Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

Expressing Event Abstractions in a Debugging Environment

B. Lazzerini, L. Lopriore

Nota interna B4-39
Dicembre 1987

EXPRESSING EVENT ABSTRACTIONS IN A DEBUGGING ENVIRONMENT

B. Lazzerini

Istituto di Elettronica e Telecomunicazioni
Facolta` di Ingegneria, Universita` di Pisa
Via Diotisalvi, 2 - 56100 Pisa - Italy

L. Lopriore

Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
Via Santa Maria, 46 - 56100 Pisa - Italy

Abstract - A debugging environment is presented, based on the event-action model for interaction between the debugging system and the program being debugged (target program). Events are defined at two different levels of abstraction. At the lower level, we have the simple events. These events are expressed in terms of the values of the program-defined entities, and the values of a set of variables, called instruction address (_ia) variables. An _ia variable is associated by the debugging system to each program block. The _ia variable for a given block is accessed when a statement of that block is executed, and its value is replaced by the label of that statement. At the higher level, we have the compound events, expressed in terms of simple events and a set of operators, the instantaneous/deferring (_id) operators. At any given time, an instantaneous operator will produce a result which depends on the value(s) of its factor(s) at that time, whereas a deferring operator produces a result which depends not only on the value(s) assumed at that time, but also on the values assumed from another given time, called the origin of the operator.

On the occurrence of a given event, the actions connected to that event will be performed. Possible actions can be moving the origins of the deferring operators, and generating traps. A trace trap displays a portion of the program state. A break trap returns control to the programmer at the console.

Instruction address variables and instantaneous/deferring operators are powerful mechanisms for monitoring the activity of the target program. They make it possible to construct event abstractions in terms of the path followed by the flow of control (flow history) and the sequence of the program states (state history). Rather than offering a fixed set of special-purpose tools, tailored to a specific program debugging approach, the resulting environment adequately supports different debugging techniques, and offers the user a considerable degree of control over the debugging experiment.

Index Terms - Action, break trap, debugging, event, flow history, state history, trace trap.

This work has been supported in part by the Italian Ministry of Education, and in part by the research contract of Selenia, Industrie Elettroniche Associate, S.p.A., Rome, Italy, and Consiglio Nazionale delle Ricerche, Pisa, Italy.

I. INTRODUCTION

Debugging is the process of diagnosing and removing program errors [36]. After a program has been written, it is executed with carefully selected test data. If the program does not behave as expected, an error is suspected. The results of the execution are analyzed, and the cause of the problem is located. A correction is proposed, and the test is repeated [14], [29].

Debugging is a time-consuming activity. Ad-hoc tools and facilities help the programmer in this phase of program development [8], [31]. These tools support different debugging techniques. An execution monitor observes the execution of the program being debugged (target program). Filtering actions select the information which interests the actual debugging experiment. The results of these filtering actions are displayed at the debugging console, or, alternatively, are recorded on some storage media for later analysis. Path tracing is the process of recording the control flow at a suitable level of granularity, e.g., branches, basic blocks [20], and subroutine invocations [22]. State tracing is the process of recording the evolution of an aspect of the program state. This aspect is expressed in terms of the values assumed by a subset of the program-defined entities (target entities). An interactive debugger makes it possible to interact with the program during execution [30]. A breakpoint is an interrupt in execution, causing control to return to the programmer at the console. He can inspect the program state by displaying the value of the target entities, and can affect the execution of the program if he so wishes by, for instance, altering the value of the target variables. Single step execution is the ability to return control to the programmer on execution of each target statement. This is useful especially when it is not clear where to place breakpoints [10].

We will refer to the event-action model of interactions between the debugging system and the target program [6], [13]. In this model, an event is a condition on the program state, and an action is an operation performed by

the debugging system. Actions can involve not only the state of the target program (e.g., by displaying the value of a target variable), but also the state of the debugging system itself (as is required, for instance, to modify the layout of the debugging experiment dynamically). On the occurrence of a given event, the debugging system performs those actions connected to that event. An ad-hoc set of debugging commands allows the user to define events and actions, and to control the connections between the events and the actions.

The set of events depends on both the debugging technique(s) and the programming language(s) supported by the debugging environment. For instance, the events for a program written in a sequential high-level language involve conditions on the values of the target variables; events for a concurrent program are also relevant for process interaction and synchronization [14].

A simple event is an event defined at the lowest level of abstraction, in terms of the elementary behavior of the target program. Simple events can be grouped to express other events at higher levels of abstraction (compound events). The set of simple events and the mechanisms to define event abstractions are peculiar aspects of a debugging environment. In one approach, a large set of simple events is supported by weak tools to build up event abstractions. In this approach, the conditions on the state of the target program are expressed by simple events at a low level of granularity. An alternative approach is based on a few simple events which can be compounded by means of a collection of sophisticated mechanisms.

Let us refer, for instance, to concurrent program debugging. In a debugging environment in the first approach, the simple events include the execution of primitives for process synchronization and communication, e.g. operations on semaphores, for a language using shared variables for interprocess communication, and the transmission and reception of a message, for a message-oriented language [2]. In such an environment, a compound event

is a sequence of simple events, and occurs when all the events in the sequence have occurred in the given order. In an environment in the second approach, the simple events monitor the behavior of a single process, and interactions between processes are monitored by compound events. These can involve conditions on both the paths followed by the control flows (flow histories) and the sequences of the states (state histories) of the interacting processes.

We believe that an important requisite for an effective debugging environment is a simple and powerful set of primitives which can be used to build up specialized tools. In this way, the environment offers the programmer a considerable degree of control over the debugging experiment, and allows him to tailor the experiment to the intended debugging technique, rather than supporting a specific approach to program debugging. This paper presents a debugging environment featuring a small, powerful set of simple events. They can involve conditions both on the values of the target entities and on the values of a set of variables, called instruction address (_ia) variables. One such variable is associated by the debugging system to each block of the target program. The _ia variable relevant to a given block is accessed at the execution of a statement in that block, and its value is replaced with the label of that statement. A compound event is an expression involving simple events and a set of operators, the instantaneous/deferring (_id) operators. The _id operators allow the programmer to monitor the activity of the target program, by keeping track of the evolution of the values assumed by the simple events. They make it possible to express conditions in terms of both the flow and the state histories.

Section II contains the rules for expressing simple and compound events. The _id operators are presented and their properties are illustrated. Section III describes the rules used to denote the target entities. Instruction address variables are introduced, and their utilization in the definition of

simple events is detailed. The interaction of the programmer with the debugging system is considered in Sections IV and V, where the command set is analysed. Section IV gives a detailed description of the commands for defining events and causing actions, and Section V examines other important aspects, such as commands for displaying and modifying the values of the target entities. Section VI considers a wide set of situations, which are likely to occur in practical debugging experiments. This section is organized as a sequence of problems. For each problem, a solution is outlined, and the commands for implementing that solution are shown. Finally, Section VII discusses the applications of the _id operators and the _ia variables. The proposed debugging environment is evaluated, with particular respect to the support given to the outstanding debugging techniques.

II. EVENTS AND INSTANTANEOUS/DEFERRING OPERATORS

A. Simple Events

An event is an expression whose value depends on the state of the program being executed (target program). Events can be simple and compound. A simple event is an expression consisting of a relational operator and two factors. A factor is a target variable, the label of a target statement, a numeric literal, or a string literal. The relational operators are the equality operator ==, the inequality operator <>, and the ordering operators < (less than), <= (less than or equal to), >= (greater than or equal to), > (greater than). The equality and the inequality operators are defined for factors of any type. The ordering operators are defined for factors of any scalar type, for statement labels and for character arrays. The definition of a simple event consists of the specification of the relation expressing that event, enclosed in round brackets. At time t, a given simple event is true if the relation expressing that event is true at that time.

B. Compound Events

A compound event is an expression consisting of _id operators and simple events. The _id operators are the unary operators ~ (instantaneous negation) and | (deferrer), and the binary operators && (instantaneous conjunction), & (deferred conjunction), || (instantaneous disjunction), | (deferred disjunction). The _id operators are partitioned into three precedence classes. One class includes unary operators. The other two classes include the conjunction operators and the disjunction operators, respectively. Classes have been given in order of decreasing precedence.

A compound event is an instantaneous event if it is expressed in terms of instantaneous operators, whereas a deferred event is a compound event whose specification includes one or more deferring operators. The meaning of all the _id operators is given below.

Definition 2.1. The unary operator ~ denotes the instantaneous negation of an event. The instantaneous negation ~E of event E is true at time t if event E is false at that time.

Definition 2.2. The unary operator | denotes the deferrer of an event. This is a function of a time t_0, called the origin of the deferrer. The deferrer |E of event E is true at time t if there exists a time t* in the interval $[t_0,t]$ such that the event is true at t*.

Definition 2.3. The binary operator && denotes the instantaneous conjunction of two events. The instantaneous conjunction E_1 && E_2 of events E_1 and E_2 is true at time t if both E_1 and E_2 are true at t.

Definition 2.4. The binary operator & denotes the deferred conjunction of two events. This is a function of a time t_0, called the origin of the conjunction. The deferred conjunction E_1 & E_2 of events E_1 and E_2 is true at time t if two times t_1 and t_2 exist in the interval $[t_0,t]$, such that E_1 is true at t_1 and E_2 is true at t_2.

Definition 2.5. The || operator denotes the instantaneous disjunction of two events. The instantaneous disjunction E_1 || E_2 of events E_1 and E_2 is true at time t if either E_1 or E_2 or both are true at that time.

Definition 2.6. The binary operator | denotes the deferred disjunction of two events. This is a function of a time t_0, called the origin of the disjunction. The deferred disjunction E_1 | E_2 of events E_1 and E_2 is true at time t if a time t* exists in the interval $[t_0,t]$ such that either E_1 or E_2 or both are true at t*.

The notations $|^{(t_o)}$ and $\&^{(t_o)}$ evidence the origin of a deferring operator, when this is not otherwise clarified by the context.

Example 2.1:

Let A be a target variable, and let us consider the events which follow:

```
(A > +20)                                   (e1)
|(A > +20)                                  (e2)
(A > -10) && (A < +10)                      (e3)
(A > -10) &  (A < +10)                      (e4)
(A < -20) || (A > +20)                      (e5)
(A < -20) |  (A > +20)                      (e6)
(A < -20) &  (A > +20)                      (e7)
```

(e1) is a simple event, all the others are compound events. (e3) and (e5) are instantaneous events, (e2), (e4), (e6) and (e7) are deferred events. Let t_0 be the origin of all the deferring operators. Supposing that at t_0 A has the value +15, and at t_1, t_2 and t_3 A assumes the values +25, 0 and -25, respectively (Fig. 1), it then follows that (e1) is true in the interval $[t_1, t_2)$; (e2) is true from t_1 onwards; (e3) is true in $[t_2, t_3)$; (e4) is true from t_2 onwards; (e5) is true in $[t_1, t_2)$ and from t_3 onwards; (e6) is true from t_1 onwards; and, finally, (e7) is true from t_3 onwards.

