Johanna M. Chow received the B.S. degree (summa cum laude) in mathematics/computer science from the University of California, Los Angeles, in 1980.

She is currently working on the Ph.D. degree in computer science at UCLA. Her thesis deals with the issues of transaction management and recoverable processes in distributed systems.

David Edwards was born on June 15, 1957 in Glendale, CA and grew up in La Crescenta. He received the B.S. degree in mathematics/computer science from the University of California, Los Angeles, in 1979.

As a graduate student at UCLA, he has worked on the LOCUS Distributed Operating System project under Gerald Popek. He has been a departmental scholar and the recipient of a fellowship from System Engineering Laboratories. His current interests are in operating system design, especially distributed and in the interaction between the designs of hardware architecture and system software.

Stephen Kiser received the B.S. degree in mathematics/computer science from the University of California, Los Angeles, in 1979.

He is currently working towards the M.S. degree at UCLA and is also employed by the Xerox Corporation, El Segundo, CA. His interests are in distributed computing, computer theory, and programming languages.

Charles Kline received the B.S., M.S., and Ph.D. degrees from the University of California, Los Angeles, in 1970, 1971, and 1980, respectively.

His Ph.D. research was in the areas of computer and network security, especially the use of security kernels and encryption. He is the author of numerous papers on the issues of computer security, security kernels, and the use of public key and conventional encryption in computer network security. He has been employed as a researcher in the Department of Computer Science at UCLA since 1966. During that period his research areas have included operating systems, languages, networks, distributed systems, and computer security problems.

Input-Output Tools: A Language Facility for Interactive and Real-Time Systems

JAN VAN DEN BOS, MARINUS J. PLASMEIJER, AND PIETER H. HARTEL

Abstract—A conceptual model is discussed which allows the hierarchic definition of high-level input driven objects, called input-output tools, from any set of basic input primitives. An input-output tool is defined as a named object. Its most important elements are the input rule, output rule, internal tool definitions, and a tool body consisting of executable statements. The input rule contains an expression with tool designators as operands and with operators allowing for sequencing, selection, interleaving, and repetition. Input rules are similar in appearance to production rules in grammars. The input expression specifies one or more input sequences, or input patterns, in terms of tool designators. An input parser tries, at run-time, to match (physical) input tokens against active input sequences. If a match between an input token and a tool designator is found, the corresponding tool body is executed, and the output is generated according to specifications in the tool body. The control structures in the input expression allow a vari-

ety of input patterns from any number of sources. Tool definitions may occur in-line or be stored in a library. All tools are ultimately encompassed in one tool representing the program.

The input-output tool model offers a nonprocedural input specification language with a parser provided by the run-time system. It forces clean and structured programs and allows for easy definition of abstract input devices and simulation of physical devices on other devices. Implementations have been completed and are being evaluated.

Index Terms—Computer graphics, dialogue, input functions, input tools, interaction language, process control, programming language, real time, specification language.

I. INTRODUCTION

INTERACTIVE computing, in which we include real-time systems and process control, forms a sizable, perhaps more than 50 percent, share of all computing. Most interactive systems have been programmed using regular (or slight derivatives of) batch programming languages. Unfortunately, these languages are badly lacking in provisions for handling input and output on an advanced level. Few, if any of these languages, have, for example, provisions to read one input source out of several specified. Facilities to define named abstract input devices in terms of (a collection of) existing physical devices

Manuscript received October 2, 1981; revised October 1, 1982. This work was supported in part by a grant from The Netherlands Organization for the Advancement of Pure Research (ZWO).

J. van den Bos and M. J. Plasmeijer are with the Department of Computer Science and the Computer Graphics Group, University of Nijmegen, Nijmegen, The Netherlands.

P. H. Hartel was with the Department of Computer Science and the Computer Graphics Group, University of Nijmegen, Nijmegen, The Netherlands. He is now with the Department of Computer Science, University of Amsterdam, Amsterdam, The Netherlands.

are absent. For interactive systems it can be stated unequivocally that input and output play a predominant, closely coupled role. Such systems typically are in an inactive state until triggered by an input stimulus. The stimulus causes some action to be performed after which the system returns to a new state of equilibrium. Conventional programming languages do not cater for this type of program. It appears, therefore, important that future programming languages be developed which are more oriented towards handling of input and output.

