J. M. Lafuente

Some Techniques for Compile-Time Analysis of User-Computer Interactions

Compile-time techniques for analyzing user-computer interactions and the relationships and dependencies among items of data that exist during the execution of interactive application programs are presented. These techniques are useful in constructing efficient compilers for languages in which such interactions and data item relationships and dependencies are described by nonprocedural statements. The practical value of using nonprocedural descriptions is that they ease the task of the application programmer.

1. Introduction

The purpose of this paper is to describe techniques for analyzing programming language statements that specify interactions to take place between the user and the computer during the execution of an application program. The kinds of application programs that we have in mind include computer-aided instruction (CAI) systems and interactive data base systems, such as may be used by banks or airlines. During execution of a typical interactive application program, a user sits at a display terminal, where text is presented. The user types values, answers questions, or pushes buttons. Based on these actions, the system checks inputs for validity and consistency and responds appropriately to the user. The response may involve requesting additional values, changing the format and content of the information being displayed, or invoking a computational procedure. In any event, after the system provides its response, the user may then key in new values or push additional buttons, and the user-computer dialogue continues until terminated by one of a number of prespecified conditions.

Some interactive systems provide a language for formatting frames of information and for specifying simple interactions (*e.g.*, which frame is to be displayed next, depending on whether the user's answer to a question is "yes" or "no," when an error message is to be displayed, etc.). More complex interactions, however, cannot usually be specified with these languages, and their programming requires substantial effort on the part of the application programmer. Many of these languages and examples of how they can be used for programming user-computer interactions are described by Martin [1].

In order to simplify the task of the application programmer, programming languages to be used for developing interactive applications should allow the nonprocedural description of interactive behavior rules. This is the case, for example, in data type extensions to PASCAL described in [2] and in an extension to COBOL reported in [3].

The designer of a compiler for a language containing interactive behavior rules faces a new set of problems. First, an internal representation of the rules is needed. This internal mechanism may be constructed as the rules are parsed or in a subsequent phase of the compilation process. Second, the compiler must contain an analysis procedure to verify that the rules are consistent and possibly to eliminate rules that are superfluous. Third, since the programmer need not be concerned with the order in which the rules are tested or executed, the compiler has the task of determining, when relevant, the order of execution.

The compile-time analysis techniques described in this paper are the following:

1. A directed graph—called the *action/response (A/R) graph*—is defined as the mechanism for representing

Copyright 1980 by International Business Machines Corporation. Copying is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract may be used without further permission in computer-based and other information-service systems. Permission to *republish* other excerpts should be obtained from the Editor.

relationships between user actions and system responses as well as for representing constraints and dependencies among items of data.
2. An algorithm is given to analyze the A/R graph, and thus the behavior rules, for consistency.
3. The use of a parameterized decision table is suggested as one way to implement the analysis algorithm efficiently.
4. For those rules for which the order of processing is important, the rules are represented by a dependency graph, and a simple algorithm is given for determining the order in which the rules are to be processed at run time.

The techniques described in this paper are intended to apply to a variety of nonprocedural languages. We do not imply that these languages must be totally nonprocedural. For our purposes, we also regard as a *language* the nonprocedural statements of a general purpose programming language or the portion of a complex interactive system that deals with the description of the user-computer interface. For illustration purposes we have taken examples from the language extension described in [2]. A complete language using all of these techniques has not been implemented. However, the algorithms given in the paper have been separately implemented.

We begin with an example in order to give the reader an idea of the kind of nonprocedural specifications that the application programmer has to deal with. This is followed in Section 3 by a development of the concepts, terminology, and internal mechanisms used. Compile-time analysis techniques are then described in Sections 4–6. Section 7 deals with a more specific topic: the analysis of rules used to express conditions under which the value and properties of a variable should be changed by the system during execution of the user-computer dialogue.

2. Use of behavior rules for describing interactions

A behavior rule is either a Boolean expression that describes a constraint on data items and their values (*e.g.*, whether or not the user must select one or more options from the item), or it is a statement that expresses the conditions under which an error should be reported, a value should be changed by the system, the text being displayed should be changed, the user-computer dialogue is to terminate, etc. Behavior rules are defined more extensively in [2]. Here we simply give an example to illustrate the kinds of rules that we have in mind.

Consider the data items shown in Fig. 1 (reproduced from [2]), which represents a typical frame as the user might see it on a display screen. On the left are shown *key-in items*, that is, items for which the user must supply a value. On the right are two *menu* items, each containing two options. The user must select either *MALE or *FEMALE and either *SINGLE or *MARRIED. When all the requested information has been supplied, the user must depress the ENTER key on the keyboard. The ENTER key is regarded as a special kind of item, called an *attention* item.

BANK OF NEW YORK

NEW ACCOUNT

Enter information. Hit ENTER when done.
NAME:
SEL. SERVICE NO.: SEX: *MALE
*FEMALE

NO. CHILDREN:
SALARY: STATUS: *SINGLE
SPOUSE'S SALARY: *MARRIED

Figure 1 A typical frame.

To control the kind of dialogue that is to take place between the user and the system, the application programmer may wish to specify constraints and relations on the items shown in Fig. 1.

1. (Requirement) At least one option must be selected from *sex*; a value must be entered for *salary*.
2. (Limit) At most one option can be selected from *status* and *sex*.
3. (Requirement) *no. children* must have a nonnegative value (which is disregarded unless the user is married).
4. (Requirement) If the user is male, then a value must be entered for *sel. service no.*
5. (Exclusion) If the user is female, then a value should not be entered for *sel. service no.*
6. (Requirement) *Spouse's salary* is required if the user is married and *salary* is less than $15 000.
7. (Termination) The given information is to be processed when the ENTER key is depressed.

To express these constraints and relations, behavior rules may be specified as statements in the programming language as follows:

require *sex*, *salary*;
allow only 1 **option from** *sex*;
allow only 1 **option from** *status*;
require *nchild* $\geq$ 0 **if** MARRIED **selected**;
require *sel-ser* **if** MALE **selected;**
exclude *sel-ser* **if** FEMALE **selected;**

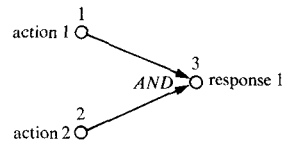

Figure 2 A simple effect graph.

enter *p1* **if** *A* **selected;**

enter *p2* **if** *A* **selected and any of** (*A2*,*B2*) **selected;**

allow only 2 options from *B*;

require *C*

Figure 3 A simple example of behavior rules.

require *spsalary* **if** MARRIED **selected and** *salary* < 15000;
terminate if ENT-KEY;

The last rule indicates that the dialogue is to terminate when the user depresses the ENTER key, provided there are no errors. If a rule has been violated, however, an error message is given to the user and the dialogue is not terminated.

3. Internal representation

In general, user-computer interactions are determined by relationships between user actions and system responses. By a *user action* we mean a terminal operator's action, usually recognized by the system via an interrupt. Actions include pushing a button, entering a value into a key-in item, and selecting an option of a menu item. A *response* is an operation or service performed by the system as a result of one or more actions, usually producing some indication to the terminal user, such as a change in the display, an error message, the lighting of an indicator, or the ringing of a bell.

In order to relate user actions to system responses, an underlying control structure is necessary. The compile-time analysis then consists in analyzing this internal structure and the relationships, constraints, and dependencies which it represents.

The kind of internal structure we use is the A/R graph mentioned earlier. Actually, the A/R graph is the superposition of two graphs: an *effect graph*, which describes how responses are produced from a combination of actions, and a *constraining graph*, which expresses dependencies and constraints on nodes of the effect graph.