Example 2.2:

As a further example, let us consider the different meanings of events (e8) and (e9) defined in terms of events E1 and E2, as follows:

E1 & E2 (e8)
|(E1 && E2) (e9)

Let t_0 be the origin of the deferring operators involved in these events. Suppose that E1 is true in the interval $[t_1, t_2)$ and from t_4 onwards, and E2 is true in the interval $[t_3, t_5)$ (Fig. 2). It follows that (e8) is true from t_3 onwards, and (e9) is true from t_4 onwards.

C. Properties of the Deferring Operators

Let TRUE and FALSE denote an event which is always true and an event which is always false, respectively. Certain properties are a direct consequence of Definitions 2.1-2.6:

$$E \,\&^{(t_o)}\, TRUE = |^{(t_o)} E \tag{2.1}$$

$$E \,|^{(t_o)}\, FALSE = |^{(t_o)} E \tag{2.2}$$

$$E \,\&^{(t_o)}\, E = |^{(t_o)} E \tag{2.3}$$

$$E \,|^{(t_o)}\, E = |^{(t_o)} E \tag{2.4}$$

$$E_1 \,\&^{(t_o)}\, E_2 = E_2 \,\&^{(t_o)}\, E_1 \tag{2.5}$$

$$E_1 \,|^{(t_o)}\, E_2 = E_2 \,|^{(t_o)}\, E_1 \tag{2.6}$$

$$E_1 \,\&^{(t_o)}\, (E_2 \,\&^{(t_o)}\, E_3) = (E_1 \,\&^{(t_o)}\, E_2) \,\&^{(t_o)}\, E_3 \tag{2.7}$$

$$E_1 \,|^{(t_o)}\, (E_2 \,|^{(t_o)}\, E_3) = (E_1 \,|^{(t_o)}\, E_2) \,|^{(t_o)}\, E_3 \tag{2.8}$$

Relations (2.5), (2.6) and (2.7), (2.8) state that the & and | operators are commutative and associative, respectively.

The relations which follow are useful in transforming and simplifying the specifications of deferred events:

$$|^{(t_o)}(|^{(t_o)}E) = |^{(t_o)}E \qquad (2.9)$$

$$|^{(t_o)}(E_1 \;\&^{(t_o)}\; E_2) = E_1 \;\&^{(t_o)}\; E_2 \qquad (2.10)$$

$$|^{(t_o)}(E_1 \;|^{(t_o)}\; E_2) = E_1 \;|^{(t_o)}\; E_2 \qquad (2.11)$$

Proof of Relation (2.9): Assume that $|^{(t_o)}(|^{(t_o)}E)$ is true at time $t \geq t_0$. By Definition 2.2 there exists a time t* in $[t_0,t]$ such that $|^{(t_o)}E$ is true at t*. Hence there exists a time t' belonging to $[t_0,t^*]$ such that E is true in t'. Thus $|^{(t_o)}E$ is true in t. Now assume that $|^{(t_o)}(|^{(t_o)}E)$ is false at t, then $|^{(t_o)}E$ is false at each t* in $[t_0,t]$. Thus $|^{(t_o)}E$ is false in t.

The proofs of Relations (2.10) and (2.11) are similar to the proof of Relation (2.9), and are left to the reader.

Relations (2.12) and (2.13) below show that the binary deferring operators can be expressed in terms of the binary instantaneous operators and the unary deferrer operator:

$$E_1 \;\&^{(t_o)}\; E_2 = |^{(t_o)}E_1 \;\&\&\; |^{(t_o)}E_2 \qquad (2.12)$$

$$E_1 \;|^{(t_o)}\; E_2 = |^{(t_o)}E_1 \;||\; |^{(t_o)}E_2 \qquad (2.13)$$

Proof of Relation (2.12): Assume that $E_1 \;\&^{(t_o)}\; E_2$ is true at time $t \geq t_0$. By Definition 2.4 two times t_1 and t_2 exist in the interval $[t_0,t]$ such that E_1 is true at t_1 and E_2 is true at t_2. Hence both $|^{(t_o)}E_1$ and $|^{(t_o)}E_2$ are true at t, and the right side of the relation is true. Now assume that $E_1 \;\&^{(t_o)}\; E_2$ is false at t, then either E_1 or E_2 or both are false at every time t* in $[t_0,t]$. Hence either $|^{(t_o)}E_1$ or $|^{(t_o)}E_2$ or both are false at t, and the right side of the relation is false.

The proof of Relation (2.13) is similar to the proof of Relation (2.12), and is left to the reader.

Relations (2.14) and (2.15) below follow directly from Relations (2.10) and (2.12), and from Relations (2.11) and (2.13), respectively:

$$|^{(t_o)}(E_1 \ \&^{(t_o)} \ E_2) = |^{(t_o)}E_1 \ \&\& \ |^{(t_o)}E_2 \qquad (2.14)$$

$$|^{(t_o)}(E_1 \ |^{(t_o)} \ E_2) = |^{(t_o)}E_1 \ || \ |^{(t_o)}E_2 \qquad (2.15)$$

A. Event Evaluation Times

Let S be a simple event. If at time t at least one of the factors in terms of which S is expressed is accessed, then t is an evaluation time for S. Let C be a compound event. If time t is an evaluation time for at least one of the events which compound C, then t is an evaluation time for C.

A simple or compound event occurs at time t if t is an evaluation time for that event, and at t the event is true.

III. TARGET ENTITIES

A. Denoting Target Entities

An outermost block of the target program is denoted by the : symbol followed by the identifier of that block. A block which is not an outermost block is denoted by a path name. This is a sequence of block identifiers separated by the : symbol. The path name of a given block is the path from an outermost block to that block. A path name not starting with the : symbol begins in the the block being executed (current block); this notation is not permitted if the target program is a concurrent program. The debugging system assigns automatic identifiers to nameless blocks. One such identifier consists of the # symbol followed by the order number (textually) of that block in the block containing its definition.

A target variable is denoted by the path name of the block containing the definition of that variable, and the variable identifier. The % symbol acts as a separator between the path name and the variable identifier. If the path name is omitted, the variable is assumed to be defined in the current block.

A target statement is denoted by the path name of the block containing that statement, and the statement label. The $ symbol acts as a separator

between the path name and the label. The debugging system assigns automatic labels to unlabeled statements. One such label is the # symbol followed by the order number (textually) of that statement in the containing block. The ^ symbol denotes the last statement of a block. If the block path name is omitted, the statement is assumed to be contained in the current block.

Example 3.1:

:MAIN denotes the block named MAIN at the outermost level in the target program. :MAIN:BLK denotes the block named BLK and defined within :MAIN. :MAIN:#1 denotes the first block (textually) defined within :MAIN. :MAIN:BLK%COUNT denotes the variable COUNT defined in block :MAIN:BLK. If :MAIN:BLK is the current block, this variable can be denoted by %COUNT. :MAIN$LOOP is the statement labeled LOOP in block :MAIN. :MAIN$#1, :MAIN$#10 and :MAIN$^ are the first, tenth and last statement of block :MAIN. $#1, $#10 and $^ are the first, tenth and last statement of the current block.

B. The Instruction Address Variables

Let the target program be a sequential program, and let B denote a block path name. The debugging system associates a variable, called instruction address (_ia) variable, to each target block. The _ia variable for block B is denoted by B_ia. The values B_ia can assume are the statement labels contained in block B, the automatic labels B$#1, B$#2, ..., B$^ of the statements of B, and the NIL value. The NIL value is the initial value of B_ia. Variable B_ia is accessed each time a statement of block B is executed. Its current value is replaced with the label of this statement. On abandoning the execution of B, the value of B_ia is equal to the label of the last statement executed in B. It should be noted that no value is shared by the _ia variables for different blocks.

Let the target program be a concurrent program, and let SB be the path name of a block shared by processes P1, P2, ..., Pn. For each process Pi, the debugging system associates an _ia variable to SB. This variable is denoted by SB_ia@Pi. The values SB_ia@Pi can assume are the statement labels contained in block SB, the automatic labels SB$#1, SB$#2, ..., SB$^ of the statements of block SB, and the NIL value. The NIL value is the initial value of SB_ia@Pi. SB_ia@Pi is accessed each time a statement of SB is executed by Pi. Its current value is replaced by the label of this statement. On completion of execution of SB, the value of SB_ia@Pi is equal to the label of the last statement executed in SB by Pi.

Finally, in a concurrent program, let LB be the path name of a block local to a single process Pi. A single _ia variable, LB_ia, is associated to LB. This variable is treated in the same way as the _ia variable for a block of a sequential program.

An _ia event is a simple event expressed in terms of an _ia variable and a label in the block of that variable. The path name of the block in the specification of the label can be omitted. It is an error to express a simple event in terms of more than one _ia variable. This constraint applies also to _ia variables for different processes. If the value of the _ia variable involved in a given _ia event is equal to NIL, the value of that _ia event is equal to false.

Example 3.2:

The simple event (B_ia > B$#100) can be written as (B_ia > $#100). This event occurs at execution of each statement of block B following the one-hundredth statement. At any given time, its value is equal to the value of the compound event (B_ia == $#101) || (B_ia == $#102) || ... || (B_ia == $#^).

IV. THE COMMAND SET - TREATING EVENTS

The debugging system executes the target program in a controlled fashion, according to the specifications of the debugging experiment. These specifications are stated by the user, by issuing proper sequences of debugging commands. Some commands make it possible to express events, to move origins, and to cause actions on the occurrence of events. These commands are considered in detail in this section. The discussion is completed in the next section, where other important aspects of the command set are considered.

A. Activating and Deactivating Events

An event declaration associates a simple or compound event with an event identifier. After declaration, the identifier can be included in the specifications of other events. In this case, the identifier is just an abbreviation for the event it represents. The command

event <event identifier> = <event>

declares an event. The association of the event with the given identifier is valid throughout the debugging session.

An event is active if its value is computed concurrently with the execution of the target program, according to the values assumed by the target entities included in the specification of that event. The command

event on <event identifier>

activates the event associated with the given event identifier. If this is a compound event, this command also activates each component event. If this command is issued at time t_0, and the event is deferred, then t_0 becomes the origin of the deferring operators involved in that event. It is an error to declare a compound event such that the values of one or more components depend on the value of that compound event. The command

```
event off <event identifier>
```

deactivates the event associated with the given event identifier.

