company records, produce payrolls, and track operations and costs.

The application addressed in this paper deals with the first of the functions listed above. In particular it decomposes each work operation specified on an engineering drawing into two sets of components. One set comprises the physical tasks that must be undertaken to complete the work operation. The second set comprises theoretical elements that are used to compute a standard company work measurement index.

Associated with each of the tasks in the former set is a Standard Time Increment (STI) allotted to perform the task. These STIs are

* The outside telephone plant is the physical part of the telecommunications network that extends from the local central office to the customer's premises. It comprises cables, service wires, interconnection facilities, signal regeneration equipment, etc., as well as supporting structures such as poles, guys, anchors, strand, manholes, and conduit.

later accumulated and used as the basis for planning, scheduling, and tracking the construction jobs. Associated with each of the elements in the latter set is a quantity of Work Units (WUs), which are a relative expression of the value and complexity of the element. This measure allows construction work to be compared to dissimilar work in other areas of the telephone business. For example, craft performance is measured in terms of WUs per hour and cost performance is measured as dollars per WU.

To identify the STI components, the application program must derive and correlate information about such things as the nature of the work to be performed (e.g., install a cable), the location (e.g., buried in a trench), the field conditions (e.g., rocky soil), and the equipment to be used (e.g., backhoe). Derivation of the WU elements is similar in concept, but differs significantly in terms of analytic detail. Throughout this paper we will use STIs as illustrations, since they are generally the more complex of the two work components.

As an example, Fig. 1 is an annotated engineering drawing that calls for the installation of a new buried cable from a manhole to a pedestal (splice point). Note that there are four construction work operations, called job steps, in this example. Step 1 involves placing 500 feet of cable, 100 feet in a duct extending from the manhole, and the remaining 400 feet in a trench. Step 2 involves the placement of a terminal,

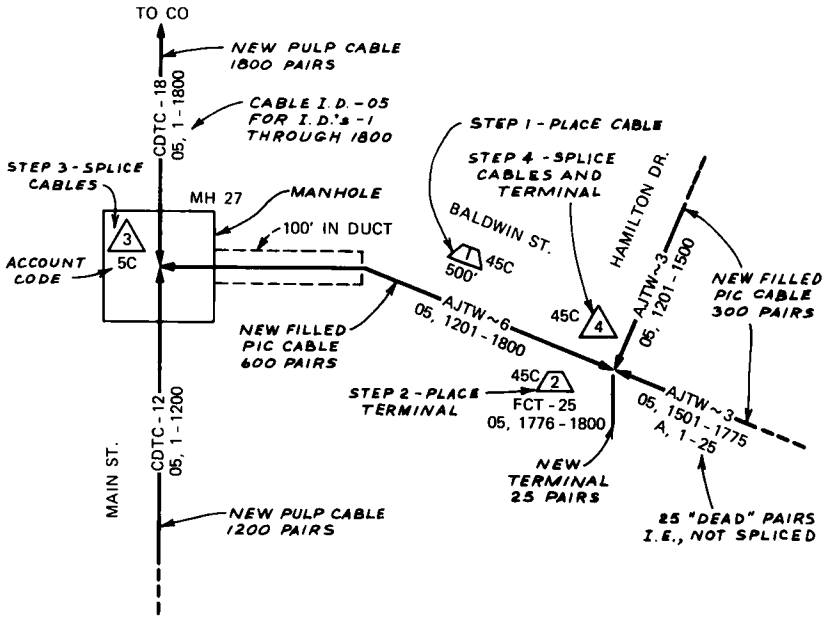

Fig. 1—Sample engineering drawing.

STEP	STI CODE	DESCRIPTION	QUANTITY	TIME
1	5P01	SITE SETUP – UNDERGROUND PLACING	1	0.9
	5P14	PUMP MANHOLE	1	0.3
	5P10	PLACE CABLE IN LATERAL DUCT	1	0.8
	4P01B	SITE SETUP – BURIED CABLE PLACING	1	0.4
	4P08	DIG TRENCH WITH TRENCHING MACHINE	400	1.9
	4P09	ADD FOR ROCKY SOIL	400	1.0
	4P13	PLACE CABLE IN TRENCH	400	0.6
	4P15	BACKFILL TRENCH BY MACHINE	400	1.0
			TOTAL	6.9
2	4P17	PLACE PEDESTAL	1	0.5
			TOTAL	0.5
3	S09	SITE SETUP – UNDERGROUND SPLICE	1	0.7
	S10	PUMP MANHOLE	1	0.3
	S11	SET MANHOLE PLATFORM	1	0.2
	S13A	PREPARE CABLE ENDS	3	0.9
	S13C	CLEAN FILLED PAIRS	600	0.6
	S13D	PREPARE CABLE ENDS	3600	2.2
	S22A	CLOSE Y SPLICE – NEW CABLE	1	0.8
	S39	JOIN PULP PAIRS	3000	4.8
	S41	JOIN FILLED PIC PAIRS	600	1.3
	S54	IDENTIFY AND TAG PAIRS	600	4.8
	S55	PROVIDE TONE FOR TAGGING	600	4.8
			TOTAL	21.4
4	S06	SITE SETUP – BURIED SPLICE	1	0.4
	S13A	PREPARE CABLE ENDS	3	0.9
	S13C	CLEAN FILLED PAIRS	1200	1.2
	S13D	PREPARE CABLE CORE	1225	0.7
	S40	JOIN AIRCORE PIC PAIRS	25	0.1
	S41	JOIN FILLED PIC PAIRS	1175	2.6
	S60	PLACE GRAVEL IN PEDESTAL	1	0.1
			TOTAL	6.0

Fig. 2—STI components derived for sample job.

or pedestal, in which the cable will be spliced to other cables and to a termination block, from which service wire can be extended into a customer's premise. Step 3 involves splicing (i.e., connecting) one end of the cable in the manhole to two other new cables installed on another portion of the drawing. Step 4 involves splicing the other end of the cable to two smaller cables and to a distribution terminal.