- *Effect graph*

An *effect graph* is a directed graph in which the nodes represent actions, responses, or intermediate conditions, and the edges represent *semantic connectives*, as defined below. For example, the effect graph in Fig. 2 represents the combination of action 1 and action 2 producing response 1.

If we label the nodes 1, 2, 3, then we say that there is a semantic connective of type *AND* from node 1 to node 3 and from node 2 to node 3. The basic semantic connectives are *AND*, *OR*, and *NOT*, which correspond to logical operators.

Those nodes of an effect graph that represent user actions or particular conditions whose effects are to be determined are called *action nodes*. (For example, nodes 1 and 2 in Fig. 2.) Similarly, those nodes that are associated with responses are called *response nodes*. Nodes with no arcs entering them are called *start nodes*, or simply *S-nodes*. Nodes with no arcs leaving them are called *end nodes*. All start nodes of an effect graph are action nodes (usually representing user actions), but the converse is not always true. End nodes are usually designated as response nodes. We use the following notation for the nodes of an effect graph:

N = set of all nodes,
AN = set of action nodes,
RN = set of response nodes,
SN = set of start nodes.

Thus, $SN \subseteq AN \subseteq N$ and $RN \subseteq N$.

Each node of an effect graph has a unique integer associated with it, called the *node index*. For an effect graph with n nodes, we assume, without loss of generality, that

$N = \{1, 2, \cdots, n\}$.

Some restrictions are imposed on the effect graph:

1. A node can only have edges of one type leading toward it.
2. There are no circular paths (cycles).

As an example of how to construct an effect graph, suppose A and B are menu items having 2 and 3 options, respectively, and C is a key-in item. Consider the rules shown in Fig. 3. One possible translation of these rules would produce the effect graph shown in Fig. 4.

In this example, $A1$, $A2$, $B1$, $B2$, $B3$, and C are action nodes and correspond to user actions (selecting an option from A or B, or entering a value in C). Nodes A, B, $R1$,

and *R2* are response nodes. The responses associated with nodes *B* and *R2* consist in displaying error messages. The response "invoke procedure *p1*" is associated with node *A*, while the response "invoke procedure *p2*" is associated with node *R1*.

In general, the effect graph indicates how responses are produced from combinations of user actions. For example, if all of the options *B1*, *B2*, and *B3* were selected, a message such as "too many options have been selected from item *B*" would be displayed. This message is the system response associated with node *B*.

The effect graph constructed from a given set of behavior rules is not unique. For example, the subgraph shown in Fig. 5 can also be used to represent all effects caused by the second rule of Fig. 3.

The effect graph defined here is based on the notion of a *cause-effect graph*, which has been used to represent program conditions and associated observable effects for use in program analysis and testing [4]. A similar *Boolean graph* has also been used to represent switching circuits (*e.g.*, [5]).

The graph in Fig. 4 could also be described by a decision table. (See, for example, [6, 7].) Decision tables provide another technique for determining the effects caused by users' actions. However, a combination of behavior rules can produce a more complex structure than that shown in Fig. 4. Furthermore, the mapping of the behavior rules to their corresponding graph representation is often not as straightforward as shown above. We show later, for example, how some behavior rules cause other types of edges to be constructed to represent dependencies among nodes.

Definition A *path* of an effect graph is a sequence of node indices $(N_1, N_2, \cdots, N_r)$ such that N_1 is an action node, and there is an edge from N_i to N_{i+1} for each i, $1 \leq i \leq r-1$.

A key-in item is represented by an *S*-node in an effect graph. A menu or attention item with *m* options is represented by the subgraph in Fig. 6, where $\{N_1, N_2, \cdots, N_m, N_k\} \subseteq AN$. Each node $N_1, \cdots, N_m$ represents an option, and the node N_k represents the item itself.

• *Status properties*

Like a variable of a program, an item may have a value. However, it may also have properties which affect its appearance and behavior during the execution of the program (*e.g.*, whether or not the item actually has a value, whether or not the user is permitted to act on the item, whether or not it is to appear on the display screen, etc.). In particular, an item has a *status* property. The status of an item is either *selected* or *unselected*, indicating whether the item has a value or whether its value is undefined. An item is unselected until it receives a value, for example, by an assignment in the program or by user's input. Similarly, an option of an item may be selected or unselected.

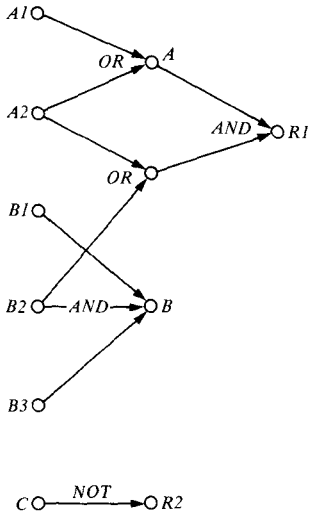

Figure 4 Effect graph corresponding to example in Fig. 3.

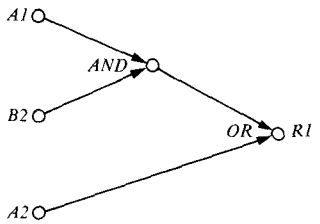

Figure 5 A subgraph corresponding to the second rule in Fig. 3.

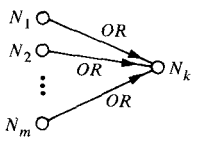

Figure 6 Menu or attention item with options.

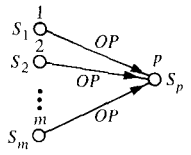

Figure 7 Derivation of possible status property.

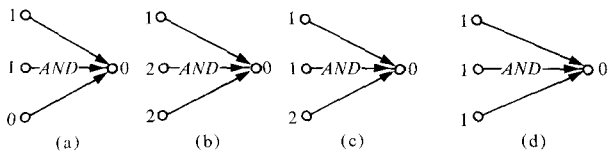

Figure 8 Checking a subgraph for inconsistencies.

We extend the concept of status property to nodes of an effect graph. For action nodes, the status corresponds to that of the item or option that the node represents. For convenience, we use 1 and 0 to represent node status (1 ≡ *true* ≡ selected; 0 ≡ *false* ≡ unselected).

Notation If x is a node index, we use the notation $node(x)\cdot p$ to represent the property p associated with node x.

One special property of nodes which we use is the *possible status*, denoted *ps*. For action nodes, this is defined simply as follows:

1. If $node(x)\cdot ps = 0$, then node x must be in unselected status.
2. If $node(x)\cdot ps = 1$, then node x must be in selected status.
3. If $node(x)\cdot ps = 2$, then node x can have either status.

The status property and the possible status property should not be confused. The *status* property can be used by a run-time interpreter to keep track of whether an item (or option of an item) is selected or unselected. On the other hand, the *possible status* property is used at compile-time to determine what status an item may have during the execution of a program. Intuitively, the condition $node(x)\cdot ps = 2$ indicates an uncertainty: node x can be selected or unselected.

For all nodes of an effect graph, the possible status property must satisfy certain relationships, depending on the specified semantic connectives. Consider nodes 1, 2, $\cdots$, m with possible status $S_1, S_2, \cdots, S_m$ connected by a semantic connective *OP* to node p, with possible status S_p, as shown in Fig. 7. Then,

1. If $OP \equiv AND$, then
   a. $S_p = 1 \Leftrightarrow S_1 = S_2 = \cdots = S_m = 1$;
   b. $S_p = 0 \Rightarrow S_i = 0$ for some i or $S_i = S_j = 2$ for some i, j;
   c. $S_i = 0$ for some $i \Rightarrow S_p = 0$;
   where $i \neq j$ and $1 \leq i, j \leq m$.