Let eid, eid1, eid2, ..., eidN denote event identifiers. The command

```
event on eid = <event>
```

is equivalent to the command pair

```
event eid = <event>
event on eid
```

The command

```
event on eid1 eid2 ... eidN
```

is equivalent to the sequence of commands

```
event on eid1
event on eid2
  .
  .
  .
event on eidN
```

The command

```
event off eid1 eid2 ... eidN
```

is expanded similarly.

Example 4.1:

```
event E1 = (A > +10)
event on E2 = (A < -10) & E1
```

The first command associates simple event (A > +10) with identifier E1. The second command activates the simple events (A > +10) and (A < -10), and the compound event (A < -10) & (A > +10). The origin of the deferred conjunction is the time at which the second command is issued.

B. Moving the Origin of an Event

Two commands, **origin** and **origin at**, make it possible to move the origins of the deferring operators included in the definition of the active events. The **origin** command has the form:

origin <event identifier>

Let t_0 be the time at which this command is issued. If the named event is a deferred event, then t_0 becomes the origin of the deferring operators in terms of which that event is expressed. If the named event is a simple event or an instantaneous event, the command has no effect. It is an error to issue an **origin** command involving an event which is not active.

The **origin at** command has the form:

origin <event identifier> **at** <event identifier> ... <event identifier>

The events named in the **at** portion are called the controllers of the command, whereas the event named in the **origin** portion is the destination. Let t* be the time at which this command is issued, and suppose that a controller occurs at time t_0, $t_0 > t^*$. If the destination is a deferred event, then t_0 becomes the origin of the deferring operators involved in that event. At the occurrence of an event which is both a component and a controller of a deferred event, the movement of the origin precedes the evaluation of the deferred event. If the destination is a simple event or an instantaneous event, the command has no effect. It is an error to issue an **origin at** command involving an event which is not active. It is an error to issue an **origin at** command such that the values of one or more controllers depend on the value of the destination.

The command

origin eid1 eid2 ... eidN

is equivalent to the sequence of commands

```
origin eid1
origin eid2
  .
  .
  .
origin eidN
```

Let dst1, dst2, ..., dstM, ctrl1, ctrl2, ..., ctrlP denote active events. The command

```
origin dst1 dst2 ... dstM at ctrl1 ctrl2 ... ctrlP
```

is equivalent to the sequence of commands

```
origin dst1 at ctrl1 ctrl2 ... ctrlP
origin dst2 at ctrl1 ctrl2 ... ctrlP
  .
  .
  .
origin dstM at ctrl1 ctrl2 ... ctrlP
```

C. Traps

A trap is a deviation in the normal flow of the target program which causes control to be temporarily or permanently transferred to the debugging system. A trace trap is a temporary control transfer which causes the debugging system to display a portion of the state of the target program at the console. The condition causing such a trap is expressed in terms of the occurrence of an event by the **trace on** command, which has the form:

trace on <event identifier> **display** <target entity> ... <target entity>

This command causes a trace trap to be generated each time the named event occurs. The trace includes the values of the target entities specified by the **display** portion of the command. In this portion, the $$ literal denotes the target statement being executed at the generation of the trap.

A trace condition is cleared by the **trace off** command, which is as follows:

trace off <event identifier>

After issuing this command, no more trace traps are generated at the

occurrence of the named event.

A break trap is a permanent transfer of control to the debugging system, which consequently enters the break mode. This mode allows the user to issue commands interactively from the console. The condition causing a break trap is expressed in terms of the occurrence of an event by the **break on** command, which has the form:

```
break on <event identifier>
```

A break condition is cleared by the **break off** command, which is as follows:

```
break off <event identifier>
```

The break mode can be abandoned by issuing the **continue** command, which has the form:

```
continue
```

This command returns control to the target program. Execution is resumed at the point where it was suspended by the break trap.

Let trg1, trg2, ..., trgT be target entities. The command

```
trace on eid1 eid2 ... eidN display trg1 trg2 ... trgT
```

stands for the sequence of commands

```
trace on eid1 display trg1 trg2 ... trgT
trace on eid2 display trg1 trg2 ... trgT
  .
  .
  .
trace on eidN display trg1 trg2 ... trgT
```

Expansions of the commands

```
trace off eid1 eid2 ... eidN
break on  eid1 eid2 ... eidN
break off eid1 eid2 ... eidN
```

are easy to imagine, and will not be shown.

V. MORE ON THE COMMAND SET

The command set presented so far must be upgraded by adding commands to display and modify the values of target entities interactively, and to redirect the traces. Other commands help the user to denote the target entities. A discussion of these and other aspects of the command set is presented in this section.

A. Displaying the Value of Target Entities

The **display** command allows the user to inspect the values of the target entities interactively. This command has the form

display <target entity> ... <target entity>

and prints the values of the specified target entities at the console.

Display formats should be congenial to the user [17]. For instance, the value of a scalar variable of a user-defined enumeration type should be printed using the enumeration literals in terms of which that type is defined. In some implementations, the type information is not available for the debugger. One solution is to try to derive the type of a variable from its size, and to solve the type ambiguities by printing the variables in all the appropriate formats [34]. A different approach is to include a format specification in the **display** command. For instance,

display V1!i V2!c

takes variable V1 to be a signed integer, and V2 a single character.

The display format of structured variables such as arrays depends on the type of the components. Of course, it is desirable for character arrays to be printed in the usual string format. Statements should be displayed as they appear in the source listing, with the addition of the automatic labels.

Of course, these considerations apply to the **display** portion of the **trace on** command as well. Furthermore, the user should be allowed to include

comments in a trace, for increased readability. For this purpose, strings should be valid parameters for the **trace on** command, as in the example which follows:

```
trace on EV display "V1: " V1!i
```

B. Modifying the Value of Target Entities

The **assign** command makes it possible to assign a new value to a target variable. This command has the form

assign <target variable> := <expression>

The expression is evaluated and the result is assigned to the named variable. This variable and the value of the expression must be of the same type. The expression may involve the target variables and constants. The implementation may restrict the applicability of this command, for instance, to the variables in the active blocks, or to the most recent activation of each variable [10], or even to the variables in the current block. Furthermore, the implementation may explicitly state that the values of the target variables will be retrieved and changed in memory. If the value of one such variable is stored in a register, the effects of the **assign** command may not be reflected by later use of that target variable [18].

C. Redirecting a Trace

The console can be replaced by a file for trace output. A trace file makes it possible to graphically replay the execution of program portions. Events can be displayed, at reduced speed if desired, without having to run the entire program again from the beginning [26]. This is especially useful in debugging concurrent programs. The result of the execution of a concurrent program depends on process scheduling and timings, and is hard to reproduce. This is true, in particular, in the presence of improper process synchronizations [7], [14]. Further applications of trace post-processing can

be found in the field of program performance evaluation [23].

The specifications of trace redirection can be included in the **trace on** command, as in the example below:

```
trace on EV display V1!i V2!c to TRFILE
```

The output of the trace is placed in the TRFILE file instead of being displayed at the console. The command

```
trace to TRFILE
```

redirects the output of every **trace on** command not featuring an explicit trace redirection. The command

```
trace to CONSOLE
```

resumes the console for trace output.

D. Changing the Current Block

As seen in Section III, if a target entity is defined in the current block, the path name of that block can be omitted, and that entity can be denoted simply by its identifier. If the user needs to access several entities, which are all defined in a block which is not the current block, the **current** command allows temporary modification of the current block. For instance, after issuing the command

```
current :MAIN:BLK
```

the current block will be :MAIN:BLK. The effects of this command cease on resuming execution of the target program.

A refinement (suggested by Holdsworth [21]) is to introduce an ad-hoc notation, e.g. ::, for the block enclosing the current block. For instance, suppose that the current block is :MAIN:BLK. After issuing the command

```
current ::
```

the current block will be :MAIN.

E. Aliasing Path Names

The same identifier may occur in different declarations in the target program. Of course, no ambiguity can occur if a target entity is denoted to the debugger using path names. However, the user has to type a full path name to denote an entity which has not been defined in the current block. An aliasing mechanism makes it possible to introduce a short, easy-to-type alias for a complex path name [21]. This mechanism takes the form of an **alias** command. For instance, after issuing the command

```
alias P :MAIN:BLK
```

P will be an alias for the path name :MAIN:BLK. The command

```
alias all
```

lists all active aliases. The command

```
unalias P
```

removes the named alias. Finally, the command

```
unalias all
```

removes all active aliases.

The **alias** command is a useful complement to the **current** command. In traditional debugging environments, the user seldom needs to issue a command involving entities defined in different blocks [10]. A single exception is perhaps the trace command. As will be shown in the next section, this is not the case as far as applications of deferring operators are concerned.

F. Scope Rules

At any given time in the execution of the target program, only a subset of the target variables will have defined values, as stated by the scope rules of the target language. This is true even in languages, such as FORTRAN, which feature a static binding between the code segment and the activation record

[15]. Even if a given variable has a defined value, accessing that variable when it is not visible is a violation of the scope rules.

On the other hand, limiting the range of possible accesses of the debugging system to the scope rules is not viable. For instance, it could be useful to inspect the internal state of a module not visible from any active block [21]. A solution is to allow full path names to be used to denote the variables not visible from the block being executed. Of course, with path names it is possible to denote every target entity, independently of the scope rules.

G. Other Aspects

The discussion of the command set, contained in this and in the previous section, has been essentially aimed at showing how the _id operators and the _ia variables can be integrated in an interactive debugging environment. We must point out that the commands introduced so far are only a subset of the command set of a practical debugger. We have intentionally omitted considering several features which are not related to the treatment of events. Examples are commands which force a diversion in the flow of control of the target program; mechanisms for saving and then restoring the state of a debugging session, such as are required to span a debugging experiment across two or more sessions; and advanced display facilities, e.g. multiple windows. These features are, however, important aspects of an effective debugging environment [12], [33], [36].

VI. EXAMPLES OF APPLICATIONS

This section considers some important applications of _ia variables and _id operators. A set of problems are proposed which reflect situations which are likely to occur in a practical debugging experiment. For each problem, a solution is outlined, and a sequence of commands to implement that solution is

illustrated.

A. Detecting Unsafe Situations

Problem 6.1: A bounded buffer, defined in the USE_BUFFER block, can hold up to 100 elements. The USE_BUFFER%COUNT variable indicates how many elements of the buffer are in use. Access to the buffer is unsafe when the value of the counter is greater than 90 (buffer nearly full): a) trace each unsafe access; b) generate a break at the first unsafe access and at each subsequent access.