The STI components of these steps are listed in Fig. 2. Figure 3 illustrates the information supplied by the user, typically a clerk in the engineering office. The function of the application program is to analyze these input data and derive the correct set of work tasks.

2.2 Problem complexity

The complexity of the problem arises from a combination of static and dynamic factors. The static complexity is due to the inherent nature of telephone plant construction. The dynamic complexity of the problem is due to rather rapid changes in construction methods.

STEP	DATA ITEM NAME	ITEM 1	ITEM 2	ITEM 3	ITEM 4
1	WORK ENVIRONMENT	B			
	MATERIAL DESCRIPTION	AJTW-6			
	PLANT ACCOUNT CODE	45C			
	MATERIAL QUANTITY	500			
	PLACING ACTION	LDUC	STAN		
	ACTION QUANTITY	100	400		
2	WORK ENVIRONMENT	B			
	MATERIAL DESCRIPTION	FCT-25			
	PLANT ACCOUNT CODE	45C			
	MATERIAL QUANTITY	1			
	PLACING ACTION	STAN			
3	WORK ENVIRONMENT	U			
	PLANT ACCOUNT CODE	5C			
	FIXTURE FLAG	N			
	CABLE DESCRIPTION	CSDC-18	AJTW-6	CSDC-12	
	CABLE STATUS	N	N	N	
	CABLE CATEGORY	C	F	F	
	CABLE IDENTITY	05	05	05	
	PAIR RANGE	1-1800	1201-1800	1-1200	
4	WORK ENVIRONMENT	B			
	PLANT ACCOUNT CODE	45C			
	FIXTURE FLAG	Y			
	CABLE DESCRIPTION	AJTW-6	AJTW-3	AJTW-3	FCT-25
	CABLE STATUS	N	N	N	N
	CABLE CATEGORY	C	F	F	F
	CABLE IDENTITY (1)	05	05	05	05
	PAIR RANGE (1)	1201-1800	1201-1500	1501-1775	1776-1800
	CABLE IDENTITY (2)			A	
	PAIR RANGE (2)			1-25	

Fig. 3—Input data for sample job.

2.2.1 Static factors

Telephone plant construction involves the use of diverse materials, methods, field conditions, equipment, and types of operations. In particular this work employs over five thousand different material items in over 50 classifications. Some 265 different work tasks and 80 work-unit elements characterize just two major classes of work operations—placing and splicing.

Placing work includes the installation, removal, and rearrangement of materials and entails the use of many different types of specialized equipment and methods. There are also four distinct field environments to consider:

1. Aerial—cable and other materials supported by poles;
2. Buried—cable buried directly in the ground;
3. Building—cable and other materials placed on and within buildings; and
4. Underground—cable installed in underground conduits extending between manholes.

The following illustrates the complexity just of the installation of buried cable:

There are two common installation methods—plowing and trench-

ing. With plowing, one of two types of vehicles may be used—one with a static plowshare or one with a vibrating plowshare. With the trenching method, either a backhoe or a specialized trenching machine may be used to dig the trench. In some cases trenches are even dug by hand. For each of these methods and types of equipment the system must assign work tasks and associated times. Additional time is required if the soil is rocky. Furthermore, there are some ten specialized operations, used in conjunction with or in place of these standard methods, that must be selected on a case by case basis for each job step.

Splicing is the second major class of work performed by the construction forces. Here there are fewer methods and types of equipment involved than with placing. However, the normal splicing process is more detailed and complex than its placing counterpart, so that a given step often involves many more work tasks. Consequently, the analysis of splicing input is inherently more difficult and requires the use of rather esoteric algorithms.

For example, consider Step 4 in Fig. 1. As Fig. 1 shows, the input for this splice includes a description of each cable, along with the identification of its component pairs. Consider the cable at the top right side of the splice. It is designated as AJTW-3, which means it contains 300 pairs, has color-coded plastic insulation on the copper wires (known as PIC cable), and is filled with a petroleum jelly compound to render it waterproof so that it can be buried directly in the ground. The pairs in the cable are numbered 05, 1201 through 05, and 1500. Algorithms in the splicing module examine the pairs in each cable and determine which are being connected, disconnected, or rearranged. Subtotals, computed by type of cable (e.g., filled PIC), are used to derive the quantities of STIs and WUs for the splice. For this splice a total of 1175 filled PIC pairs and 25 regular PIC pairs are being joined. Similar analyses are performed to derive the other STIs and WUs for the step.

Describing the work operations outlined above requires the use of five different data-entry formats. The original word specification describing the application was over 150 pages long. The new specification described in this paper is over one thousand pages long.

2.2.2 Dynamic factors

The application faces additional complexity due to dynamically changing construction operations. New materials, tools, and techniques are constantly emerging. The application must be able to be modified frequently to keep abreast of these changes. For example, lightguide cable is currently being introduced very rapidly. This requires new placing and splicing procedures.

In addition to this normal evolution, the problem was recently complicated further by the reissuance of completely revised sets of both work tasks and work-unit elements. These revisions were triggered by the recent completion of extensive statistical studies of telephone construction methods.

III. EVOLUTION OF APPROACH

3.1 Informal word specification

Development of the prototype version of JMOS was started in late 1980. At that time the preliminary specifications described the derivation of work tasks (and associated standard times) to be associated with different outside plant construction operations. The document was over 150 pages long and its interpretation required extensive discussions between the software developers and the system engineer.

An example of the style of the original specification is the following:

Original Specification Style

If the step is entered on a Placing Work mask, the work environment is "buried," the material class is "cable," and one of the following conditions is met:

- The placing action is "trench with backhoe," "trench by machine," or "trench by hand."
- The placing action is "standard," the account code classification is "c," and the Profile Table specifies trenching with a backhoe or a trenching machine as the standard buried cable placing method.