The meaning of conditions 1(a) and 1(c) is clear. The intuitive meaning of condition 1(b) is: if node p has status 0, then one of two things must happen: either some node on the left side has status 0, or there is an uncertainty (at least two nodes on the left side have status 2).

Similarly, we define relationships for the semantic connectives *OR* and *NOT* as follows:

2. If $OP \equiv OR$, then
   a. $S_p = 0 \Leftrightarrow S_1 = S_2 = \cdots = S_m = 0$;
   b. $S_p = 1 \Rightarrow S_i = 1$ for some i or $S_i = S_j = 2$ for some i, j;
   c. $S_i = 1$ for some $i \Rightarrow S_p = 1$;
   where $i \neq j$ and $1 \leq i, j \leq m$.
3. If $OP \equiv NOT$, then
   a. $S_p = 1 \Leftrightarrow S_1 = S_2 = \cdots = S_m = 0$;
   b. $S_p = 0 \Rightarrow S_i = 1$ for some i or $S_i = S_j = 2$ for some i, j;
   c. $S_i = 1$ for some $i \Rightarrow S_p = 0$;
   where $i \neq j$ and $1 \leq i, j \leq m$.

Let us illustrate, informally, how the possible status relationship defined above can be used to check for inconsistencies. Suppose $m = 3$, $S_p = 0$, and $OP \equiv AND$ in the subgraph of Fig. 7. Figure 8 shows several possibilities that may occur. Case (a) presents no problem; conditions 1(b) and 1(c) are observed. Similarly, case (b) does not violate any of conditions 1(a)–1(c); in particular, condition 1(b) is observed. Case (c) causes an inconsistency because condition 1(b) is violated. However, the inconsistency can be resolved by forcing $node(S_3)\cdot ps = 0$. Case (d) violates rules 1(a) and 1(b), producing an inconsistency that cannot be resolved.

This way of checking for inconsistencies is made more general and precise in Section 4.

- *Constraining graph*

The effect graph simply describes how responses are produced from a combination of actions in the absence of any interdependencies. It is also often necessary to represent dependencies and constraints on nodes of the effect graph. For this we introduce the constraining graph.

Definition A *constraining graph* is a directed graph (not necessarily connected) in which each edge represents one of two types of *links*: an exclusion link (*e-link*) or an implication link (*i-link*), defined below.

Given two nodes m and p, e-links and i-links express relationships on the possible status of nodes.

Definition An e-link from m to p satisfies the following constraints:

$node(m)\cdot ps = 1 \Rightarrow node(p)\cdot ps = 0$ and
$node(p)\cdot ps = 1 \Rightarrow node(m)\cdot ps = 0$.

Definition An i-link from m to p satisfies the following constraints:

$node(m)\cdot ps = 1 \Rightarrow node(p)\cdot ps = 1$ and
$node(p)\cdot ps = 0 \Rightarrow node(m)\cdot ps = 0$.

The representations of e-links and i-links are shown in Fig. 9.

Figure 9 Representations of e-links and i-links.

- *Action/response graph*

We are now ready to define our internal mechanism for analyzing behavior rules.

Definition An *action/response graph* is the superposition of an effect graph and a constraining graph. Each node in the constraining subgraph must also appear in the effect graph.

Restriction We impose the following restriction on the A/R graph: If there is a link from node j to node k, then

$k \in AN$ and $j \in AN \cup RN$.

Notation The notation $j \Rightarrow_{OP} k$ is used to specify that there exists an edge of type *OP* from node j to node k, where *OP* is replaced by *AND*, *OR*, *NOT*, e (e-link), or i (i-link).

4. Compile-time analysis

After the internal representation of the behavior rules has been constructed, it must be analyzed by the compiler. This process would reveal inconsistencies or potential problems that would arise at execution time. In this section we show how the A/R graph can be checked for the following conditions:

1. The A/R graph is inconsistent (over-constrained).
2. The A/R graph is consistent, but some items or options cannot be selected or unselected.

As an example, consider items A, B, C, D, and E, and the following behavior rules:

1. **exclude** A **if** E **selected;**
2. **require** A **if** (B, C) **selected;**
3. **require** B **if** D **selected;**
4. **require** A **if** C **not selected;**
5. **require** E **if** D **selected;**
6. **require** C.

The A/R graph corresponding to these rules is shown in Fig. 10.

In this example, the first rule produces the link $E \Rightarrow_e A$; rule 2 produces the edges $B \Rightarrow_{AND} F$ and $C \Rightarrow_{AND} F$ and the link $F \Rightarrow_i A$. The edge $C \Rightarrow_{NOT} G$ and the remaining i-links are similarly constructed from rules 3–5. Rule 6 does not produce any edges or links; it simply causes the possible status of node C to be set to 1. If the programmer had specified the two rules

require C; **exclude** C

an inconsistency would be detected during the semantic analysis of the **exclude** rule by the compiler, since we cannot have $node(C)\cdot ps = 1$ and $node(C)\cdot ps = 0$ at the same time.

As indicated earlier, the e-links and i-links are used to impose constraints on the nodes of the A/R graph. It is possible, however, that these constraints are inconsistent. We would be aware of this situation if we tried to make assignments of possible status (either 0 or 1) to the nodes of the graph while making sure that all possible status relationships defined in Section 3 (for semantic connectives, e-links, and i-links) are satisfied. We may find that any assignment leads to the contradiction $node(x)\cdot ps = 0$ and $node(x)\cdot ps = 1$ for some node x. We make these ideas more precise in the next subsection, "Dependencies and inconsistencies."

It is also possible that, even though the constraints are consistent, some S-nodes are restricted to have status 0 or status 1. For example, the reader may verify that node D in Fig. 10 cannot have status 1. This means that a dialogue would not terminate if a value were assigned to item D. The compiler, therefore, should make the application programmer aware of this situation.

- *Dependencies and inconsistencies*

Definition Let $N = \{1, 2, \cdots, n\}$ be the set of nodes of an A/R graph with corresponding possible status property

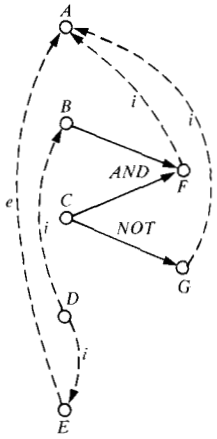

Figure 10 Example of an A/R graph.

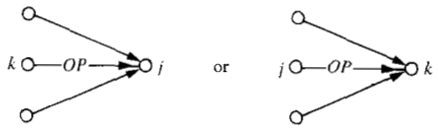

Figure 11 Edges into and out of node k.

given by the list (or ordered set) $P = (P_1, P_2, \cdots, P_n)$, that is,

$P_k = node(k)\cdot ps$ for $1 \leq k \leq n$.

Let S be a subset of N. The subset S is said to be *inconsistent with respect to* P if there exists a node $i \in S$ such that

1. If $P_i = v = 0$ or 1, then $P_i = v \Rightarrow P_i = \neg v$;
2. If $P_i = 2$, then $P_i = 0 \Rightarrow P_i = 1$ and $P_i = 1 \Rightarrow P_i = 0$.

The meaning of $P_i = v \Rightarrow P_i = \neg v$ is as follows: if $P_i = v$ and if the relationships given by the semantic connectives, e-link, and i-link are to hold, we must then have $P_i = node(i)\cdot ps = \neg v$, a contradiction.

Definition An A/R graph with possible status property P is *inconsistent* if the set AN (action nodes) is inconsistent with respect to P.

Before we give the algorithms for analyzing the A/R graph, we need one more definition.