Solution:

a) Event OFLW below will be true when the buffer is nearly full. A trace trap is generated at each occurrence of OFLW. The trace includes both the statement causing the access and the value of the counter, as follows:

```
event on OFLW  = (USE_BUFFER%COUNT > 90)
trace on OFLW display $$ USE_BUFFER%COUNT
```

b) The deferred event DOFLW is defined in terms of the deferrer of OFWL. A break is generated at each occurrence of DOFLW, as follows:

```
event on DOFLW = | OFLW
break on DOFLW
```

The command

```
origin DOFLW
```

suspends the generation of break traps until occurrence of the next situation of near-overflow.

Problem 6.2: Same as Problem 6.1, but a buffer access is unsafe even when the value of the counter is less than 10 (buffer nearly empty).

Solution:

a) Event UFLW below will be true when the buffer is nearly empty. The event UOFLW is defined in terms of the instantaneous disjunction of UFLW and of event OFLW defined in the solution of Problem 6.1. The trace is gathered at the occurrence of UOFLW, as follows:

```
event UFLW  = (USE_BUFFER%COUNT < 10)
event on UOFLW = UFLW || OFLW
trace on UOFLW display $$ USE_BUFFER%COUNT
```

b) The event DUOFLW is defined in terms of the deferred disjunction of UFLW and OFLW. A break is generated at the occurrence of DUOFLW, as follows:

```
event on DUOFLW = UFLW | OFLW
break on DUOFLW
```

B. Control Path

Problem 6.3: Detect when the control path of the target program has passed over the statements of block B labeled L1, L2 and L3:

Solution:

```
event on PATH = (B_ia == $L1) & (B_ia == $L2) & (B_ia == $L3)
break on PATH
```

Problem 6.4: Detect when the control path has passed over statements B$L1 and B$L2, in that order.

Solution:

```
event EL1 = (B_ia == $L1)
event EL2 = (B_ia == $L2)
event on EL1L2 = EL1 & EL2
origin EL1L2 at EL1
break on EL1L2
```

Events EL1 and EL2 will become true on execution of B$L1 and B$L2, respectively. Event EL1L2 is defined in terms of the sequential conjunction of EL1 and EL2. The **origin at** command moves the origin of EL1L2 on the occurrence of EL1, and this prevents EL1L2 from becoming true if EL2 occurs before EL1.

Problem 6.5: Detect when the control path has passed over statements B$L1, B$L2 and B$L3, in that order.

Solution:

```
event EL3 = (B_ia == $L3)
event on EL1L2L3 = EL1L2 & EL3
origin EL1L2L3 at EL1 EL2
break on EL1L2L3
```

Events EL1, EL2 and EL1L2 have been defined in the solution of Problem 6.4. The solutions of Problems 6.4 and 6.5 can be easily extended to treat conditions on control paths expressed in terms of an arbitrarily large number of statements.

C. Tracing the Execution of Program Portions

Problem 6.6: Trace the execution of a procedure P.

Solution:

```
event on IP = (P_ia > $#1) && (P_ia < $^)
trace on IP display $$
```

Problem 6.7: The program fragment between statements B$L1 and B$L2 contains a call to procedure P. We must: a) trace the execution of the procedure when called from inside the fragment; b) trace the execution of both the procedure and the fragment; c) trace the execution of the procedure when called from outside the fragment, after the first execution of the fragment.

Solution:

a)

```
event on TRP = (B_ia > $L1) && (B_ia < $L2) && IP
trace on TRP display $$
```

Event IP has been defined in the solution of Problem 6.6.

b)

```
event on TRPF = (B_ia > $L1) && (B_ia < $L2) || IP
trace on TRPF display $$
```

c)

```
event on IF =  (B_ia == $L1)
event on OF = |(B_ia == $L2)
origin OF at IF
event on TROF = IP && OF
trace on TROF display $$
```

D. Tracing the Values of a Variable

Problem 6.8: Statement B$INIT initializes a variable B%V. Trace each access to B%V before initialization.

Solution:

```
event INV = |(B_ia == $INIT)
event ACCV = (B%V <= MAXV)
event on TRV = ~INV && ACCV
trace on TRV display $$ B%V
```

The constant MAXV denotes the largest value that can be assumed by B%V. The trace includes both the statement causing the access to B%V and the value of B%V.

E. Breaking the Execution of a Loop

Problem 6.9: The fragment between B$P1 and B$P2 is the body of a loop. Trace the first iteration of every execution of the loop.

Solution:

```
event on TRLOOP = (B_ia > $P1) && ~(|(B_ia == $P2))
event on ENDL = (B_ia > $P2)
origin TRLOOP at ENDL
trace on TRLOOP display $$
```

Problem 6.10: The fragment between B$F1 and B$F2 is the body of a loop having a for iteration scheme, and controlled by variable B%I. Break execution at the one-hundredth iteration.

Solution:

```
event on BL = (B_ia == $F1) && (B%I == 100)
break on BL
```

F. Single Stepping

Problem 6.11: Single-step each iterated execution of the program fragment between B$F1 and B$F2, introduced in Problem 6.10.

Solution:

```
event on SNG = (B_ia >= $F1) && (B_ia <= $F2)
break on SNG
```

G. Unreached Code

Problem 6.12: Consider the following program fragment:

```
(F$#100)  while ...
(F$#101)  do if ...
(F$#102)     then ...
(F$#103)     else if ...
(F$#104)          then ...
(F$#105)          else ...
(F$#106)  ...
```

The test data are such that each alternative sequence of statements should be executed [24]. Generate a break if the while statement terminates and one or more branches have not been tried at least once.

Solution:

```
event on INITW = (F_ia == $#100)
event on BR = (F_ia == $#102) & (F_ia == $#104) & (F_ia == $#105)
event on ERR = (F_ia == $#106) && ~BR
origin BR at INITW
break on ERR
```

H. Shared Variables

Problem 6.13: A bounded buffer of size one is shared between a sender process S and a receiver process R. The former deposits data into the buffer by executing the DEPOSIT procedure, the latter fetches data from the buffer by executing the FETCH procedure. Detect a possible synchronization error, resulting in a fetch issued before the corresponding deposit.

Solution:

```
event on D = |(DEPOSIT_ia@S == $#1)
event on F =  (FETCH_ia@R == $#1)
event on ENDF = (FETCH_ia@R == $^)
origin D at ENDF
event on ERROR = F && ~D
break on ERROR
```

Problem 6.14: A few concurrent processes use a binary semaphore MUTEX to implement mutually exclusive execution of critical sections. Detect possible errors in the usage of the P and V operations on the semaphore.

Solution: The event

```
event PP = (MUTEX > 1)
```

detects a missing P. The event

```
event VV = (MUTEX < 0)
```

detects a missing V. The event

```
event PV = PP || VV
```

detects both a missing P and a missing V. Let MAXM denote the largest value that can be stored in the area reserved for the value of MUTEX. The following commands detect the accesses to MUTEX, and, for each access, display the value of the semaphore together with the statement causing the access:

```
event on TRM = (MUTEX <= MAXM)
trace on TRM display $$ MUTEX
```

I. Message Passing

Problem 6.15: The following fragment of a concurrent program is written in a message-oriented language with synchronous (blocking) message passing. The fragment involves a multiple-clients/single-server relationship:

```
                process S;
                  var MS: MSG_TYPE;
                    .
                    .
(S$LS)              receive MS from REQUEST_PORT;
                    .
                    .
                end;

                process C1;
                  var M1: MSG_TYPE;
                    .
                    .
(C1$LC1)          send M1 to REQUEST_PORT;
                    .
                    .
                end;

                process C2;
                  var M2: MSG_TYPE;
                    .
                    .
(C2$LC2)          send M2 to REQUEST_PORT;
                    .
                    .
                end;
```

a) trace the contents of the messages received by the server S from client C1; b) trace the contents of all the messages received by S; c) generate a break when both C1 and C2 have sent at least one message to S.

Solution:

a)

```
event on MC1 = (C1_ia == $LC1) && (S_ia == $LS)
trace on MC1 display MS
```

b)

```
event on MC1C2 = ((C1_ia == $LC1) || (C2_ia == $LC2)) && (S_ia == $LS)
trace on MC1C2 display MS
```

c)

```
event on C12 = (C1_ia == $LC1) & (C2_ia == $LC2)
break on C12
```

VII. EVALUATION OF THE DEBUGGING ENVIRONMENT PROPOSED

This section discusses the properties of the debugging environment proposed. We will give particular attention to the outstanding debugging methodologies, and will evaluate the support given by the environment to the implementation of these methodologies.

A. State History

Most debuggers make it possible to express events in terms of the current state of the target program, whereas the user often needs to specify an event as a function of a sequence of program states (state history). Another user requirement is the dynamic control over traps, i.e. trap generation can be turned on or off, according to the program state history, without manual intervention.

The _id operators and the **origin at** command allow the user both to express conditions on the program state history, and to specify the interval when a trap is generated as a function of the state history (see Problems 6.1b) and 6.2b)).

B. Flow History

Triggering the generation of a trap on the execution of a given statement is possible in most debugging environments. However, frequently a statement can be reached by executing several control paths, but only a few of these paths are of actual interest to the debugging experiment. Thus, in an experiment based on break traps, the user must inspect the program state and try to understand the path actually followed by the flow of control (flow history). In an experiment based on trace traps, the trace must contain not only the values of the entities to be actually monitored, but also the state information aimed at the flow analysis. This complicates the tracing activity, and also places new burdens to the user.

In our approach, the _ia variables make it possible to treat flow-history events in the same way as state-history events. This is true, in particular, for dynamic control over trap generation. Applications of flow-history techniques have been shown in the solutions of Problems 6.3, 6.4, 6.5 and 6.15c). A dynamic control over trap generation based on the flow history has been used to solve Problems 6.7a), 6.7c), 6.9 and 6.12.

C. Selective Tracing

A trace can be generated by inserting print statements at appropriate points in the target program [24]. This technique is slow, and imposes a considerable manual effort on the user [23]. The tracing statements themselves are prone to errors [34], and this complicates the debugging activity. Automatic tracing techniques have therefore been devised. These techniques have a high cost in terms of both the storage for maintaining the trace, and of the time required for collecting it. An automatic trace usually contains much more information than is of actual interest to the debugging experiment. Effective utilization of such a voluminous amount of data requires proper filtering actions, to be carried out after trace gathering [14].

An alternative approach is to record only those events strictly pertinent to the experiment [13], [14]. This selective tracing is only effective if the user can express the events causing the trace traps at a suitable level of detail. For instance, most trace commands feature an option making it possible to limit the tracing activity to one or more subprograms, but the user often needs to restrict this activity to even smaller program portions. This feature is important in both bottom-up and top-down debugging [24].