Then for each foot of cable, generate:

1 4P13 Plc Ca in Trench

Although this example captures the flavor of the original specification, it does not indicate the level of complexity of the specification or of the changes that resulted from either errors in specification or typing. Suffice it to say that there were pages of the original specification in which the marginal notes and corrections were comparable to the typing on the page. Indeed, the changes were so extensive, and the specification so complex, that it was never reissued with corrections.

3.2 Manual translation to design

At that time it seemed clear that the amount of detail in the specification document, and the need for interpretation, would probably induce a fairly extensive debugging effort once the programs were written. When coupled with the fact that there were also undoubtedly errors in the specification itself, that some parts of the specification were incomplete, and, finally, that the specification would be extensively revised in 1982, we decided to use decision tables.

3.3 Decision table and processor

Decision tables are a well-known schema for specifying algorithms. They are, in effect, a deterministic version of the production systems currently popular in artificial intelligence applications. The advantages of decision tables are that they are quite change tolerant and fairly readable. Their disadvantage is that they represent a natural solution to only a part of the problem—there are algorithms that do not readily fit into decision table form.

A search of various software resources failed to turn up any decision table processor, both available and supported, that could generate PL/1 code—although we did find several that met two of these three criteria. Accordingly, a decision table processor was built that has been used in JMOS for the development of an operational prototype.

The format of the decision table that was used is shown below:

```
!D-TABLE:sti-p-bu
!VERSION:%I%,DATE,%D%
!=============================================
%logical
  buriedpla
  cable
  backhoe
  trencher
  hand
  standard
  pacc
  opf_backhoe
  opf_trencher
%conditions
  buriedpla
    masktype = 'p' & workenv = 'b'
  cable
    mattype = 'cable'
  backhoe
    plaatn = 'ctrb'
  trencher
    plaatn = 'ctrm'
  hand
    plaatn = 'ctrh'
  standard
    plaatn = 'stan'
  pacc
    paccls = 'c'
  opf_backhoe
    opf_tbl('bu_cbl_mth') = 'backhoe'
```

```
  opf_trencher
    opf_tbl('bu_cbl_mth') = 'trencher'
%actions
  s4p13
    CALL add_sti('4p13',matqty,jobstep_ptr,index,
      total);
    $indexer
%specifications
  buriedpla & cable & (backhoe | trencher | hand |
    (standard & pacc & (opf_backhoe | opf_trencher)))
    % 4p13
```

The interpretation of this decision table specification is as follows:

- Lines beginning with '!' are comment lines and are ignored by the decision table compiler.
- Lines beginning with a '%' are syntactic markers, which denote the headings of sections of the decision table.

There are four sections to the decision table.

1. Logical—This section consists solely of a list of the logical conditions occurring in the decision table. It is used to compile declarations in the object program.

2. Conditions—This section assigns values to the logical variables. In the object program, `buriedpla` will be true if `masktype` is `'p'` and `workenv` is `'b'`. The decision table will compile this into an assignment statement initializing the logical variable.

3. Actions—This section identifies action stubs of the decision table. In general, the actions are a sequence of PL/1 statements and PL/1 preprocessor statements. In the example, `s4p13` consists of a procedure call, and a preprocessor statement.

4. Specifications—This section relates conditions and actions. A set of specific actions to the right of the '%' symbol will be invoked if the Boolean expression to the left of the '%' symbol is true.

The decision table processor is a very straightforward implementation in the *UNIX* system, written in a combination of shell-level tools, primarily `awk`. Although there was no optimization performed in generating object code from the decision table, the performance was quite acceptable supporting our thesis that, in general, application code is not performance limited. However, had performance been a problem, it would have been possible to include optimization in the decision table processor without changing the decision table source. (The source code of the decision table is almost two orders of magnitude larger that the source code of the decision table processor, so clearly the processor would have been the place to optimize.)

The decision table approach did, in fact, accomplish what we thought it would. It was change tolerant, and more readable if only because it

was more concise than the PL/1 code. The decision table source resides in some 17 decision tables, each of which translates into a PL/1 program. These 17 PL/1 programs comprise in excess of ten thousand lines of code.

But once extensive debugging was under way, the decision table did not satisfy our needs completely, since there was no direct way to relate outputs to inputs. The same work task output could be generated by any one of a number of decision tables since these were organized by generic categories of the application. Thus, the problem was to develop a means of tracing an output back to its input.

3.4 Decision table inverter

The format that evolved to solve this problem of correlating outputs to inputs was an *inverted decision table.* These tables, which are generated by shell programs, list each output, the programs that generate each output, and, for each program, the conditions under which the output arises. This inverted decision table, which is an extended form of cross-reference listing, made the debugging easier and became an invaluable aid in this phase of the project.

The concept of a decision table inverter can best be visualized by considering the tabular form of a decision table:

Conditions					
	C1	T	F	T	T
	C2	T	—	F	T
	C3	F	—	—	T
Actions					
	A1	x	x	—	x
	A2	—	x	—	x

In this standard form of decision table, the relationship between conditions and actions is specified by the vertical columns. Each vertical column corresponds to a given set of input conditions. To determine what output actions apply, one starts with the specification of input conditions and searches for a column that matches those input conditions. The corresponding action portion of that column defines the output actions in that case.

Suppose that one wanted to know what input conditions could generate a given output. Then, even though the information is contained in the decision table, it is not in a convenient form. On the other hand, if one *inverts* the decision table, then the actions appear as the entry points that one uses to search for conditions that could generate those actions. The inverted decision table for our example is

Actions					
	A1	x	x		
	A2			x	x
Conditions					
	C1	F	—	F	—
	C2	—	T	—	T
	C3	—	—	—	T

Although the inverted decision table is the current conceptual model of our methodology, it was first conceived of as a generalized cross-reference listing to assist in debugging the prototype software. This generalized cross-reference listing, or inverted decision table, was produced automatically from the decision table source listings.