Definition Given a node k, the *predecessor set* of k, denoted $PS(k)$, is the set of all nodes i for which there is an edge in the effect graph from i to k, *i.e.*,

$PS(k) = \{i \in N \mid i \Rightarrow_{OP} k$ where $OP \equiv AND$, OR, or $NOT\}$.

The algorithm for determining dependencies on the nodes of an A/R graph is now given. For simplicity, step 2 below is elaborated in the Appendix.

Algorithm A Let N be the set of nodes of an A/R graph, and let $S \subseteq N$. Let p be the corresponding possible status property of N, *i.e.*, $p_i = node(i)\cdot ps$ for each $i \in N$. An ordered set q is created, representing a new possible status property such that the relationships among nodes defined by the semantic connectives *AND*, *OR*, and *NOT*, and by the e-link and i-link, are satisfied. If they cannot be satisfied, then $q \equiv \varnothing$ (the empty set).

Step 1 (Initialization.) Set $q = p$. If $p = \varnothing$, the algorithm terminates.

Step 2 For each node $k \in S$, do the following:

(a) Remove k from S. Then, examine edges into and out of node k, as shown in Fig. 11. Here, $OP \equiv AND$, OR, NOT, e-link, or i-link. Let M be the set of nodes whose possible status can be derived from q_k. (This step is elaborated in the Appendix.)

(b) (Check status of dependent nodes.)
For each $j \in M$ do the following:
Let st be the derived possible status of j (0 or 1). If $q_j \neq 2$ and $st \neq q_j$, an inconsistency exists; in this case, set $q = \varnothing$, and the algorithm terminates. Otherwise, let $q_j = st$.

(c) Set S to $S \cup M$. If $S \neq \varnothing$, repeat step 2.

For convenience in the discussion that follows, we define a function *depn* which is implemented by Algorithm A. Given a set $S \subseteq N$ and an ordered set p representing the possible status of N, that is, $p_i = node(i)\cdot p$, then

$depn(S,p)$ = a nonempty ordered set q such that
$i \in N$ and $p_i \neq 2 \Rightarrow q_i = p_i$,
$i \in N$ and $p_i = 2 \Rightarrow q_i = 0$, 1, or 2,

provided that the constraints given by p can be satisfied; otherwise, $depn(S,p) = \varnothing$.

The function *depn* has the following characteristics:

1. $depn(\varnothing,p) = p$ (trivial case),
2. $depn(S \cup S',p) = depn(S',depn(S,p))$,
3. $depn(S,\varnothing) = \varnothing$ (trivial case),

and from (2) it follows that

4. $depn(S,depn(S,p)) = depn(S,p)$.

It is now a simple matter to give an algorithm to test an A/R graph for consistency.

Algorithm B Let AN be a set of action nodes, where $AN \subseteq N$, and let p be the corresponding possible status

property of N. The algorithm returns a Boolean result R (*true* or *false*) such that

R = *false* if and only if the A/R graph is inconsistent.

Step 1 (Initialization.) Set R to *true*.
Step 2 Let $NS = \{i \in AN$ such that $p_i \neq 2\}$.
Step 3 Let $q = depn(NS,p)$.
Step 4 If $q = \emptyset$, then set R = *false*, and the algorithm terminates.
If $q \neq \emptyset$, then
(a) Let $M = \{i \in AN$ such that $p_i = 2\}$;
(b) For each $k \in M$ do the following:

$$\text{Let } q_i' = \begin{cases} q_i \text{ for } i \neq k, \\ 0 \text{ for } i = k. \end{cases}$$

$$\text{Let } q_i'' = \begin{cases} q_i \text{ for } i \neq k, \\ 1 \text{ for } i = k. \end{cases}$$

If $depn(\{k\},q') = \emptyset$ and $depn(\{k\},q'') = \emptyset$, then set R = *false*.

• *Analysis of termination conditions*
The above analysis techniques can be extended to examine **terminate** rules. Consider, for example, the rule

terminate if ⟨*condition*⟩

We wish to verify that the ⟨*condition*⟩ is such that it can actually cause the user-computer dialogue to terminate. To do this, a response node, say r, is associated with ⟨*condition*⟩. Then, in the analysis procedure we set $node(r)\cdot ps = 1$ and verify that the set $\{r\}$ is consistent with respect to ps using Algorithm A.

• *Other dependencies and relationships*
We have so far restricted our analysis to Boolean conditions. Accordingly, the only semantic connectives that we have considered are *AND*, *OR*, and *NOT*. It is, however, possible to define other types of semantic connectives to keep track of other relationships and dependencies, such as the result of evaluating an expression and the number of options that have been selected from a menu item. Some of these connectives are defined in [3]. The problem is that these other types of connectives require *ad hoc* techniques and complicate the compile-time analysis procedures.

5. Analysis using Petri nets

Petri nets (see, for example, [8]) have been successfully used to analyze the flow of information in systems exhibiting concurrent activities or in which events can occur concurrently but there are constraints on these occurrences. It is possible to model the dependencies and constraints among the nodes of an A/R graph by a Petri net.

For example, consider the implication link $k \Rightarrow i\,j$. This can be represented by the Petri net shown in Fig. 12. The

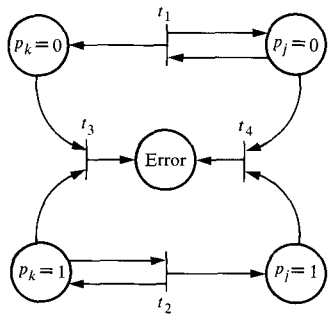

Figure 12 Petri net corresponding to the implication link $k \Rightarrow ij$.

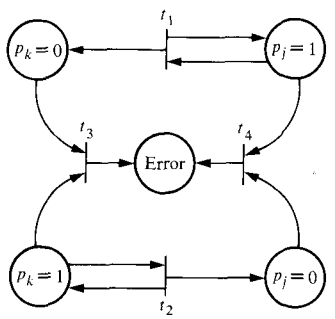

Figure 13 Petri net corresponding to the exclusion link $k \Rightarrow ej$.

two conditions $node(k)\cdot ps = 0$ and $node(k)\cdot ps = 1$ are represented by the two *places* labeled $p_k = 0$ and $p_k = 1$, respectively. Similarly, there are two places corresponding to node j. The place labeled "error" is used to represent an undesirable condition that results if an inconsistency occurs. When an inconsistency occurs, either transition t_3 or t_4 is enabled.

Similarly, the exclusion link $k \Rightarrow e\,j$ can be represented by the Petri net shown in Fig. 13. Nets for *AND*, *OR*, and *NOT* can also be constructed, although they are more complex, particularly as the number of nodes increases. Proceeding in this manner, we can construct one Petri net that models the entire A/R graph. The resulting Petri net appears more difficult to analyze than the A/R graph, but its construction can be used to show that an algorithm (such as Algorithm B) that analyzes an A/R graph by keeping track of possible status of nodes indeed exists.

Given an A/R graph with n nodes, we can construct a Petri net having a set of places $P = \{m_1, \cdots, m_n, r_1, \cdots,$

	CONDITIONS	RULES			
1	$ST[k] = 0$			X	X
2	$ST[k] = 1$	X	X		
3	$k \Rightarrow i\ j$	X	X		
4	$j \Rightarrow i\ k$			X	X
5	$ST[j] = 0$	X			
6	$ST[j] = 1$			X	
7	$ST[j] = 2$		X		X
	ACTIONS				
1	$ST[j] := 0$				X
2	$ST[j] := 1$		X		
3	$S := S \cup \{j\}$		X		X
4	$ST := \varnothing$ (error)	X		X	

Figure 14 Example showing how to construct a decision table with four rules.