The _ia variables make it possible to treat an event defined in terms of the execution of a statement in the same way as an event defined in terms of the value of a data item. In this way, trace gathering can be limited to one or more program fragments (see Problems 6.6 and 6.7). The _id operators and

the origin at command make it possible to specify selective tracing in terms of both flow and state histories (see Problems 6.7a), 6.7c), 6.8, and 6.9). An event can even involve conditions on both instruction execution and data values (see Problem 6.10).

D. Range Checks

An instantaneous range check is a range check expressed in terms of an instantaneous event. An instantaneous range check involving a given variable is carried out at each access to that variable, and will succeed if the value of the variable satisfies the given in-range or out-range condition. A deferred range check is a range check expressed in terms of a deferred event. A deferred range check involving a given variable is carried out at each access to that variable. The check succeeds at the first access such that the value of the variable satisfies the given range condition. The check also succeeds at each access following the first successful access.

Instantaneous range checks allow us to detect, for instance, a variable of a given type assuming a value which is not consistent with that type, if the pertinent run-time checks are not inserted by the compiler. A further application is to ascertain a violation of possible constraints on the value of a given variable, holding in a specific program portion [19]. The main applications of deferred range checks are in inspecting the behavior of the target program in critical, near-error situations.

Most debuggers make it possible to express instantaneous range checks, whereas the implementation of deferred range checks is based on repeated manual intervention by the user. The _id operators, on the other hand, allow us to express both classes of range checks. Examples are the solutions of Problems 6.1 and 6.2.

E. Unreached Code

Internal testing is a debugging technique based on the construction of test data such that all branches of the program are executed at least once [24]. This technique can help to detect an unreachable program portion. Unreachable code is often a consequence of an error, for instance, in the specification of a branch condition.

Internal testing is an application of flow-history techniques. As seen in the previous paragraphs, these techniques are well supported by our environment. An example of the application of internal testing for the detection of unreachable code has been presented in Problem 6.12.

F. Exception Handlers

Raising an exception is one way to handle anomalous processing states [21]. As such, it is an occasion when special attention from the programmer is often required during debugging [23]. This is especially true for unexpected exceptions, requiring careful inspection of the program state to detect the cause of the exception. For instance, on raising an exception of a given class, the programmer may need to understand the specific violation causing the exception. Of course, a further application is to debug the exception handler itself.

Facilities for getting the attention of the debugging system on encountering an exception have been actually introduced in existing debuggers [18], [21]. The _id operators allow us to meet an even more stringent requirement, i.e., capturing the exceptions of a given class only if they are raised from inside a given program fragment (a similar application has been considered in Problem 6.7a)). This can be useful, for instance, to tailor a library handler to a specific program unit.

G. Single Step

Single-stepping, or returning control to the programmer at the execution of each statement, is useful especially when it is not clear where to place break traps [10]. In a few debugging environments, in order to single-step a given program fragment, the programmer must insert a break trap at the first statement of the fragment. When this break is encountered, he must issue an ad-hoc command, causing the debugging system to enter the single-step mode of operation. On the execution of the last statement of the fragment, he will issue another command, causing the single-step mode to be abandoned. In other debugging environments, a single command makes it possible to single-step a given number of statements. In both approaches, the manual effort is heavy, especially for a repeatedly-executed fragment such as the body of a loop.

The instruction address variables and the **origin at** command allow us to single-step one or more fragments, by defining a few _ia events in term of the first and last statement of each such fragment. No manual intervention is required even to monitor iterated execution of one such fragment. An example of this application has been shown in Problem 6.11.

H. Break Traps Vs. Trace Traps

The properties of the debugging techniques based on break traps have been compared with the properties of the techniques based on trace traps [14], [34]. The main advantage advocated to break traps is the better control over the layout of the debugging experiment. At a breakpoint, the user can activate and deactivate traps, and dynamically decide the items to be displayed. Data display is experimenter command driven in a debugging experiment based on break traps, whereas it is event driven in an experiment based on trace traps [6].

On the other hand, the dynamic control over the tracing activity, made possible by the state and flow histories, allows us to gain the advantages of

a precise specification of the actions to be taken on occurrence of a given trap, even in the utilization of trace traps. This is of particular interest as far as concurrent program debugging is concerned.

I. Sequential Program Debugging Vs. Concurrent Program Debugging

The techniques for sequential program debugging have been considered inadequate for concurrent program debugging [14], [26]. This is, essentially, a consequence of the lack of reproducibility of the execution of concurrent programs, due to communication and scheduling delays [13]. Break traps have been deemed useful in the debugging of a concurrent program only within critical sections, when the program actually behaves as if it were sequential [12].

On the other hand, the time and space problems, proper of automatic tracing, are compounded by the fact that events are generated by many processes. A careful selection of events of actual interest is therefore mandatory [5], [13], [14], [26]. For instance, if the target program is written in a message-oriented programming language, important events are the transmission and reception of a message. However, even tracing every message transmitted or received has a high cost. The user should be allowed to restrict trace gathering to a single process, as well as to exclude a given process from the tracing activity [12].

In our opinion, break traps can hardly be used in concurrent program debugging as long as the events controlling the traps must be expressed in terms of entities local to a single process, and cannot take into account the past interactions of the concurrent processes. In a sequential program, the flow of control and the ordering of the events are well-known factors, whereas this is not the case for concurrent programs, on account of their non-deterministic nature. The behavior of one such program is independent of the relative execution speed if the program works correctly. Of course, this is

not the case when the program is being debugged.

Our environment features powerful tools for expressing events in terms of entities defined in different processes. The _ia variables make it possible to specify conditions on the execution of statements in different control flows. The _id operators allow us to express relations involving the value of target variables defined in program units to be executed concurrently. In this way, a break trap can be controlled by concurrent events. This is a suitable means for monitoring the interactions between processes, and extends the range of applicability of break trap techniques. The user can now take profit of the salient advantages of these techniques, for example, the better control over the debugging experiment, even in concurrent program debugging.

Important advantages can however also be gained using trace-based techniques. The definition of compound events concerning different processes makes it possible to also apply the techniques for selective tracing, illustrated previously, to the execution of concurrent programs. For instance, we can limit trace gathering to just a few message exchanges (see Problem 6.15a)). In this way, the user tailors event filtering to the debugging experiment, with a level of granularity which is much higher than that made possible when a list of important events is part of the debugger design.

J. Language-Independent Debugging Systems

The main reason for a language-independent debugging system is that the user does not have to learn more than a single debugging command set [4]. This is especially useful in multi-lingual applications. Packages written in different languages can be integrated in a single program, and their behavior monitored by means of a single debugger [33].

However, very few debuggers actually support multiple high-level languages. This is especially true for concurrent program debuggers. As already seen, these debuggers are often strongly oriented towards a specific

language. For instance, a debugger for a message-oriented language essentially monitors message exchanging activities [26]. In a debugger for a programming language featuring a synchronization mechanism based on path expressions, the operations on semaphores are the primitive events [13]. A debugging tool for a language featuring constructs for sharing variables traces the values assumed by these variables in order to detect the critical sections possibly defined improperly [12].

On the other hand, the _id operators and the _ia variables are inherently independent of the target programming language. In particular, they can be used to monitor interactions between concurrent processes independently of the mechanisms for interprocess synchronization and communication (see Problems 6.13-6.15). Interactions are monitored when they actually occur, instead of by indirect reference to their effects.

VIII. SUMMARY AND CONCLUSIONS

A debugging environment has been presented which is based on the event-action model for interaction between the debugging system and the target program. The salient features of this environment are the instruction address variables and the instantaneous/deferring operators. The instruction address variables make it possible to treat an event expressed in terms of the execution of a statement in the same way as an event expressed in terms of the value of a program-defined variable. The instantaneous/deferring operators make it possible to define event abstractions in terms of both the flow and the state histories. In this way, the behavior of the target program can be monitored at the two levels of the program state and of the program activity [3].

The proposed environment offers the user a considerable degree of control over the debugging experiment, and allows him to build up those tools which most adequately support his intended debugging technique. This approach is in

direct contrast with that in which the environment offers a fixed set of special-purpose tools, tailored for a specific approach to program debugging.

APPENDIX

IMPLEMENTING EVENTS

This appendix examines certain important aspects of the implementation of the proposed debugging environment. The evaluation of _id operators is first considered. A possible approach to event processing is then presented. Finally, mechanisms which can be used to reveal the evaluation times of simple events are analysed, and a few observations concerning efficiency are presented.

A. Evaluating _id Operators

Instantaneous negation. Let $t_1, t_2, \ldots$ be the evaluation times of the instantaneous negation ~E of event E. Let $i_j^{(E)}$ and $n_j^{(E)}$ denote the values of E and of ~E in the interval $[t_j, t_{j+1})$. Then

$$n_j^{(E)} = \overline{i_j^{(E)}} \qquad (A.1)$$

This relation follows directly from Definition 2.1.

Deferrer. Let $t_1, t_2, \ldots$ be the evaluation times of the deferrer $|^{(t_0)}E$ of event E. Let $i_j^{(E)}$ and $d_j^{(E)}$ denote the value of E and of $|^{(t_0)}E$ in the interval $[t_j, t_{j+1})$. Then

$$d_j^{(E)} = \begin{cases} i_0^{(E)} & j=0 \\ i_j^{(E)} \vee d_{j-1}^{(E)} & j=1,2,\ldots \end{cases} \qquad (A.2)$$

This relation follows directly from Definition 2.2.

Instantaneous conjunction. Let $t_1, t_2, \ldots$ be the evaluation times of the

instantaneous conjunction E_1 && E_2 of events E_1 and E_2. Let $i_j^{(E_1)}$, $i_j^{(E_2)}$ and $ic_j^{(E_1,E_2)}$ denote the value of E_1, E_2 and E_1 && E_2 in the interval $[t_j, t_{j+1})$. Then

$$ic_j^{(E_1,E_2)} = i_j^{(E_1)} \wedge i_j^{(E_2)} \tag{A.3}$$

This relation follows directly from Definition 2.3.

Deferred conjunction. Let $t_1, t_2, \ldots$ be the evaluation times of the deferred conjunction $E_1 \&^{(t_0)} E_2$ of events E_1 and E_2. Let $d_j^{(E_1)}$, $d_j^{(E_2)}$ and $dc_j^{(E_1,E_2)}$ denote the value of $|^{(t_0)}E_1$, $|^{(t_0)}E_2$ and $E_1 \&^{(t_0)} E_2$ in the interval $[t_j, t_{j+1})$. Then

$$dc_j^{(E_1,E_2)} = d_j^{(E_1)} \wedge d_j^{(E_2)}$$

$$= \begin{cases} i_0^{(E_1)} \wedge i_0^{(E_2)} & j=0 \\ (i_j^{(E_1)} \vee d_{j-1}^{(E_1)}) \wedge (i_j^{(E_2)} \vee d_{j-1}^{(E_2)}) & j=1,2,\ldots \end{cases} \tag{A.4}$$

This relation follows directly from Relations (2.12) and (A.2).