In practice the handling of input in areas such as computer graphics [2], [3] and real-time programming [7], is often done at a level close to assembly language. Some attention has been paid in the past to higher level input primitives [21], [4]. A systematic and hierarchical approach to building higher level input functions based on a set of input primitives was until recently [1], [8], [11], [17], [18] lacking.

The following sections describe such an approach. Section II describes language facilities based on the input-output tool model (IOT), an elaboration and refinement of our 1978 input tool model (see also [14]). This presentation is then illustrated by means of several appropriate examples in Section III. Section IV discusses additional facilities of the IOT model as well as problem areas. In addition, several completed implementations are reported. Finally, a comparison is made with recent related approaches.

II. THE INPUT-OUTPUT TOOL MODEL

A. Overview

An input-output tool (IOT) is a named, typed object. When active it may be triggered by a predefined pattern of inputs to execute an internal procedure and to produce output. An IOT, or tool, for short, may represent any physical input device, as well as abstract input devices or functions. In using these objects as a language facility, an interactive program is defined as an IOT consisting of a hierarchy of lower level IOT's. The latter are either defined internally or inserted at compile or link time by a (macro) call facility from a library. A general input-output tool consists of five parts:

- init section,
- input rule,
- cleanup section,
- tool body,
- output rule.

As far as the syntax is concerned, the only part required is an explicit input rule. Without it the IOT would reduce to a mere procedure. The other parts are optional, although they are semantically implicitly present. The init section, the cleanup section and the tool body contain executable code. The input rule contains an input expression; this expression must not be empty. The expression specifies the input pattern capable of triggering the tool. The output rule specifies the type and order of the information produced when a triggered tool completes. There is some resemblance between the output of an input-output tool and messages.

Corresponding with the five tool parts just mentioned, five

partially ordered steps may be distinguished. These steps constitute the life of an active input-output tool.

init: The init section performs initialization functions for the tool.

input: Incoming input tokens are compared with the specification in the input rule. As long as the input matches, the comparison proceeds. If the entire input rule is *matched* the tool is triggered, otherwise the tool *fails*.

cleanup: If the input rule of a tool has been matched, then the cleanup sections of the tools which are alternatives to this tool (see below) and which fail, are executed. All failing tools now *terminate* without producing output.

tool body: If the input rule of this tool has been matched it triggers the fourth step, the execution of the procedural part, the tool body. Part of its semantic action is usually the preparation of the output information.

output: In the last step the output information is made available to the successor of this tool in the input expression designating this tool. The information is transmitted via parameters specified in the output rule. This step *completes* the tool and deactivates it.

Tool names may be qualified by means of instance parameters in square brackets, these parameters are read-only. Among other things qualification allows the creation of instances of a family of tools with related properties, for example an array of IOT's.

Before going into more detail, we first illustrate the above by presenting an example of an input-output tool. It is called 'Sample' and consists of a coordinate digitizer equipped with a ready switch. Depression of this switch causes the tool 'Sample' to produce one sampled coordinate pair via its output rule. One way of representing this physical device in an input-output tool environment might be the following:

```

tool Sample = input Readyswitch end
                output (int x, y) end
    x := REGX; # copy hardware registers #
    y := REGY  # X and Y of digitizer    #
end

```

This tool consists of three parts: the input rule, the output rule, and the tool body consisting of two assignment statements. Informally this is what happens. When tool 'Sample' is activated, first its init section (here absent) is executed. Subsequently, it waits for a signal (input token) from 'Readyswitch.' When this signal occurs the input rule of tool 'Sample' is matched. As a consequence the tool body is executed, copying values from hardware registers to variables. Finally, these variables are made available through output parameters 'x,y' as indicated in the output rule, between the parentheses. This act completes and deactivates the tool 'Sample.' The input expression in this case was simple, it consisted of only a single input source.