The major problem still remaining was that the translation from the specifications to decision tables was very labor intensive, entailing considerable human interaction between programmers and systems engineers to interpret the specifications. To solve this problem, the inverted decision table was used as a model for the next iteration of the specification documents. In this next iteration, the specification is used as an input to a processor that directly generates the decision tables, as an intermediate step, and then the PL/1 code. The specification itself then acts as a cross reference and is accurately embodied in the code.

It would, of course, be possible to produce the object code directly from the formal specification without producing the intermediate form of decision table. However, there were at least three reasons for not doing this:

1. In generating software it is desirable to decompose the process into a sequence of simple transformations such that each transformation is easy to implement efficiently and correctly.

2. The intermediate form of the code represented by the decision table can be used to provide several different versions of the final program. Some of these versions may have more extensive diagnostics to be used in the debugging phase. In fact, some of the versions were so used.

3. We had at this stage of the project a working decision table processor that did a significant part of the transformation for us.

IV. CURRENT APPROACH

The development of the prototype system began in the fall of 1980, and at that time the decision table processor was written. During the first half of 1981, the decision table processor was used to produce PL/1 code. In the late summer of 1981, the decision table inverter was written to assist in the debugging of the prototype. In planning for the

production release, we decided to use the inverted decision tables as a model for the software specification itself.

The current approach requires that a systems engineer write a set of specifications, which are then automatically translated into production software. The specification must conform to a rigid documentation format and use a propositional-calculus-type language with a limited vocabulary and a simple syntax. The specification is machine readable so that the two-stage code generation program can transform it first into decision table format and finally into PL/1 code.

Section 4.1 details the specification generation process. This includes a description of the specification structure and the language developed for the application. Also described are several software tools designed to aid in the preparation of the specifications themselves.

Section 4.2 describes the process of translating the specifications into finished code. This includes the two stages of the translation process as well as the handling of several special problems, such as the addition of program elements not expressly included in the specification.

Section 4.3 discusses the performance improvements that result from this current approach. These include programming efficiency, streamlined debugging, and vastly simplified maintenance.

4.1 Specification: language and preparation

4.1.1 Document structure

The specification document contains three parts—logic modules, data structures, and function definitions. A logic module defines a set of conditions required to generate a single STI task or WU element. There are about 650 of these modules, which constitute over 90 percent of the total specification document. The data structures define the vocabulary of the specification language. There are five parts to the data structure section, corresponding to the five data-entry formats. The function definition sections define 20 special functions referenced in the logic modules. These functions perform numerical computations or evaluate logical conditions that either cannot be handled by the logic modules or that are common to many logic modules.

Presently our code generation tools work only on the logic modules. The data structures are used to manually produce the PL/1 data declarations. The 20 functions are manually coded.

4.1.1.1 Logic modules. Each logic module comprises six parts (see Fig. 4, which defines the conditions for STI task `4P13`).

1. Title—The first line of the module is the title. It indicates whether the module applies to an STI task or to a WU element and identifies the specific task or element involved. Thus if the conditions

```
STI   4P13
-----------------------------------------------------------------
Description:
    Place cable in a trench.
-----------------------------------------------------------------

Level:   job step
Factor:  matqty

Conditions:

        masktype  =  epw   &
        <like_cable>  =  `true`  &
        {  plaatn  =  ctrb    |
           plaatn  =  ctrm    |
           plaatn  =  ctrh    |
           plaatn  =  ctr     |
          {  wrkenv  =  b   &
             plaatn  =  stan   &
             pacclass  =  c   &
             OPF(bu_cbl_plc_mth,argl)  ~=  plow
          }
        }



Notes:

   1.  No work environment check against actions ctrm,
       ctrb, ctrh, ctr, since these apply
       unambiguously to buried plant.

   2.  Actions bore and push are not allowed here
       (i.e., to generate cable placement), since they imply
       only that structure work is to be done.  If cable is
       placed in conjunction with this other work, then the
       EPW mask should be used to enter multiple actions.
```

Fig. 4—Logic module for STI 4P13.

specified in the remainder of the module are satisfied, then some quantity of this task is required to work the job step being analyzed.

2. Description—The second section is a word description of the STI task or WU element denoted by the title. In cases where more than one logic module applies to a given task, the description differentiates among the cases. The purpose of the word description is simply to aid persons reading the specification.

3. Level—The third section is a one-line entry that specifies the operating level of the logic module. The set of possible levels that may apply are defined by the data structures document and correspond to the hierarchical layers of data defined for each of the five entry forms. For example, for a placing step entry form, the possible levels are job step, supplementary placing action, and plant account code.

The level factor controls the processing of the logic module. The conditions in the module are evaluated once for each appearance in a job step of the data denoted by the level parameter. Thus if the level is job step, the module is evaluated once for a step. In the placing step example cited above (see also Fig. 4), the level is supplementary placing action, so the module is evaluated once for each action entered for the

step. (From 0 to 20 of these entries are allowed for a single placing step.)

The reason for varying levels is that different tasks apply at each of these levels. For example, since the task of setting up a work site applies to a job step as a whole, the logic module for a site set-up task is evaluated only once (job-step level) per step. On the other hand, a task associated with a placing action (e.g., STI `4P13` as shown in Fig. 4) must be evaluated for each action entered for a step in order to determine how often it applies.

4. Factor—The fourth section is another one-line entry. It specifies the quantity of the associated task to be generated if the conditions of the module are satisfied. This value must be numerical. It may be a fixed number (a constant), a variable name corresponding to an input-data item, or the result of a function computation. In the example in Fig. 4, the factor is the quantity associated with the placing action and represents the length in feet of the trench to be dug.