Instantaneous disjunction. Let $t_1, t_2, \ldots$ be the evaluation times of the instantaneous disjunction $E_1 \,||\, E_2$ of events E_1 and E_2. Let $i_j^{(E_1)}$, $i_j^{(E_2)}$ and $id_j^{(E_1,E_2)}$ denote the value of E_1, E_2 and $E_1 \,||\, E_2$ in the interval $[t_j, t_{j+1})$. Then

$$id_j^{(E_1,E_2)} = i_j^{(E_1)} \vee i_j^{(E_2)} \tag{A.5}$$

This relation follows directly from Definition 2.5.

Deferred disjunction. Let $t_1, t_2, \ldots$ be the evaluation times of the deferred disjunction $E_1 |^{(t_0)} E_2$ of events E_1 and E_2. Let $d_j^{(E_1)}$, $d_j^{(E_2)}$ and $dd_j^{(E_1,E_2)}$ denote the value of $|^{(t_0)}E_1$, $|^{(t_0)}E_2$ and $E_1 |^{(t_0)} E_2$ in the interval $[t_j, t_{j+1})$. Then

$$dd_j^{(E_1,E_2)} = d_j^{(E_1)} \vee d_j^{(E_2)}$$

$$= \begin{cases} i_0^{(E_1)} \vee i_0^{(E_2)} & j=0 \\ (i_j^{(E_1)} \vee d_{j-1}^{(E_1)}) \vee (i_j^{(E_2)} \vee d_{j-1}^{(E_2)}) & j=1,2,\ldots \end{cases} \tag{A.6}$$

This relation follows directly from Relations (2.13) and (A.2).

B. Evaluating Events

Let C be a compound event. A Boolean variable I is associated with each unary deferring operator in terms of which C is expressed. This variable is called the image of the factor of that operator. Two Boolean variables I' and I" are associated to each binary deferring operator in terms of which C is expressed. These variables are called the left image and the right image, and are associated to the left and right factors of the operator, respectively. When a deferring operator is evaluated, the value of the deferrer of that factor is assigned to the factor's image. The evaluation of event C at time t_j consists in evaluating each operator included in that event, according to Relations (A.1)÷(A.6). This action involves not only the value at time t_j of the factor(s) of each operator, but also the value at time t_{j-1} of the deferrer(s) of the factor(s) of each deferring operator. These values are stored in the image(s) of that operator.

C. Event Descriptors

A descriptor $DESCR_E$ is associated to each event E. The descriptor contains the whole state of that event, consisting of the following items: i) the specification of the event; ii) the value V_E, i.e. the result of the most recent evaluation of the event; iii) the images of the deferring operators in terms of which E is expressed; iv) an evaluation list E_LIST_E; v) an origin list G_LIST_E; vi) an origin flag G_E; vii) an evaluation flag L_E; viii) a trap flag T_E; ix) a break flag B_E; and, finally, x) a trace list T_LIST_E. The evaluation list contains the identifiers of the events expressed in terms of E, i.e., the events to be evaluated as a consequence of the evaluation of E. The origin list G_LIST_E contains the identifiers of the events controlled by

E, i.e., the events whose origin must be moved when E occurs. The origin flag G_E, if set, specifies that a controller of E has occurred, and, therefore, the origin of E must be moved. The evaluation flag L_E, if set, specifies that an evaluation time of E has been generated. The trap flag T_E, if set, specifies that E has occurred, and, therefore, any traps connected with E must be triggered. The break flag B_E, if set, specifies that a break trap is connected with E, and the trace list T_LIST_E, if not empty, specifies that a trace trap is connected with E. The items whose value must be actually traced are specified by the contents of this list.

D. Event Processing

The event descriptors are the nodes of a directed graph, the descriptor graph. In this graph, if E is a component of a compound event C, a directed edge connects the descriptor $DESCR_E$ to the descriptor $DESCR_C$ of event C. If E controls a destination event D, a directed edge connects $DESCR_E$ to the descriptor $DESCR_D$ of event D. As stated in Section IV-A and Section IV-B, for a given compound event, the value of each component event never depends on the value of that compound event, and, for a given destination event, the value of each controller never depends on the value of that destination event. Therefore, the descriptor graph contains no cycles.

Later in this appendix, we will discuss mechanisms for revealing the accesses to the factors of the active simple events. When an access to a factor of one or more such events is revealed, control is returned to the debugging system, which sets the evaluation flags of these events, and processes all active events. The actions connected with event processing are carried out in two phases. In the first phase, the origins of the active events may be moved, and the events may be evaluated. In the second phase, the traps connected with the occurring events are executed. In both phases, events are processed in the order which results from applying a topological

sort to the descriptor graph. As the graph is acyclic, the topological sort is always successful [35]. In this way, the evaluation of a given event never precedes the evaluation of other events which must take place first, and the value of the deferrer of a factor of a deferred event is never assigned to the factor's image before evaluating that factor.

First phase. For each event E, the following actions are carried out:

A. If the origin flag G_E is set
then

   1. clear each image possibly stored in the descriptor $DESCR_E$;
   2. store the value FALSE in V_E;
   3. set the evaluation flag L_E;
   4. clear the origin flag G_E;

B. if the evaluation flag L_E is set
then

   1. evaluate event E, and store the result in V_E;
   2. if V_E is true
      then

      - set the trap flag T_E;
      - set the origin flag of each event in the origin list G_LIST_E;

   3. set the evaluation flag of each event in the evaluation list E_LIST_E;
   4. clear the evaluation flag L_E.

Second phase. For each event E, the trap flag T_E is considered. If this flag is set, the trace specified by the trace list T_LIST_E is generated, and then, if the break flag B_E is set, the break mode is entered.

Example A.1:

Let us consider the following sequence of commands:

```
event  S1 = ...
event  S2 = ...
event on   C = S1 & S2
origin C at S2
break on C
```

These commands define two simple events S1 and S2, and a compound event C. Suppose that S1 is evaluated at t_1, t_2 and t_4, and is true in $[t_1,t_2)$ and from t_4 onwards. Suppose also that S2 is evaluated at t_3, and is true from t_3 onwards (Fig. 3). Thus, C is evaluated at t_1, t_2, t_3 and t_4, and is true from t_4 onwards. The origin of C is moved at t_3, as a consequence of the occurrence of S2.

The descriptor graph for events S1, S2 and C is shown in Fig. 4, The descriptors are displayed in the linear ordering which results from a topological sort. The evolution of the state of the three events for the first phase is shown in Table 1. For each evaluation time, the evaluation flag set when that evaluation time is revealed is specified in round brackets. For each event, the table shows the modifications of the state of that and of the other events, occurring when that event is processed.

E. Revealing the Evaluation Times of Simple Events

The mechanisms to reveal the evaluation times of a simple event depend on both the factors and the relational operator involved in that event. For some simple events, these mechanisms can be fully implemented at the software level, for others, ad-hoc hardware support is required. The hardware support is always mandatory if we are concerned with real-time program debugging [16], [25].

Let us first refer to a simple event expressed in terms of a target variable which is not an _ia variable, and a constant factor such as a numeric or string literal. An example could be the event (A <= 100). The evaluation times of such an event are the times when the data location implementing that

TABLE 1. EVOLUTION OF THE STATE OF THE EVENTS S1, S2 AND C, CONSIDERED IN EXAMPLE A.1

time t_1 ($L_{S1} \leftarrow 1$):

event S1: $V_{S1} \leftarrow$ true; $T_{S1} \leftarrow 1$; $L_C \leftarrow 1$; $L_{S1} \leftarrow 0$

event S2:

event C: $I'_{\&} \leftarrow$ true; $I''_{\&} \leftarrow$ false; $V_C \leftarrow$ false; $L_C \leftarrow 0$

time t_2 ($L_{S1} \leftarrow 1$):

event S1: $V_{S1} \leftarrow$ false; $L_C \leftarrow 1$; $L_{S1} \leftarrow 0$

event S2:

event C: $I'_{\&} \leftarrow$ true; $I''_{\&} \leftarrow$ false; $V_C \leftarrow$ false; $L_C \leftarrow 0$

time t_3 ($L_{S2} \leftarrow 1$):

event S1:

event S2: $V_{S2} \leftarrow$ true; $T_{S2} \leftarrow 1$; $G_C \leftarrow 1$; $L_C \leftarrow 1$; $L_{S2} \leftarrow 0$

event C: $I'_{\&} \leftarrow$ false; $I''_{\&} \leftarrow$ false; $V_C \leftarrow$ false; $L_C \leftarrow 1$; $G_C \leftarrow 0$;
$I'_{\&} \leftarrow$ false; $I''_{\&} \leftarrow$ true; $V_C \leftarrow$ false; $L_C \leftarrow 0$

time t_4 ($L_{S1} \leftarrow 1$):

event S1: $V_{S1} \leftarrow$ true; $T_{S1} \leftarrow 1$; $L_C \leftarrow 1$; $L_{S1} \leftarrow 0$

event S2:

event C: $I'_{\&} \leftarrow$ true; $I''_{\&} \leftarrow$ true; $V_C \leftarrow$ true; $T_C \leftarrow 1$; $L_C \leftarrow 0$

variable is accessed. At the software level, these times can be revealed by replacing every instruction referencing that variable by a software interrupt, e.g. an instruction raising an exception. This interrupt transfers control to the debugging system, which must execute (or emulate execution of) the instruction replaced before resuming execution of the target program. Of course, this approach cannot detect an access to a variable expressed in terms

of a computed addressing mode (e.g., the indexed mode) [22]. At the hardware level, an ad-hoc circuitry inspecting the address bus can detect each access to the variable to be monitored, by generating a hardware interrupt when the address of that variable is recognised [25]. Tagged architectures allow us to flag the data locations to be monitored by means of a specific tag configuration. An access to one such location generates a call to the debugging system [32]. In an object-oriented architecture, a specific data configuration can be used to mark a variable to be monitored. This data configuration is inserted at a fixed location in the segment implementing that variable. The contents of this location must be inspected by the microprogram at each access to that variable (a similar approach is adopted, for instance, in [28], in order to reveal erroneous accesses to uninitialized objects). In a capability architecture featuring a capability dereferencing mechanism based on capability registers [27], an ad-hoc field in each such register can be used to generate an interrupt at each access to the object referenced by that register.

Let us now refer to a simple event expressed in terms of two target variables, e.g. (A < B). Revealing the evaluation times of such an event means detecting the accesses to both factors. Of course, each factor can be treated independently of the other.