$r_n, e\}$ and a set of transitions $T = \{t_1, \cdots, t_s\}$. Let $\mu^\circ = \{\mu_1, \cdots, \mu_n, \mu_{n+1}, \cdots, \mu_{2n}, \mu_{2n+1}\}$, be an initial *marking* of P, where

$\mu_i = 1$ if and only if $node(i) \cdot ps = 0$, for $1 \leq i \leq n$,
$\mu_i = 1$ if and only if $node(i) \cdot ps = 1$, for $n + 1 \leq i \leq 2n$,
$\mu_{2n+1} = 0$ (corresponding to e, the error place.).

The problem of whether an A/R graph is inconsistent can be stated in terms of the following problem for the above Petri net.

Problem Let $S = \{\mu \mid \mu_{2n+1} = 1\}$. Is the set S reachable from μ°, *i.e.*, can the Petri net be executed so that there is a final marking $\mu' \in S$ such that $\mu'_{2n+1} = 1$?

The above is the reachability problem for Petri nets, which is known to be solvable. It follows, therefore, that the consistency of the corresponding A/R graph can be determined.

6. Using decision tables

Decision tables can be helpful in implementing algorithms to analyze constraints and dependencies arising from the processing of behavior rules. One such use is illustrated here, although the reader may skip this section without losing the continuity of the presentation. This technique was used to implement step 2 of Algorithm A.

For convenience, we introduce some notation.

Notation If k is a node of an A/R graph, let

$ST(k) = node(k) \cdot ps$
$= q_k$ (using the notation in the Appendix).

The symbol := is the assignment symbol; thus $S := S \cup \{k\}$ means "add the node k to the set S."

Consider, for example, steps 3(b) and 4(a) in the Appendix. Using these steps only, we can construct the table shown in Fig. 14. This is an example of a limited-entry decision table (see, for example, [6]). Each column of the table represents a *rule*. If all of the conditions of a rule (those marked with X's) are satisfied, the selected actions are performed, in order, from top to bottom.

Proceeding in this manner, we can construct the entire decision table shown in Figs. 15 and 16. All conditions and actions contained in the Appendix are listed in the corresponding condition and action parts of Figs. 13 and 14. For example, the conditions $k \Rightarrow i\ j$ and $q_j = 2$ of step 3(b,ii) correspond to the conditions in rows 4 and 13 in Fig. 13. Similarly, the action "add j to S" of step 3(b,ii) is listed in row 5 of the action part of the table.

In particular, the rules of Fig. 14 appear as rules 5, 6, 17, and 18 in Figs. 15 and 16. In order to simplify this table, however, the parameter table shown in Fig. 17 is used. This parameter table was constructed by taking advantage of the similarity of the relationships given by the semantic connectives *AND*, *OR*, and *NOT*, and by the *e*-links and *i*-links.

Let us illustrate how the parameter table of Fig. 17 is constructed. Consider conditions 5, 6, and 7 of Fig. 14. These correspond to the condition $ST[j] = par2$ of Fig. 15. For rule 5 of Fig. 15, for example, *par2* is obtained from column 5 of the parameter table (*i.e.*, 0), while for rule 17, *par2* is obtained from column 17 of the parameter table (*i.e.*, 1). There are three special entries in the second row of Fig. 17: columns 9, 24, and 28. They indicate that for these rules the value of *par2* can be either 0 or 1.

Thus, the parameterization mechanism works as follows. When a condition of rule n is being tested, the parameter values are obtained from column n of the parameter table. A condition is considered satisfied (true) if the actual condition (*i.e.*, that obtained after substituting the parameter values) is satisfied. Similarly, when an action of rule n is performed, the parameter values are obtained from column n of the parameter table.

Decision tables are helpful in understanding the logic of algorithms and in the actual implementation of the algorithms. A number of techniques exist for translating decision tables into efficient programs (see, for example, [7]). Furthermore, the use of parameters reduces the amount of code produced in much the same way as the use of subroutines with parameters in a program reduces the amount of code that would otherwise be necessary.

CONDITIONS	Rule No. 1 . . . 5 . . . . 10 . . . . 15 . . . . 20 . . . . 25 . . . 29
$ST[k] = par1$	X X X X X X X X X X X X X X X X X X X X X X X X X X X X X
$k \Rightarrow e\ j$	X X
$j \Rightarrow e\ k$	X X
$k \Rightarrow i\ j$	X X
$j \Rightarrow i\ k$	X X
$k \Rightarrow AND\ j$	X X X X X
$j \Rightarrow AND\ k$	X X
$k \Rightarrow OR\ j$	X X X X X
$j \Rightarrow OR\ k$	X X
$k \Rightarrow NOT\ j$	X X X X X
$j \Rightarrow NOT\ k$	X X
$ST[j] = par2$	X X X X X X X X X X X X X X X X
$ST[j] = 2$	X X X X X X X X X X X X X
$ST[i] = ST[k]\ \forall i \in PS(j)$	X X X X X X
If $\exists m \in PS(j) \mid ST[m] = 2$ and $\forall i \in PS(j)$: $ST[i] = par3$ for $i \neq m$	X X X X X X

Figure 15 Parameterized decision table (condition part) for Algorithm A.

ACTIONS	Rule No. 1 . . . 5 . . . . 10 . . . . 15 . . . . 20 . . . . 25 . . . 29
$ST[j] := par4$	X X X X X X X X X X X X X X X X
$ST[m] := \neg par3$	X X X X X
$ST[m] := \lvert par5 - ST[j] \rvert$	X X X X
$\forall i \in PS(k) \mid ST[i] = 2$: $ST[i] := par6$	X X X X
$S := S \cup \{j\}$	X X X X X X X X X X X X X X X X
$S := S \cup \{m\}$	X X X X X X X X X X
$S := S \cup \{i \in PS(k) \mid ST[i] = 2\}$	X X X X
$ST := \varnothing$ (error)	X X X X X X X X X X X X X X X X

Figure 16 Parameterized decision table (action part) for Algorithm A.

PARAMETERS	Rule No. 1 . . . 5 . . . . 10 . . . . 15 . . . . 20 . . . . 25 . . . 29
$par1$	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$par2$	1 1 0 0 1 0 1 1 1 1 1 0 1 0 1
$par3$	1 0 1 0 1 0
$par4$	0 0 1 1 1 0 0 0 0 1
$par5$	0 1 0 1
$par6$	1 0 0

Figure 17 Parameter table for decision table.

Table 1 Processing **let** rules in different orders.

Variable	*Old value*	*New value*	*New value*	*New value*
x	0	3	3	3
y	5	8	8	8
z	5	11	11	8
v	10	10	19	16
				order is 3-2-1
			order is 2-3-1	
		order is 1-2-3		

7. Determining the order of processing

As previously mentioned, an item has a value and certain properties which affect its behavior and appearance during the execution of the program. The analysis techniques described in the previous sections dealt with item properties. Thus, for example, we were interested in the fact that selecting an item (*i.e.*, assigning *any* value to it) could lead to an inconsistency.

In this section, we consider a more general type of rule, the **let** rule, which is used to express conditions under which the value (and properties) of a variable should be changed by the system. We sketch here the syntax and semantics of **let** rules, as given in [2].

Syntax (general form)

let ⟨*variable*⟩ **be** ⟨*expression*⟩ [**if** ⟨*condition*⟩]

Semantics The conventional rules about matching types of ⟨*expression*⟩ and ⟨*variable*⟩ in an assignment apply. After each response by the user, all **let** rules are processed (in an unspecified order and perhaps several times each) as follows: if a ⟨*condition*⟩ of a **let** rule is *true*, then the ⟨*expression*⟩ is evaluated and assigned to the ⟨*variable*⟩. After processing all the **let** rules, for any rule with ⟨*condition*⟩ *true*, the ⟨*variable*⟩ will have the value of the ⟨*expression*⟩.