5. Conditions—The fifth section is the main body of the logic module. It defines the logical conditions that must be satisfied in order to generate the associated task. The conditions are written in a propositional-calculus-language form developed specifically for this application (see Section 4.1.1.2 for a detailed description).

The conditions specify the various combinations of data values that indicate the need for a given STI task or WU element. They may include references to data entered by the user, parametric data derived from these entries, the output of functions applied to these entries, or fixed parametric data extracted from tables that define the standard operating methods used by the local construction organization.

6. Notes—The last section contains an optional set of notes. Notes are provided to explain certain aspects of the conditions that may not be obvious to the reader. They may also be used to further differentiate a given module from others that apply to the same STI or WU.*

4.1.1.2 Data structures. Each data structure lists the set of data items used in the logic modules and functions associated with a particular data-entry form. Figure 5 illustrates the data structure for the placing-step entry form.

Note that the data items are partitioned by operating level (see Section 4.1.1.1). For placing steps there are three such levels. The job-step level contains all those data items that apply universally to the step (e.g., only one material description—`matdsc`—is entered for a placing step). The second level is supplementary placing action. Two

* Typical reasons for specifying multiple logic modules for a given task are that the task can be generated from more than one input form, at more than one data level, or with more than one quantity factor.

```
Structure:  EPW
------------------------------------------------------------
Description:
    Data structure for EPW (placing work) job steps.
------------------------------------------------------------

Layers:  step[1] supp_plc_action[2] plt_acct_code[2]

Contents:


         STEP

         jobstp#    --  job step number
         rskeystp   --  RS key step flag
         rsgrpid    --  RS group i.d.
         masktype   --  mask type
         pacclass   --  stppac(1) class
         wrkenv     --  work environment
         matdsc     --  material description
         matclass   --  material class
         mattype    --  material type
         matsize    --  material size
         plaby      --  placed by indicator
         factstub   --  factory stub flag
         matqty     --  material quantity
         plaatn     --  placing action
         rdsd       --  road side flag
         hivltpro   --  high voltage protection flag
         #suppatn   --  number of supplementary placing actions
         #stppac    --  number of plant account codes


         SUPPLEMENTARY PLACING ACTION

         suppatn    --  supplementary placing action
         atnqty     --  supplementary placing action quantity


         PLANT ACCOUNT CODE

         stppac     --  plant account code
         pacqty     --  plant account code quantity
         paccls     --  plant account code class
         meas       --  measured account flag
         plttype    --  plant type



Notes:

  1.  Note that the SUPPLEMENTARY PLACING ACTION & PLANT ACCOUNT CODE
      portions of the data structure are parallel.  They each relate
      directly to the STEP level.
```

Fig. 5—Data structure for placing data-entry forms.

data items are associated with each supplementary placing action in a step. Up to 20 pairs of such data may be entered for a placing step. The third level is plant account code. Two data items are also associated with each plant account code and up to three pairs of such data are permitted per step. Note that the second and third levels in this case are parallel, in that each homes on the job-step level. For other types of steps, the data-structure hierarchy may be three levels deep.

The location of a given data item in the data structure denotes its

availability to the logic modules. The general rule is that a data item may be used in a logic module or function for which the specified level is the same as or lower than the level at which the data item is defined. For example `matdsc` may be used in any placing-step logic module or function, since it is defined at the highest level of the data structure. However, `suppatn` may only be used in those logic modules or functions that apply to the supplementary-placing-action level.

The data items listed in the data structures define most of the variables used by the specification language. These data items may be entered directly by the user via an entry form (e.g., `matdsc`), or they may be derived from an entered value (e.g., `matclass` is a parametric data element associated with `matdsc` and extracted from a table of material descriptions). Other than items in the data structures, the only data variables allowed in the logic modules or functions are function references (e.g., ⟨`like_cable`⟩ in Fig. 4) or references to the Operations Profile Table, which defines the standard operating methods used by each local construction organization (e.g., `OPF(bu_cbl_plc_mth)` in Fig. 4 defines the standard method for placing buried cable).

4.1.1.3 Function definitions. A function is used either to perform numerical computations, which cannot be handled by the propositional calculus language, or to evaluate complicated logical conditions that appear in numerous logic modules or functions. The function definitions are structured much like the logic modules. Each has seven parts (see Fig. 6, which defines the logical function ⟨`like_cable`⟩ and Fig. 7, which defines the numerical function ⟨`#reg_pr_trans_sp`⟩).

1. Title—Name of the function.
2. Description—Word description of the function.

```
Function:  <like_cable>
-------------------------------------------------------------
Description:
    Determine whether material item for step is cable
    or some item similar to cable (eg, ground wire,
    air pipe, innerduct, or lightguide).
    Used on EPW steps and with RS logic.
-------------------------------------------------------------

Level:     job step
Step Type: epw
Data Type: logical

Conditions:

        matclass  =  cable     |
        mattype   =  ground_wire    |
        mattype   =  electrolys_wire     |
        mattype   =  air_pipe    |
        (  matclass  =  fiber_op_eqp     &
          (  mattype  =  innerduct    |
             mattype  =  lightguide
          )
```

Fig. 6—Definition for logical function ⟨`like_cable`⟩.

```
Function:  <#reg_pr_trans_sp>
-----------------------------------------------------------------
Description:
    Calculates the total number of regular pairs transferred for
    an SP step.
-----------------------------------------------------------------

Level:     job step
Step Type: sp
Data Type: numeric

Algorithm:

         total = 0 ;

         for ( i = 1; i <= #cblsht; i++ )

               total = total + #trspr ;

         return ( total - #spcpr ) ;
```

Fig. 7—Definition for numeric function ⟨ `#reg_pr_trans_sp` ⟩.