Finally, let us refer to _ia events. As seen in Section III, the evaluation times of these events are all the times at which a statement is executed in the block referenced by that _ia event. This implies that the debugging system must be invoked on the execution of every statement in that block. At the software level, this can be achieved by replacing the first machine instruction translating each such statement by a trap instruction. At the hardware level, a small amount of circuitry is required to generate an interrupt on the execution of each instruction [1]. This feature is usually controlled by the software, for instance, by means of a bit in the program

status word of the processor. This bit will be set and cleared at the beginning and at the end of the block to be monitored, respectively. If all the _ia events active at a given time are relevant to the same block, a couple of bound registers, such as those provided by the architecture of the IBM System/370 [9], can be loaded with the address of the first and last instruction of that block. If more than a single block is referenced by the active simple events, the contents of the bound registers will be updated at the beginning of execution of each block. In a paged environment, execution of an instruction in a given page can be detected by setting the access right field of that page to no right. In a tagged architecture, a specific tag configuration can be reserved to denote instructions whose execution causes a call to the debugging system [11]. A debugging co-processor can inspect the value of tags in parallel with elaborations of the main processor [20]. A different approach uses a high-speed associative memory to hold a list of the interrupt addresses [1].

F. Optimizing the Implementation of _ia Events

Invoking the debugging system at the execution of each statement of a block in order to implement an _ia event has a high cost in terms of the execution times of the debugging system. In concurrent program debugging, the resulting delays may give rise to the so-called probe effect: the behavior of the target program is influenced to such an extent as to invalidate the results of the debugging experiment [12]. A careful implementation can limit the consequences of this effect. Of course, the time requirements are even more stringent as far as real-time program debugging is concerned [16]. A few improvements in the implementation of _ia events significantly reduce this time overhead.

Consider, for instance, the following sequence of commands:

```
event IA = (BLK_ia == $#100)
event on CMP = |IA
trace on CMP display ...
```

The trace activated by the last command is controlled by the compound event CMP, and this event is expressed in terms of the _ia event IA. The trace is generated on each occurrence of CMP, i.e., at each evaluation time of IA after its first occurrence. Now consider the following command pair:

```
event on IB = (BLK_ia == $#200)
trace on IB display ...
```

The trap action is controlled by the _ia event IB, which is not included in the definition of any compound event. Therefore, we can reveal the occurrence of IB, rather than its evaluation times.

In conclusion, when an _ia event is activated, we can set up the mechanisms to reveal only the occurrence of that event. The mechanisms for the detection of the evaluation times will be eventually set up later, on the activation of a compound event expressed in terms of that _ia event.

This approach is of particular interest for _ia events, such as event IB introduced above, which are defined in terms of the equality operator and are used to generate a trap on execution of a statement. The widespread use of this application of _ia events makes an optimized implementation particularly attractive. The occurrence of such an event can be easily revealed via software, by simply replacing the first machine instruction implementing that statement by a trap instruction [4].

A different optimization is possible if the event controlling the trap is expressed in terms of the instantaneous conjunction of two _ia events for the same block, and neither of the events are components of other events. An example is shown below:

```
event INF = (BLK_ia > $#10)
event SUP = (BLK_ia < $#100)
event on IR = INF && SUP
trace on IR
```

In this case, we only need to generate a call to the debugging system on the execution of the statements between BLK$#10 and BLK$#100, instead of on the execution of every statement of BLK. Note that no advantage can be gained here by using the optimization technique described previously.

REFERENCES

1. D. Abramson, J. Rosenberg, "Hardware Support for Program Debuggers in a Paged Virtual Memory", Computer Architecture News, Vol. 11, No. 2 (June 1983) pp. 8-19.

2. G. R. Andrews, F. B. Schneider, "Concepts and Notations for Concurrent Programming", Computing Surveys, Vol. 15, No. 1 (March 1983), pp. 3-43.

3. P. Bates, J. C. Wileden, "An Approach to High-Level Debugging of Distributed Systems", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, Pacific Grove, California (March 1983), in: Software Engineering Notes, Vol. 8, No. 4 (August 1983), SIGPLAN Notices, Vol. 18, No. 8 (August 1983), pp. 107-111.

4. B. Beander, "VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, Pacific Grove, California (March 1983), in: Software Engineering Notes, Vol. 8, No. 4 (August 1983), SIGPLAN Notices, Vol. 18, No. 8 (August 1983), pp. 173-179.

5. H. K. Berg, M. G. Smith, "A Distributed System Experimentation Facility", Proceedings of the Third International Conference on Distributed Computing Systems, Miami, Florida, October 1982, pp. 324-329.

6. D. Bhatt, M. Schroeder, "A Comprehensive Approach to Instrumentation for Experimentation in a Distributed Computing Environment", Proceedings of the Third International Conference on Distributed Computing Systems, Miami, Florida, October 1982, pp. 330-340.

7. P. Brinch Hansen, "Testing a Multiprogramming System", Software-Practice and Experience, Vol. 3, No. 3 (July-September 1973), pp. 145-150.

8. A. R. Brown, W. A. Sampson, Program Debugging, Macdonald/American Elsevier (1973).

9. R. P. Case, A. Padegs, "Architecture of the IBM System/370", Communications of the ACM, Vol. 21, No. 1 (January 1978), pp. 73-96.

10. B. Elliot, "A High-Level Debugger for PL/I, Fortran and Basic", Software-Practice and Experience, Vol. 12, No. 4 (1982), pp. 331-340.

11. E. A. Feustel, "On the Advantages of Tagged Architecture", IEEE Transactions on Computers, Vol. C-22, No. 7 (July 1973), pp. 644-656.

12. J. Gait, "A Debugger for Concurrent Programs", Software-Practice and Experience, Vol. 15, No. 6 (June 1985), pp. 539-554.

13. M. E. Garcia, W. J. Berman, "An Approach to Concurrent Systems Debugging", Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, Colorado, May 1985, pp. 507-514.

14. H. Garcia-Molina, F. Germano, W. H. Kohler, "Debugging a Distributed Computing System", IEEE Transactions on Software Engineering, Vol. SE-10, No. 2 (March 1984), pp. 210-219.

15. C. Ghezzi, M. Jazayeri, Programming Language Concepts, Wiley, 1982.

16. R. L. Glass, "Real-Time: The 'Lost World' of Software Debugging and Testing", Communications of the ACM, Vol. 23, No. 5 (May 1980), pp. 264-271.

17. W. C. Gramlich, "Debugging Methodology", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, Pacific Grove, California (March 1983), in: Software Engineering Notes, Vol. 8, No. 4 (August 1983), SIGPLAN Notices, Vol. 18, No. 8 (August 1983), pp. 1-3.

18. J. J. Hart, "The Advanced Interactive Debugging System (AIDS)", ACM Sigplan Notices, Vol. 14, No. 12 (December 1979), pp. 110-121.

19. D. D. Hill, "A Hardware Mechanism for Supporting Range Checks", Computer Architecture News, Vol. 9, No. 4 (June 1981), pp. 15-21.

20. C. R. Hill, "A Real-Time Microprocessor Debugging Technique", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, Pacific Grove, California (March 1983), in: Software Engineering Notes, Vol. 8, No. 4 (August 1983), SIGPLAN Notices, Vol. 18, No. 8 (August 1983), pp. 145-148.

21. D. Holdsworth, "A System for Analysing Ada Programs at Run-time", Software-Practice and Experience, Vol. 13, No. 5 (May 1983), pp. 407-421.

22. M. S. Johnson, "Some Requirements for Architectural Support of Software Debugging", Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, Palo Alto, California (March 1982), in: Computer Architecture News, Vol. 10, No. 2 (March 1982), SIGPLAN Notices, Vol. 17, No. 4 (April 1982), pp. 140-148.

23. J. D. Johnson, G. W. Kenney, "Implementation Issues for a Source Level Symbolic Debugger", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, Pacific Grove, California (March 1983), in: Software Engineering Notes, Vol. 8, No. 4 (August 1983), SIGPLAN Notices, Vol. 18, No. 8 (August 1983), pp. 149-151.

24. S. Lauesen, "Debugging Techniques", Software-Practice and Experience, Vol. 9, No. 1 (January 1979), pp. 51-63.

25. B. Lazzerini, C. A. Prete, L. Lopriore, "A Programmable Debugging Aid for Real-Time Software Development", IEEE Micro, Vol. 6, No. 3 (June 1986), pp. 34-42.

26. R. J. LeBlanc, A. D. Robbins, "Event-Driven Monitoring of Distributed Programs", Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, Colorado, May 1985, pp. 515-522.

27. H. M. Levy, Capability-Based Computer Systems, Digital Press, 1984.

28. L. Lopriore, "Capability Based Tagged Architectures", IEEE Transactions on Computers, Vol. C-33, No. 9 (September 1984), pp. 786-803.

29. M. A. F. Mullerburg, "The Role of Debugging Within Software Engineering Environments", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, Pacific Grove, California (March 1983), in: Software Engineering Notes, Vol. 8, No. 4 (August 1983), SIGPLAN Notices, Vol. 18, No. 8 (August 1983), pp. 81-90.

30. G. J. Myers, Software Reliability, Wiley, 1976.

31. G. J. Myers, The Art of Software Testing, Wiley, 1979.

32. G. J. Myers, Advances in Computer Architecture, 2nd Edition, Wiley-Interscience (1982).

33. R. Seidner, N. Tindall, "Interactive Debug Requirements", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, Pacific Grove, California (March 1983), in: Software Engineering Notes, Vol. 8, No. 4 (August 1983), SIGPLAN Notices, Vol. 18, No. 8 (August 1983), pp. 9-22.

34. J. L. Steffen, "Experience with a Portable Debugging Tool", Software-Practice and Experience, Vol. 14, No. 4 (April 1984), pp. 323-334.

35. J. Tremblay, P. G. Sorenson, An Introduction to Data Structures With Applications, McGraw-Hill, 1984.

36. F. van der Linden, I. Wilson, "An Interactive Debugging Environment", IEEE Micro, Vol. 5, No. 4 (August 1985), pp. 18-31.