With the above semantics, it may be difficult to implement **let** rules efficiently. Processing of the rules in different orders can produce different results, the processing may not terminate, and there may be inconsistencies that cannot be resolved. Consider, for example, the following set of rules.

Example 1

1. **let** v **be** $y + z$;
2. **let** y **be** $x + 5$;
3. **let** z **be** $x + y$

and suppose that the current values of x, y, z, and v are 0, 5, 5, and 10, respectively. If the value of x is changed by the user to 3, new values may be computed for y, z, and v, as shown in Table 1.

In this example, each rule has been evaluated once, and the results are different, depending on the order of processing. However, successive iterations yield, eventually, the same result, as shown in Table 2.

Thus, perhaps we should continue processing the rules as long as the values of variables change. But then the problem of termination arises, as in the following example.

Example 2

1. **let** x **be** $w - y$;
2. **let** y **be** $x - 5$

and suppose that the current values of x, y, and w are 6, 1, and 7, respectively. If the value of w is changed to 9 and the order of evaluation is assumed to be 1-2, successive iterations will produce the values shown in Table 3. In this case, processing of the rules does not terminate.

- **Let** *rules viewed as a production system*

Consider the following set of **let** rules:

let v_1 **be** *expr1* **if** *cond1*;
let v_2 **be** *expr2* **if** *cond2*;
let v_3 **be** *expr3* **if** *cond3*

There is a similarity between **let** rules and the nonprocedural formalism of production systems, which should be mentioned. In the notation used by Newell and Simon [9] for this formalism, we can write the above rules as follows:

cond1 $\rightarrow v_1 :=$ *expr1*
cond2 $\rightarrow v_2 :=$ *expr2*
cond3 $\rightarrow v_3 :=$ *expr3*

Two problems arise: (1) The order of processing of the rules is not known, and (2) we want to evaluate the rules only when necessary. As the rules are processed, we need to keep track of variables that change. Thus we rewrite the above productions as follows:

cond1 **and** *cond1*$'$ $\rightarrow v_1 :=$ *expr1*; *action1*
cond2 **and** *cond2*$'$ $\rightarrow v_2 :=$ *expr2*; *action2*
cond3 **and** *cond3*$'$ $\rightarrow v_3 :=$ *expr3*; *action3*

where

cond i$'$ ≡ any of the values of the variables in *expr i* changed,
action i ≡ indicate that v_i has changed (where $1 \leq i \leq 3$).

Processing the rules as in a true production system would cause problems, as we encountered above. We wish, therefore, to impose restrictions on **let** rules that would make it possible to determine the order in which the above productions should be written. This is done in the section that follows. With these restrictions, we also gain one further advantage over a production system formalism: the rules can be evaluated in one pass without the need for repeated processing from the beginning.

• *Restrictions on* **let** *rules*

The problem is to establish sufficient conditions to ensure termination of processing and uniqueness of the computed values. Moreover, since behavior rules can be written by the programmer in any order, the compiler must determine the order of processing of **let** rules.

In the discussion that follows, the ⟨*variable*⟩ specified on the left side of a **let** rule is called an *output* variable, while the variables contained in the ⟨*expression*⟩ are called *input* variables.

We impose the following restrictions on **let** rules:

1. A variable cannot appear as an output variable in more than one rule.
2. A variable cannot appear as both input and output in the same rule.
3. There are no side effects resulting from the evaluation of the ⟨*expression*⟩, that is, only the value of the output variable can change after processing of the rule.

The first restriction is not as severe as it may appear. If the user specifies the two rules

let *x* **be** *expr1* **if** *cond1*;
let *x* **be** *expr2* **if** *cond2*

the compiler could transform these into an equivalent rule of the form

let *x* **be** (**if** *cond1* **then** *expr1* **else** *expr2*) **if** (*cond1* **or** *cond2*)

assuming that the conditions are mutually exclusive.

In addition, we impose the following restriction on the processing of **let** rules by the compiler:

4. The rule that assigns a value to a variable must be processed before any rule that uses that value.

In example 1, the proper order of processing is 2-3-1 because, in this order, the new value of each variable is computed before the value is used in other rules. On the other hand, restriction 4 is not obeyed in example 2. Restrictions 2 and 4 guarantee that processing of the **let** rules terminates.

Table 2 Iterative evaluation of **let** rules.

Variable	*Values*			*Variable*	*Values*			*Variable*	*Values*		
x	0	3	3	x	0	3	3	x	0	3	3
y	5	8	8	y	5	8	8	y	5	8	8
z	5	11	11	z	5	11	11	z	5	8	11
v	10	10	19	v	10	19	19	v	10	16	19
order is 1-2-3				order is 2-3-1				order is 3-2-1			

Table 3 Iterative evaluation of **let** rules.

Variables	*Old values*	*New values*				
x	6	8	6	8	6	
y	1	3	1	3	1	···
w	7	9	9	9	9	

• *Internal representation of* **let** *rules*

The following definition of *dependence* is adapted from Tesler and Enea [10], who suggest a compile-time analysis to determine which statements in a program can be executed concurrently.

Definition If A is the output variable and B is an input variable of a **let** rule, then A is said to be *directly dependent* on B, written $A\ dep\ B$. If the rules are numbered 1, 2, ···, and the dependency relation is given by rule k, we can also write $A\ dep(k)\ B$.

Definition If A is an output variable and B is an input variable, then the relation dep^+ is defined recursively as follows:

$A\ dep^+\ B$ if either $A\ dep\ B$ or $\exists\ C$ such that $A\ dep\ C$ and $C\ dep^+\ B$.

From these definitions and from restrictions 2 and 4 given above, it is easy to verify that the relation dep^+ is

(a) Transitive: if $A\ dep^+\ B$ and $B\ dep^+\ C$, then $A\ dep^+\ C$.
(b) Antisymmetric: if $A\ dep^+\ B$ then $\neg B\ dep^+\ A$.
(c) Irreflexive: $\neg A\ dep^+\ A$.

We now represent dependency relations by means of a directed graph $\mathcal{C} = (V,R)$ of nodes $v_i \in V$ and edges $r_i \in R$ in which

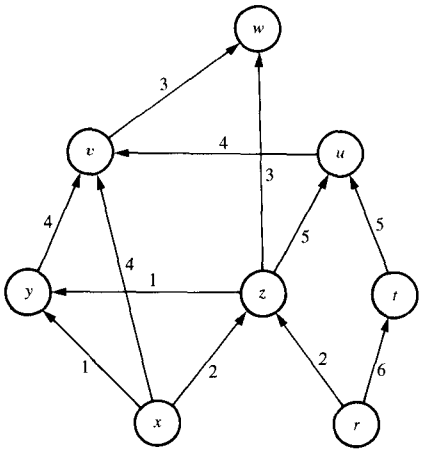

Figure 18 Example of dependency graph.

1. The finite set of nodes V represents the set of input and output variables appearing in the set of **let** rules represented by R.
2. There exists an edge $k \in R$ in $\mathscr{C}$ from node B to node A if and only if A *dep*(k) B.

Such a graph is called a *dependency graph*.

From the definition of dep^+ above and properties (b) and (c) it follows that

1. $A\ dep^+\ B$ if and only if there exists a path in the dependency graph from B to A.
2. The dependency graph contains no cycles.