3. Level—Operating level of the function. It is basically the same as defined for the logic modules (see Section 4.1.1.1). There is one slight difference, however: In numerical functions it is possible to selectively operate at levels lower than the one specified for the function as a whole by means of an iteration segment. These segments are denoted as *for* loops in the function algorithm.

4. Step type—Denotes the type(s) of data-entry form(s) to which the function applies.

5. Data type—Denotes whether the function computes and returns a numerical value, or evaluates a set of logical conditions and returns a true or false value.

6. Conditions/algorithm—The main body of the function. If the function is logical, this section contains the conditions that must be satisfied in order to return a value of true (if the conditions are not satisfied, false is returned). The conditions are written in the same propositional-calculus-language form used in the logic modules (see Section 4.1.1.1). If the function is numerical, this section contains the algorithm used to compute the numerical value. The algorithm is written in a predicate calculus form and includes iteration capabilities (patterned after the C programming language).

7. Notes—An optional set of notes to provide an expanded explanation of the function to the reader.

4.1.2 Specification language

As noted previously, the conditions section of a logic module or function is written in a propositional-calculus-language form. This section describes some of the grammatical rules of that language, incuding the syntax and vocabulary.

4.1.2.1 Syntax. The primary syntactic rules are:

1. A condition statement comprises one or more logic expressions

```
<expression>    ::= <expr1> | <expr1> or <expression>
<expr1>         ::= <expr2> | <expr2> & <expr1>
<expr2>         ::= <variable name><logical operator><data value> |
                    { <expression> }
<variable name>     ::=     any variable listed in the data structure |
                    <function> | <opf ref>
<opf ref> ::=  OPF(<opf op name><opf arg position>)
<opf op name>   ::= any operation listed in the OPF table
<opf arg pos>   ::= arg1 | arg2 | arg3
<function>      ::=     < <function name> >
<function name>     ::=     any function listed in the function definitions
<logical operator> ::= <lexical op> | <numerical op>
<lexical op>    ::= = | ~=
<numerical op>  ::= = | ~= | < | <= | > | >=
<data value>    ::=     <variable name> | <constant>
<constant>      ::= <number> | <string> | `true` | `false`
<number>   ::=  any integer or decimal number
<string>   ::=  any character string
```

Fig. 8—BNF syntax of specification.

joined by the connectors "and" (&) or "or" (|), and possibly grouped by curly brackets, { }.

2. A logic expression is a comparison between a data variable and a data value of the form

⟨variable name⟩ ⟨logical operator⟩ ⟨data value⟩

3. Permissible logical operators include = and ~=, which may be used with any variable, regardless of type. The other operators, ⟨, ⟨=, ⟩, and ⟩=, may be used only with numerical variables.

4. By convention, each logic expression is stated on a separate line.

5. By convention, curly bracket groups are indented to show the level of imbedding. Also, the closing curly bracket is positioned on a separate line directly below the opening bracket in a given group.

6. By convention, connectors (&, *or*)* are placed at the end of the last line containing the first of the two expressions being joined.

Figure 8 formally specifies the syntactic rules of the language. The Backus-Naur Form (BNF) definition given here leaves out the format of the specification that was incorporated as essentially a syntactic component. It would be possible, by extending the BNF notation to include horizontal and vertical positioning symbols, to include the format as part of the grammar. For example, if CR is used as the symbol for a carriage return, then the second part of the first definition would read ⟨expr 1⟩ or CR ⟨expression⟩. A full discussion of the considerations of including formats as part of the grammar would be tangential to the main concerns of this paper.

4.1.2.2 Vocabulary. Other than the connectors and logic operators noted in the previous section, the specification language vocabulary is limited to two classes of terms: variables and constants.

* In the actual object language, "or" is denoted by |. Here we have reserved the use of | for the metalanguage disjunction.

The variables have already been defined (see Sections 4.1.1.1 and 4.1.1.2). They include the items listed in the data structures, the functions, and the references to the Operations Profile Table. Variables may appear on either side of the logical operator in a logic expression, though they usually appear on the left.

Constants may be numbers or character strings and may appear only on the right side of a logic expression. Common examples are

```
plaatn = plac
matsize <= 400
```

A constant must match in type and value one of the acceptable values of the variable to which it is being compared.

Two special constants are true and false. These correspond to the binary values of logical variables. They are used to make the specification more readable.

4.1.3 Derivation

Several programming tools were developed to aid in preparing the specifications. These enable the systems engineer to write a source file using severely abbreviated terminology and a very simple, loose format. The tools translate and expand the abbreviations into finished vocabulary and syntax. They also produce a finished format that adheres to all of the necessary conventions. For example, Fig. 9 is the source file corresponding to the logic module shown in Fig. 4.

Because of the magnitude of the specification, this tool provides several major benefits.

- It reduces typing significantly and enables much of the specification to be drafted directly on-line from working notes;
- It eliminates the need for word-processing support to convert the draft into a finished document—a finished specification is produced within minutes;
- It reduces typing errors;
- It guarantees that formatting is consistent and adheres to all conventions; and
- It simplifies correction and maintenance.

In addition to the translating and formatting programs, the *UNIX* operating system screen-editing tools were used to create templates of recurring logical conditions. These templates were then inserted where appropriate, usually requiring only minor editing changes or no changes at all. This eliminated the need to retype these sections. Nearly half of the specifications for placing steps were written using such templates.

```
gn GENERIC 1.0
tl SPECIFICATION FOR STI LOGIC -- BURIED PLANT PLACING WORK
id task STI 4P


hd 13
dsc
Place cable in a trench.
enddsc
lv step matqty
cn
epw &
<like_cable> = `true` &
{ pa ctrb |
pa ctrm |
pa ctrh |
pa ctr |
{ wb &
pa stan &
pc &
OPF(bu_cbl_plc_mth,arg1) ~= plow
}
}



Note
1.  No work environment check against actions  ctrm,
    ctrb, ctrh, ctr, since these apply
    unambiguously to buried plant.


2.  Actions "bore" and "push" are not allowed here
    (i.e., to generate cable placement), since they imply
    only that structure work is to be done.  If cable is
    placed in conjunction with this other work, then the
    EPW mask should be used to enter multiple actions.
endnote
```

Fig. 9—Specification source file for STI 4P13.