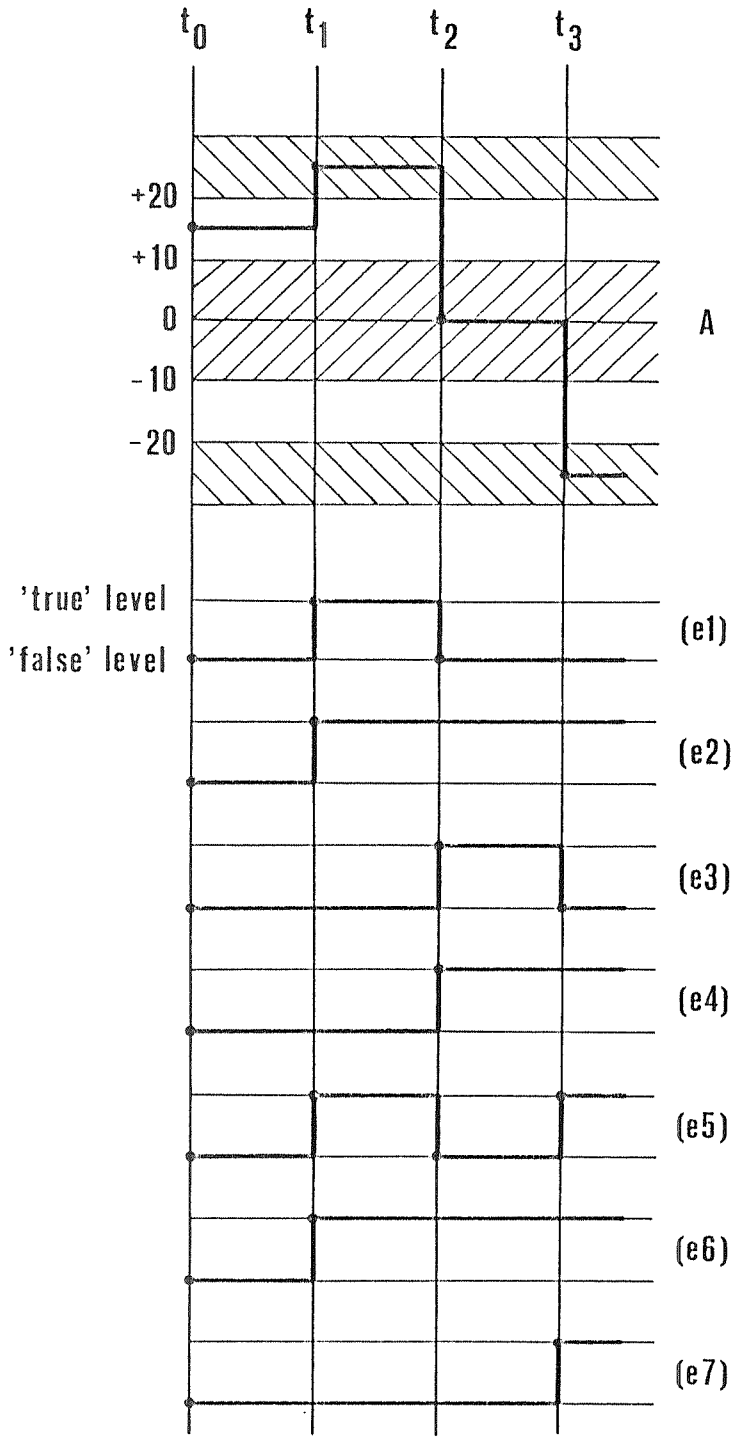

Fig. 1. Timing diagrams for variable A and events (e1)÷(e7) considered in Example 2.1.

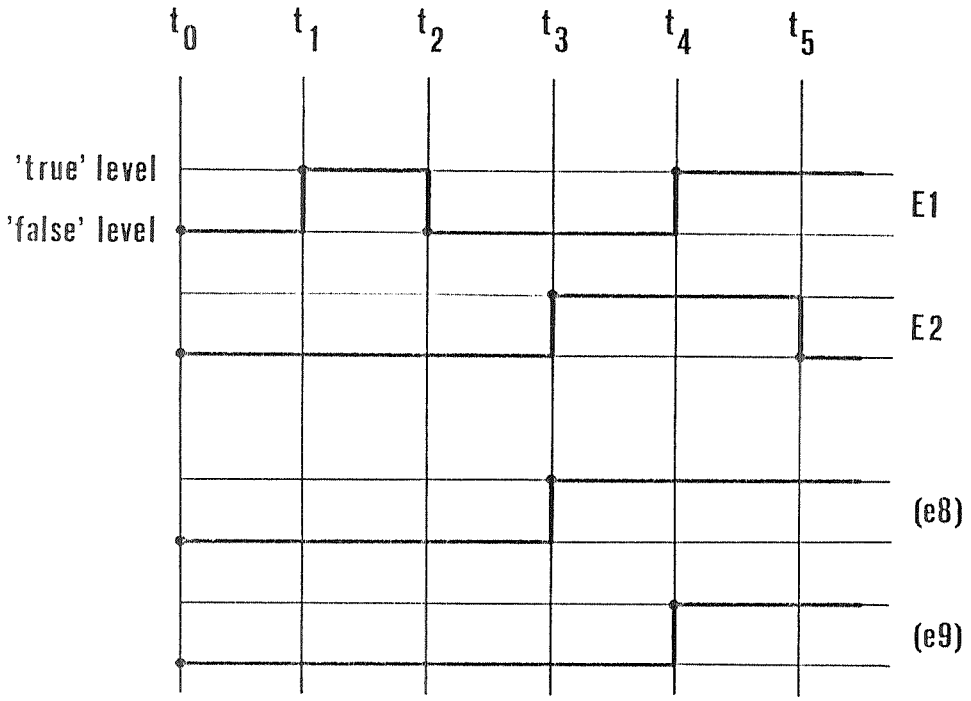

Fig. 2. Timing diagrams for events E1, E2, (e8) and (e9) considered in Example 2.2.

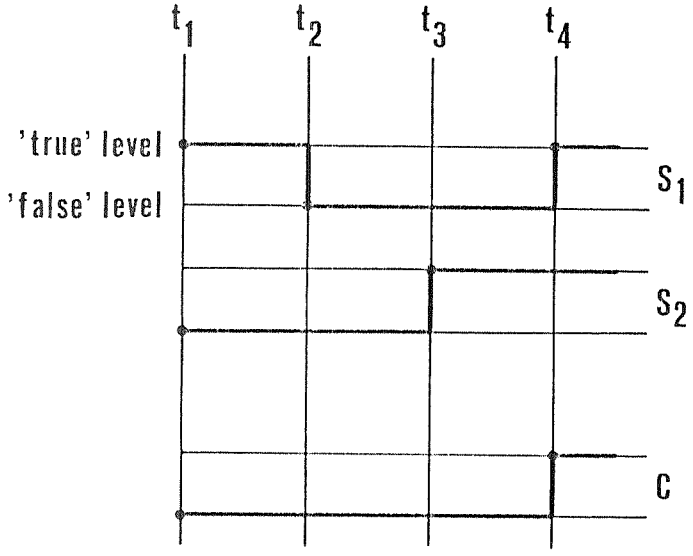

Fig. 3. Timing diagrams for events S1, S2 and C considered in Example A.1.

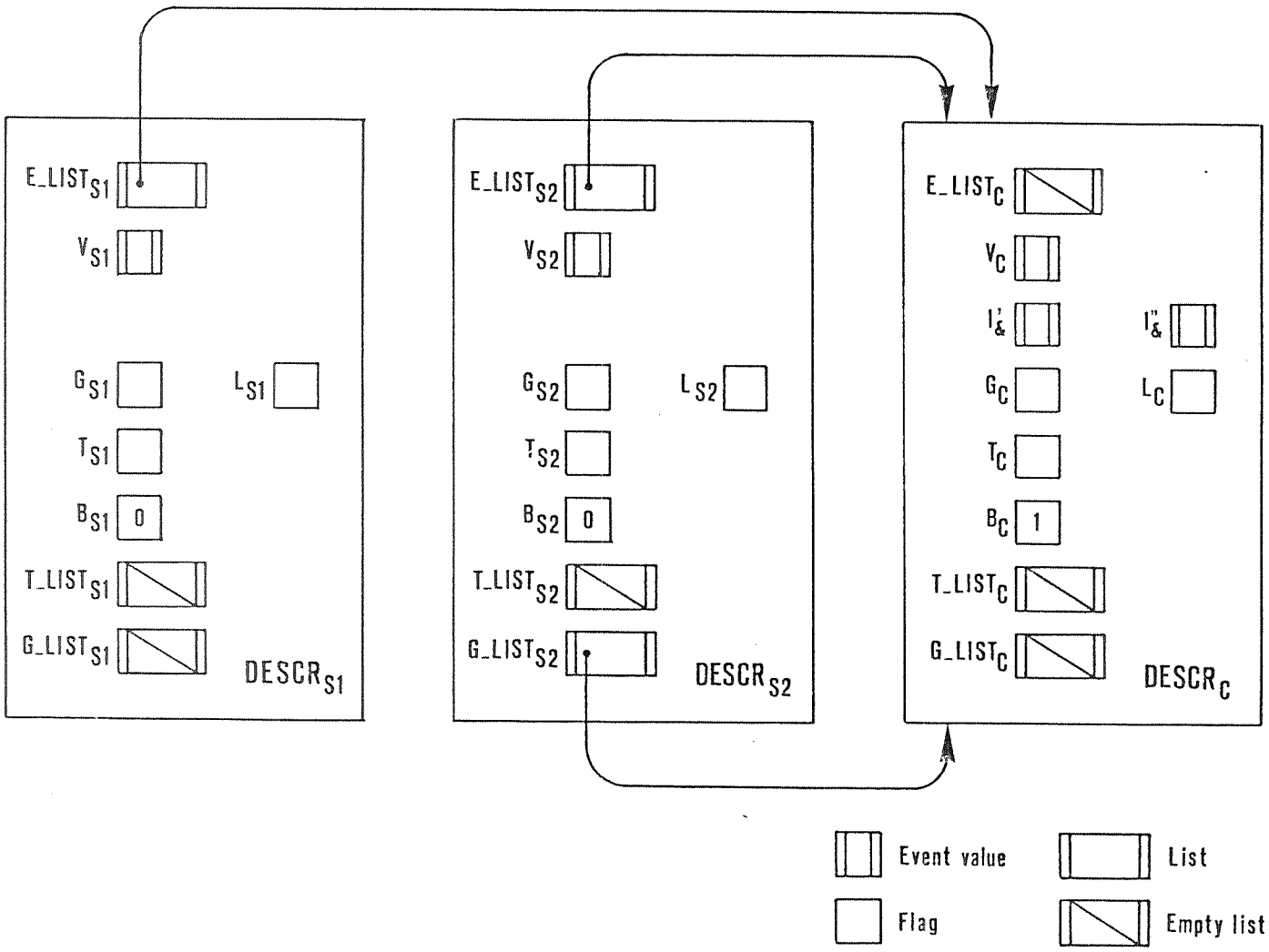

Fig. 4. Descriptor graph for events S1, S2 and C considered in Example A.1.