We now proceed by embedding the tool 'Sample' in a higher level tool 'Locator' which returns so-called normalized coordinates, i.e., values between 0 and 1, inclusive. In the example it is assumed that the minimum and maximum device coordinates are known (predefined in its init section). The re-

sulting tool 'Locator' (of level depth 2) is given by the following definition:

```

tool Locator = input Sample (xx,yy) end
               output (real xn,yn) end
  int xx,yy,xlo,ylo,xhi,yhi;
  tool Sample = input Readyswitch end
               output (int x,y) end
    x := REGX;   # copy hardware registers #
    y := REGY    # X and Y of digitizer    #
  end
  init xlo := ..., ylo := ..., xhi := ...,
        yhi := ...
  end
  xn := (xx-xlo)/(xhi-xlo); # tool body #
  yn := (yy-ylo)/(yhi-ylo) # of Locator #
end

```

When the tool 'Locator' is activated (how this actually takes place is explained in Section II-C), its init section is executed, initializing the four device constants 'xlo,ylo,xhi,yhi.' Following this the tool designated in the input rule of 'Locator' (input-output tool 'Sample'), is activated. The consequences of this activation were described above. We skip this part and take up again when 'Sample' has produced its output and completes. This completion causes the input rule of tool 'Locator' to be matched. As a result its tool body is executed. This tool body computes the normalized coordinates and assigns them to the output parameters. Once this output becomes available, IOT 'Locator' completes and is deactivated.

Note that 'Readyswitch' is treated as an implicit input-output tool. Such tools are called *basic* input-output tools and are to be considered as 'given' input-output tools (see Section II-G).

B. Input Rule Expressions

The input rule is a prominent part of an input-output tool. Via its input expression it specifies one, or sometimes several alternative input patterns or sequences, at least one of which has to be matched by the actual physical input in order for the tool body to be executed. The input expression is an expression over tool designators (also called input sources) using the following operator set:

```

$ 'exponentiation'
* <>
;
&
?
,
+

```

All operators, except the ones in the top row, are dyadic. In the vertical direction downward, operator priority decreases; in the horizontal direction operators have the same priority. Parentheses may be used to override default priority rules. Otherwise, left-to-right evaluation applies.

Operating on operands symbolized by 'T' and 'V,' the operators produce regular expressions. Only part of the operators

cannot be expressed in finite combinations of each other. They are considered primitives. Acting on operands these operators generate expressions composed of the following constituents:

repetition	T \$	—an infinite sequence of symbols 'T'
range	T * V	—a sequence of zero or more symbols 'T' followed by 'V'
sequencing	T ; V	—a sequence consisting of 'T' followed by 'V'
interleaving	T & V	—a sequence of 'T' and 'V,' order being irrelevant
significance	T ? V	—the left part of the sequence defined by expression 'T,' when terminated by 'V' and significant
selection	T + V	—either 'T' or 'V'

When occurring in an input expression the symbols 'T' and 'V' may denote tool designators as well as tool designator expressions. The occurrence of a tool designator, or input source, 'T,' in an input expression is to be interpreted as input from the source, or the designated tool, 'T.' Because 'T' may denote a hierarchical input-output tool, the input from 'T' may be quite complex.

For the **repetition** operator '\$' this means, for instance, that 'T\$' is to be interpreted as an infinite sequence of (possibly compound) input from 'T.' The expression 'T&V' for input is to be interpreted as the **interleaved** or parallel reception of (possibly compound) input from sources 'T' and 'V.' The **significance** operator '?' indicates that the input corresponding to expression 'T' may be considered to have occurred when in fact only a part of the sequence (directly) defined by expression 'T' has physically occurred, followed by input corresponding to 'V' (the terminator). But only when the partial sequence so far received is unique w.r.t. other active sequences. For example, an input expression 'e;n;d?return' for a tool 'A' is equivalent to 'e;return + e;n;return + e;n;d;return,' but only when the actual input tokens uniquely determine 'A.' The **selection** operator '+' is used to define alternatives among a number of input sources.

In the interactive practice other operators in addition to the primitive operators defined above are desirable. Although the intention is to eventually provide a general facility for defining new operators, at this moment only the following nonprimitive operators are valid:

exponentiation	$T^n \equiv T; T; T; \dots; T$	'n' symbols 'T'
range	$T^{i \dots k} V \equiv T^i; V + T^{i+1}; V + \dots + T^{k-1}; V + T^k; V$	with $0 \geq i > k$
option	$\langle T \rangle V \equiv V + T; V$	

The **exponentiation** operator is used to indicate a fixed number of symbols; the exponent is evaluated once when encountered during parsing. The **range** operator '*' indicates a number of symbols, where the number has to fall within a range. The range limits are evaluated once when encountered during parsing; from then on they remain constant until the

required number of symbols has been received. An infinite upper limit of a range may be indicated by '\$.' The default range for the range operator is $i=0$ and $k=.$. This defaulted operator is frequently referred to as the Kleene star operator. Finally, there exists the ',' operator. It is a variant of the selection operator and is presented in Section II-E.