4.2 Mechanical translation to design

The objective of a mechanical translation to design is to take the specification from a file on the *UNIX* system Programmers' Workbench and after performing the appropriate transformations, submit the PL/1 source code corresponding to the specification for compilation via a Remote Job Entry (RJE) link. Once such a mechanical translation is realized, many problems at the object level can be resolved at the metalevel where they are often easier to deal with.

The complete process is a shell procedure called `spec to pli`. `Spec_to_pli` performs the following steps in sequence:

- Retrieve files—Retrieves the specification files.
- Preprocess specifications—A buffer step to filter out any notational variations in the specifications, and ensure that the input to the rest of the code generation process is under control of the code generator.
- Split out the component files—A single specification file may specify several decision tables. These are broken out in this step.
- Alphabetize the Booleans—A housekeeping routine. To make the

object code more systematic, the logical variables corresponding to the condition codes in the decision tables are alphabetized. (There are often close to one hundred such variables.)

- Shorten the lines—The code generator may use very long lines internally in its list of logical variables, but the PL/1 environment limits the line length to 72 characters.
- Rename files—An interface to match the file names to those expected by the decision table processor.
- Assemble files into decision tables—The four component parts of the decision table, described above, are assembled here.
- Generate PL/1 from decision tables—The decision table processor itself.
- Add JCL to PL/1 code—The job control statements needed by Time-Sharing Option (TSO) are added here.
- Cleanup—Remove miscellaneous working files that have been generated in previous steps.
- Send to TSO—Submit to RJE.
- Save the decision table and the PL/1 code—These items are stored in a separate directory, where they may be inspected, or printed out.

Translation from a specification to a design requires the addition of those elements that are present in a design but are not present in a specification, or that are present in both but in a different form:

- Variable declarations, defining the types of the variables, are generally not present in a specification;
- Organization of sets of variables into a larger structure is present in a design because of programming or database considerations, but is not usually present in the specification.
- If the design uses components that are referred to in the specification, but not described there, the design must solve the problem of resolving these references by including, or linking, the appropriate code.
- Occasionally, aliasing problems arise because of the preceding item, and the mechanical translation must deal with these.

The solution to these problems is a program that constructs a structure to which the code in the decision table refers. In this structure, the organization, types, and specific names of the variables as seen by the decision table are all under our control. While it would be possible to write the code manually for the procedure to populate this mediating structure, it is rather simple to generate both the declaration of the structure to be populated and the program to fill it from a small data dictionary maintained for this purpose. This data dictionary can be a rather simple, ad hoc facility since it is designed for a rather limited purpose.

4.2.1 Current status of the code generator

The current version of the code generator accepts as input the specification files, in the formatted version that is the customer copy of the specification. This is a propositional logic description of the conditions that generate the appropriate work credits—and, as noted above, is essentially an inverted decision table. The 20 specification files comprise 850 pages. The outputs of the code generators are 41 decision tables and their associated PL/1 programs, since most of the specifications produce more than a single decision table. The PL/1 code is approximately 16,500 lines of source code, the lines being relatively densely populated. Both the decision tables and the PL/1 code are stored in files, with the PL/1 being sent for compilation as well.

The code generator running time is about 0.1 second/line of system time, and 0.3 second/line of user time. In off hours the elapsed real time is about 0.6 second/line. Thus it is quite reasonable to regenerate the PL/1 code overnight if necessary.

4.2.2 Productivity

The literature on programming productivity describes several high-productivity methods for generating programs. Foremost among these are two techniques: reusable code and code generators. The reuse of code is applicable when one is developing a set of programs for similar applications, and common functions can be identified among the various applications. In that case the first application in the set to be developed pioneers the code, and subsequent applications reuse it. If the subroutines cannot be reused without modification, then they should be generalized in the hope of anticipating future requirements. Thus after several iterations, a considerable library of common functions and subroutines can be developed. The foremost example of reusable code is the extensive scientific subroutine package of Fortran.

In the present application, the cases of enumerative complexity are sufficiently specialized that it does not seem likely that any other application would have need for these particular specifications. For this reason we have chosen to use the techniques of code generation here.

As far as programmer productivity, a productivity rate in excess of ten thousand lines of tested code per year is readily achievable using code generation techniques. Indeed, the goal of code generation techniques is to ultimately put the programmer in the position of only developing and maintaining the code generators. In that case, applications experts would retain all of the knowledge about the application and would generate formal specifications that could be compiled directly. Indeed, the purpose of formalization is to abstract the seman-

tics of the problem so that the programmer can deal with it as a purely syntactic problem.

The code generation time required to produce all of the decision tables and associated PL/1 code for one 62-page specification was 1.5 minutes of system time—4.5 minutes of user time—in a *UNIX* system on a *VAX 11-780* computer.* The four decision tables generated in this case were 350 to 400 lines each.

Using this code generator, it was possible to generate a completely fresh set of decision tables and PL/1 code each time that the specifications were changed. In this way the specifications did indeed become the primary program description.

V. ASSESSMENT

The methodology and tools described in this paper were effective in producing a complex product on schedule, with the assurance that the program corresponded to the specification. Indeed, most of the specification files were revised, some of them extensively, within the final week prior to delivery. When such revisions were necessitated because the specification did not accurately state the conditions for various work credits, the production code was regenerated from the revised specification. This has the advantage that the last-minute changes to the code are not patches, and avoids the problem of introducing new bugs when fixing old ones.