As an example, consider the following rules (the asterisk represents the multiplication operator):

1. **let** y **be** $x + 2 * z$
2. **let** z **be** $x + r$
3. **let** w **be** $v - z$
4. **let** v **be** $x + 3 * y - 2 * u$
5. **let** u **be** $2 * z + t$
6. **let** t **be** $r - 5$

These rules can be depicted by the dependency graph shown in Fig. 18. Each edge of the graph is labeled with the number of the rule that gives the dependency relation.

As in the case of the A/R graph, we associate a *possible status* property with each node of the dependency graph. However, in this case the possible status ps has the following meaning:

$z \cdot ps = 1$	if a new value has been assigned to z or if the old value of z can be used,
$z \cdot ps = 0$	otherwise.

Now, suppose that n is a rule such that y *dep*(n) x, and x is assigned a new value (either by the user or as a result of processing a **let** rule). Then, rule n can be processed to produce a value for y provided that

$z \cdot ps = 1$ for all $z \neq x$ such that y *dep*(n) z and $z\ dep^+ x$.

This says that to proceed along an edge of the dependency graph and obtain a new value for an output variable, the value of each input variable in the corresponding rule must be available: either the old value can be used or a new value has been assigned.

To illustrate, suppose that in Fig. 18 a new value has been assigned to x. Thus we set $x \cdot ps = 1$. Since r does not depend on anything, we can also set $r \cdot ps = 1$. Before a value for y is computed using rule 1, a new value must be computed for z. But rule 2 can be processed because $x \cdot ps = 1$ and $r \cdot ps = 1$.

By properties (a), (b), and (c) above, the relation dep^+ produces a partial ordering on the nodes of the graph $\mathscr{C}$. We assume the existence of a topological sorting routine, such as that given by Knuth [11]. This routine would give us an initial evaluation order for a given set of rules. However, when the value of a variable changes during the execution of a program, the values of other variables may need to be recomputed. To determine these, we define a new function *recomp*.

Definition If A is an output variable of a set of rules R, then

$recomp(A) = \{x \mid x\ dep^+ A\}$.

The function *recomp* can be extended to a set of variables in the obvious manner.

Definition If A and B are output variables, then

$recomp(A,B) = recomp(A) \cup recomp(B)$.

It is now a simple matter to give an algorithm for determining the order of processing of a set of rules.

Algorithm C Let $x_1, x_2, \cdots, x_k$ be variables whose values have changed. An ordered set $R' = \{r'_1, r'_2, \cdots\}$ of rule numbers is produced, which indicates that rule r'_i must be processed before r'_j if $i < j$.

Step 1 (Initialization.)
Let S be an ordered set containing all of the nodes (variables) of $\mathscr{C}$ after a topological sorting. Let $R = \{r_1, r_2, \cdots, r_m\}$ be the corresponding rules of S, *i.e.*, the initial ordering of the rules. Let $R' = \varnothing$.

Step 2 (Find the variables that require recomputation.)
Let $S' = recomp(x_1, x_2, \cdots, x_k)$.

Step 3 (Select the rules.)
Form the ordered set R' by selecting from R those rules whose output variables appear in S'. That is, for $i = 1, \cdots, m$: If $A\ dep(r_i)\ v$ for some v such that $A \in S'$, then append r_i to R'.

An example
Consider again the earlier example for which the dependency graph is shown in Fig. 18. Figure 19 shows the nodes (variables) of $\mathscr{C}$ after a topological sorting. Thus $S = \{r,t,x,z,y,u,v,w\}$ and $R = \{6,2,1,5,4,3\}$. From Fig. 19 (actually, from the internal representation used in the topological sorting routine), we compute

$$
\begin{aligned}
recomp(r) &= \{t,u,v,w,z,y\}\\
recomp(t) &= \{u,v,w\}\\
recomp(x) &= \{z,y,v,w,u\}\\
recomp(z) &= \{y,v,w,u\}\\
recomp(y) &= \{v,w\}\\
recomp(u) &= \{v,w\}\\
recomp(v) &= \{w\}\\
recomp(w) &= \varnothing
\end{aligned}
$$

Now, suppose that new values have been assigned to x and u. We compute

$$S' = recomp(x,u) = \{z,y,v,w,u\}$$

and from inspection of the rules given above we find that $R' = \{2,1,5,4,3\}$.

Essentially, what Algorithm C is doing is again a topological sorting of the subset S' of the original set of nodes S of $\mathscr{C}$. We know that dep^+ is also a partial ordering on S' (or for that matter, any subset of S). The advantage of using Algorithm C instead of picking the rules in the order given by the original sorting routine is that some of the rules need not be evaluated. Note, for example, that rule 6 does not appear in the final ordered set R' in the above example.

The order of processing of the rules is not unique. In the above example, the rules can also be processed in the order 2-5-1-4-3. However, the fact that the new value of a variable is always computed before it is used in other rules guarantees that the computed values obtained by processing the rules in any order given by Algorithm C are unique.

- *Implementation of Algorithm C*

The dependency graph $\mathscr{C} = (V,R)$ can be represented by an $n \times n$ Boolean matrix M in which $M_{ij} = 1$ if and only if $v_j\ dep\ v_i$, where $v_i,v_j \in V$. Dependency relations dep^+ are then given by the Boolean matrices M, $M^2 = M \times M$, $M^3 = M^2 \times M$, $\cdots$, M^{n-1}, as follows:

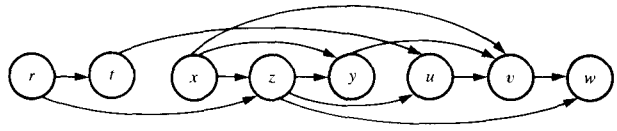

Figure 19 The ordering of the nodes of $\mathscr{C}$ of Fig. 18 after a topological sorting.

$M^k_{ij} = 1$ if and only if there exists a path of length k from v_i to v_j.

The matrices M, M^2, M^3, etc., can be used at compile-time to ensure that the dependency graph has no cycles. The graph has no cycles if and only if there exists an integer $q \leq n$ such that $M^p = 0$ for all $p \geq q$.

8. An implementation

The compile-time analysis techniques described in Section 4 have been implemented as a set of APL functions. The implementation of Algorithm A uses the decision table approach suggested in Section 6.

In this implementation, the possible status property and the node sets used in Algorithms A and B were easily represented and manipulated by means of APL vectors and operators.

9. Summary

In this paper we described compile-time techniques for analyzing user-computer interactions, as well as relationships and dependencies among items of data, that occur during the execution of interactive applications. These kinds of compile-time analysis techniques are necessary to construct efficient compilers for languages in which such interactions and data item relationships and dependencies are described by nonprocedural behavior rules. The practical value of using nonprocedural descriptions is that they simplify the programming of interactive applications.

We have considered in this paper some of the problems that a compiler designer faces when implementing nonprocedural or declarative languages—the kind of language in which the programmer asserts things about the structure of data, without explicit specification of sequencing. Suggestions for other kinds of nonprocedural languages have appeared in the literature. For example, in the language proposed by Homer [12], there are several types of statements (*e.g.*, assignment, READ, WRITE, etc.) Values for variables become available as they are computed or introduced by READ statements, and a statement

is processed when all the input variables in a statement have values. Another example is the ABSYS1 language and compiler described by Foster and Elcock [13].

A related area of research is the design of data flow machines [14] for implementing nonprocedural languages. The aim is to design these machines using a computer architecture based on data flow models. The languages under consideration usually have the single-assignment property mentioned in Section 7, *i.e.*, no variable can appear as an output variable in more than one statement.

Acknowledgment

The author wishes to acknowledge the valuable suggestions given by an anonymous referee, which greatly improved this paper. In particular, this person pointed out the connection with Petri nets described in Section 5 and also provided other suggestions which considerably simplified Algorithm C.