The development of the module described in this paper supports the thesis that increasingly large roles in the software-development process should be played by code generation techniques. Indeed, the major problems encountered in the process of producing the code arose because we did not have some tools that should be part of the code generation facility. Examples are

- Syntax checking—There was no syntax check of the specification. Syntax errors in the specification showed up only as compile-time or run-time errors.
- Semantic checking—No tests were run to determine the correctness of the specification, until it was translated into production code. Most of the errors that were detected in the operational tests of the software were errors that could have been found by a diagnostic tool capable of executing the specification.

The code generation process also provided severe tests of other software. The symbols and data structures that were generated exceeded the capacity of some of the available tools. Examples are

- Variable names are constructed automatically by the code gener-

* Trademark of Digital Equipment Corporation.

ator and are semantically significant as an aid in diagnosis. Thus the statement in the specification:

```
pacclass = 'c'
```

generates the logical identifier:

```
pacclass_eq_c.
```

In some cases, this translation yields symbols that exceed the 31-character PL/1 limit on identifiers. (In those cases, provision was made in the generator to shorten the names. It was necessary in those cases to take special precaution to avoid name conflicts where two distinct names might have the same 31-letter prefix.)

- Lines constructed by the code generator often contained the entire specification of the conditions for a work task. In unabbreviated form these lines might be as long as 1,500 characters. Since `awk` is one of the primary tools used by the code generator and has a line length limit just under 512, this caused a problem. The solution was to use data compression and expansion at different stages in the generation process. (The primary tool used to effect the data compression and expansion is `sed`—the *UNIX* system stream editor.)

In summary, the development of formal specifications and their automatic processing by computer yielded significant dividends in the JMOS project. Considering that the technology of application software generation is still quite new, we expect that much greater dividends can be obtained as we learn how to manage and organize software-development projects to more fully exploit this technology. In addition, new and improved tools and techniques will arise from aspects of the code generation process that are different from the manually produced code.

Although we attempted to quantify the productivity gains that result from the processes that we have described, we feel that the technology is too new and there are too many variables to support particular claims. Much of the time of the authors was spent in developing tools and learning how to use them. Still a significant improvement was obtained in both the quality and quantity of the specifications and the delivered code, even when no allowances are made for the time spent developing the tools and learning how to use them.

REFERENCES

1. J. Goodenough and S. L. Gerhardt, "Toward a Theory of Test Data Selection," IEEE Trans. Software Eng., *SE-1*, No. 2 (June 1975), pp. 156–73.
2. R. Davis and J. King, "An Overview of Production Systems," *Machine Intelligence, Vol 8*, E. W. Elcock and D. Michie, Eds., New York: Wiley, 1977, pp. 300–32.
3. M. D. Rychener, "Production Systems as a Programming Language for Artificial

Intelligence Applications," doctoral dissertation, Carnegie-Mellon University, Pittsburgh, December 1976.
4. *Computer—Special Issue on Application Oriented Specifications, 15,* No. 5 (May 1982).
5. C. Jones, *Programming Productivity: Issues for the Eighties,* New York: IEEE Computer Society, 1981.
6. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Reading, MA: Addison-Wesley, 1977.
7. A. F. Cardenas, "Technology for Automatic Generation of Application Programs—A Pragmatic View," MIS Quarterly, *1,* No. 1 (September 1977), pp. 49–72.
8. E. Levinson, L. S. Levy, and J. B. Salisbury, "CARL—Experience of an Application Using Clusters," Proc. NCC, Chicago, 1981, pp. 241–8.
9. J. C. Zolnowski and P. D. Ting, "An Insider's Survey on Software Development," Proc. Sixth Int. Conf. Software Eng., Tokyo, 1982, pp. 178–87.
10. J. G. Rice, "Build Program Techniques: Objectives, Processes, and Processes," Interactive Systems, 1975 European Computing Congress, Online, Uxbridge, UK, September 26, 1975.
11. A. F. Cardenas and W. P. Grafton, "Challenges and Requirements for New Application Generators," Proc. NCC, Houston, 1982, pp. 341–9.
12. J. M. Grochow, "Application Generators: An Introduction," Proc. NCC, Houston, 1982, pp. 389–92.
13. R. L. Roth, "Program Generators and Their Effect on Programmer Productivity," Proc. NCC, Houston, 1982, pp. 351–8.
14. A. M. Goodman, "Application Generators at IBM," Proc. NCC, Houston, 1982, pp. 359–62.

AUTHORS

Leon S. Levy, B.A. (Physics), 1952, Yeshiva College; S.M. (Applied Science), 1955, Harvard; M. E. (Applied Mathematics), 1958, Harvard; Ph.D. (Computer and Information Science), 1970, University of Pennsylvania; AT&T Bell Laboratories, 1979—. Mr. Levy's work at Bell Laboratories has been in the field of application code generation. In 1983 he received a Distinguished Staff Award for this work. He was on leave for the academic year 1983–84 as a visiting Full Professor of Computer Science at Ben Gurion University in Beer Sheba, Israel. In 1984 he returned to AT&T Bell Laboratories. Member, ACM, ACL, and an affiliate member, IEEE Computer Group.

H. Theodore Stump, B.A. (Mathematics), 1967, Franklin and Marshall College; M.S. (Applied Mathematics), 1970, Stevens Institute of Technology; Bell Laboratories, 1967–1976; New Jersey Bell Telephone, 1976–1979; Bell Laboratories, 1979–1983. Present affiliation Bell Communications Research, Inc. When Mr. Stump returned to Bell Laboratories in 1979, he worked as a systems engineer in the area of construction operations systems. In 1983 he was appointed Supervisor of a group responsible for systems engineering of new loop technology and services maintenance. In 1984 he joined Bell Communications Research as District Manager for Construction Systems and New Technology Maintenance. Member, IEEE.