Appendix: Determination of dependent nodes

The following is an expansion of step 2 of Algorithm A to determine node dependencies implied by a given set of nodes and corresponding possible status property.

Let S be the set of nodes to be considered, $S \subseteq N$, and q the corresponding possible status property of N.

Step 1 (Any more nodes?) If $S = \varnothing$ or $q = \varnothing$, we are finished; q is the new possible status property.

Step 2 (Examine each node of S.) Let k be some node of S. If $q_k = 2$, go to step 5.

Case q_k (Execute step 3 or 4, depending on the value of q_k.)

Step 3 (Case $q_k = 1$. Examine edges into and out of current node.)

(a) (Exclusion)

(i) If $k \Rightarrow_e j$ or $j \Rightarrow_e k$, and $q_j = 1$, then S is inconsistent with respect to q; go to step 6.

(ii) If $k \Rightarrow_e j$ or $j \Rightarrow_e k$, and $q_j = 2$, then set $q_j = 0$; add j to S, that is, set $S = S \cup \{j\}$.

(b) (Implication)

(i) If $k \Rightarrow_i j$ and $q_j = 0$, then S is inconsistent with respect to q; go to step 6.

(ii) If $k \Rightarrow_i j$ and $q_j = 2$, then set $q_j = 1$; add j to S.

(c) (*AND* connective) If $k \Rightarrow_{AND} j$:

Recall that $PS(j)$ is the predecessor set of j.

(i) If $q_j = 2$ and $q_i = 1$ for all $i \in PS(j)$, then set $q_j = 1$; add j to S.

(ii) If $q_j = 0$ and $q_i = 1$ for all $i \in PS(j)$, then S is inconsistent; go to step 6.

(iii) If $q_j \neq 2$ and if $\exists m \in PS(j)$, such that $q_m = 2$ and $q_i = 1$ for all $i \in PS(j)$ such that $i \neq m$, then set $q_m = q_j$; add m to S.

If $i \Rightarrow_{AND} k$:

(iv) Set $q_i = 1$ for all $i \in PS(k)$ such that $q_i = 2$. Set $S = S \cup \{i \in PS(k) \mid q_i = 2\}$.

(d) (*OR* connective) If $k \Rightarrow_{OR} j$:

(i) If $q_j = 2$, then set $q_j = 1$; add j to S.

(ii) If $q_j = 0$, then S is inconsistent; go to step 6.

If $i \Rightarrow_{OR} k$:

(iii) If $\exists m \in PS(k)$ such that $q_m = 2$ and $q_i = 0$ for all $i \in PS(k)$ such that $i \neq m$, then set $q_m = 1$; add m to S.

(e) (*NOT* connective) If $k \Rightarrow_{NOT} j$:

(i) If $q_j = 2$, then set $q_j = 0$; add j to S.

(ii) If $q_j = 1$, then S is inconsistent; go to step 6.

If $i \Rightarrow_{NOT} k$:

(iii) Set $q_i = 0$ for all $i \in PS(k)$ such that $q_i = 2$. Set $S = S \cup \{i \in PS(k) \mid q_i = 2\}$.

Step 4 (Case $q_k = 0$. Examine edges into and out of current node.)

(a) (Implication)

(i) If $j \Rightarrow_i k$ and $q_j = 1$, then S is inconsistent; go to step 6.

(ii) If $j \Rightarrow_i k$ and $q_j = 2$, then set $q_j = 0$; add j to S.

(b) (*AND* connective) If $k \Rightarrow_{AND} j$:

(i) If $q_j = 2$, then set $q_j = 0$; add j to S.

(ii) If $q_j = 1$, then S is inconsistent; go to step 6.

If $i \Rightarrow_{AND} k$:

(iii) If $\exists m \in PS(k)$ such that $q_m = 2$ and $q_i = 1$ for all $i \in PS(k)$ such that $i \neq m$, then set $q_m = 0$; add m to S.

(c) (*OR* connective) If $k \Rightarrow_{OR} j$:

(i) If $q_j = 2$ and $q_i = 0$ for all $i \in PS(j)$, then set $q_j = 0$; add j to S.

(ii) If $q_j = 1$ and $q_i = 0$ for all $i \in PS(j)$, then S is inconsistent; go to step 6.

(iii) If $q_j \neq 2$ and if $\exists m \in PS(j)$ such that $q_m = 2$ and $q_i = 0$ for all $i \in PS(j)$ such that $i \neq m$, then set $q_m = q_j$; add m to S.

If $i \Rightarrow_{OR} k$:

(i) Set $q_i = 0$ for all $i \in PS(k)$ such that $q_i = 2$. Set $S = S \cup \{i \in PS(k) \mid q_i = 2\}$.

(d) (*NOT* connective.) If $k \Rightarrow_{NOT} j$:

(i) If $q_j = 2$ and $q_i = 0$ for all $i \in PS(j)$, then set $q_j = 1$. Add j to S.

(ii) If $q_j = 0$ and $q_i = 0$ for all $i \in PS(j)$, then S is inconsistent; go to step 6.

(iii) If $q_j \neq 2$ and if $\exists m \in PS(j)$ such that $q_m = 2$ and $q_i = 0$ for all $i \in PS(j)$ such that $i \neq$

m, then set $q_m = \neg q_j$; add m to S.

If $i \Rightarrow_{NOT} k$:

(iv) If $\exists m \in PS(k)$ such that $q_m = 2$ and $q_i = 0$ for all $i \in PS(k)$ such that $i \neq m$, then set $q_m = 1$; add m to S.

Step 5 Remove node k from set S. Go to step 1.

Step 6 Set $q = \varnothing$.

References

1. J. Martin, *Design of Man-Computer Dialogues*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.
2. J. M. Lafuente and D. Gries, "Language Facilities for Programming User-Computer Dialogues," *IBM J. Res. Develop.* **22,** 145 (1978).
3. J. M. Lafuente, "The Specification of Data-Directed Interactive User-Computer Dialogues," Ph.D. Thesis, Cornell University, Ithaca, NY, 1977.
4. W. R. Elmendorf, "Cause-Effect Graphs in Functional Testing," *Technical Report No. TR00.2487*, IBM Corporation, 1973.
5. J. M. Galey, R. E. Norby, and J. P. Roth, "Techniques for the Diagnosis of Switching Circuit Failures," *IEEE Trans. Commun. Electron.* **83,** 509 (1964).
6. *ACM SIGPLAN Notices* **6,** Special Issue on Decision Tables (1971).
7. I. W. Pooch, "Translation of Decision Tables," *Computing Surv.* **6,** 125 (1974).
8. J. L. Peterson, "Petri Nets," *Computing Surv.* **9,** 223–252 (1977).
9. A. Newell and H. A. Simon, *Human Problem Solving*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.
10. L. G. Tesler and H. J. Enea, "A Language Design for Concurrent Processes," *AFIPS Conf. Proc.* **32,** 403 (1968).
11. D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1969, pp. 258–265.
12. E. D. Homer, "An Algorithm for Selecting and Sequencing Statements as a Basis for a Problem-Oriented Programming System," *Proceedings of the 21st ACM National Conference*, New York, 1966, p. 305.
13. J. M. Foster and E. W. Elcock, "ABSYS 1: An Incremental Compiler for Assertions: An Introduction," *Machine Intelligence* **4,** 423 (1969).
14. "Workshop on Data Flow Computer and Program Organization," D. P. Misumas, Ed., *ACM Computer Architecture News*, Vol. 6, 1977, pp. 11–13.

Received December 5, 1979; revised June 2, 1980

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.