To add to the power of input expressions conditionals have been introduced. They occur as prefixes or postfixes to tool designators or tool expressions. The conditionals allow for case constructions and conditional loops in input expressions. They are discussed in Section II-E.

Recursive tools are disallowed, and Algol 60 scope rules apply. The fact that the operands of the input expression designate tools, indicates that each tool defines a tree-structured hierarchy of input-output tools, although the same IOT may be designated in more than one branch of the tree. In this way an entire program is defined as an input-output tool, or to be more precise as a tree of tools, the root of which may be considered as the name of the tool program. Starting a tool program is equivalent to activating the highest level input-output tool. A single parser, which is the intermediate acceptor of all input tokens, controls the tool program.

C. Activation and Matching

As stated above the input expression specifies in general a set of (allowable) input patterns, or sequences. An input sequence consists of a sequence of tool designators or input sources. For example, if the input expression reads 'A;B;C + X;Y,' there would be two sequences, namely 'A;B;C,' and 'X;Y.' Upon activation of an input-output tool the init section is executed and the first tool designators of all sequences defined by the input expression of this tool become candidates for matching with input tokens from the input stream. In our example these first tool designators would be 'A' and 'X.' Because each input source, which has just become a candidate for matching, itself designates a tool (at the next lower level), the designated tool(s) will also be activated, and so on. This chain of activations propagates until the bottom level of the tool tree, the basic tools (the leaves), has been reached. This is the normal way tools get activated, the only apparent exception is the activation of the tool program itself.

The tool designator sequences are scanned from left to right by the parser. If the input stream is empty the parser waits until some input token arrives. The parser then removes the first token from the input stream and tries to match it against the tool designator(s) most recently activated. If no match can be made, the input token is rejected (thrown away). In any sequence in which the active input source can be matched, the parser advances to the next tool designator which becomes candidate for matching. If in our example 'X' would have been matched, the new candidate would be 'Y.' The parser activates the tools designated (in our example 'Y,' and at lower levels the heads of the input sequences defined by 'Y,' etc.), and waits for the next input token.

An input rule is matched when an entire input sequence defined by the input expression of that input rule has been

matched. When this match has been made, the cleanup sections of the unmatched tool alternatives are executed, usually in order to undo initializations. These alternative tools now terminate and are deactivated. Sequences in which the active tool designator could not be matched are discarded. In our case, discarding amounts to the following: just after matching 'X,' the cleanup section of 'A' would be executed, 'A' would terminate, and the sequence 'A;B;C', and its descendants would be dropped. Any sequences in which the active tool designator is not an ancestor of the designator just matched are also discarded. Following these cleanup operations the tool body of the tool with the matched input rule is executed. This tool completes by making its (optional) output parameters available, and is then deactivated. The completion of a (lower level) tool with a matched input rule, serves as a symbolic input token for the next higher level input sequences, and may in turn cause input rules to be matched, tool bodies to be executed, and so on. Empty input rules, normally caused by semantic action on prefixes, cause the tool to fail (see Section II-E).

Output rule execution is formally the last phase of an active, matched tool. In many cases, however, no explicit output parameters are transmitted. In that case the output rule may be omitted entirely, or written with an empty parameter list.

D. Example

A tutorial example of a tool with internal tool definitions and a simple expression in the input rule is given by the following tool. In this example we assume three basic tools: 'Inputa,' 'Inputb,' and 'Inputc.'

```

tool High = input LowA(p);LowB(q) + LowC(p,q) end
              output (int r) end

int p,q;
tool LowA = input Inputa end
              output (int a) end
              init prompt('LowA ready to go') end
              cleanup remove prompt end
              remove prompt; a:=4
end
tool LowB = input Inputb end
              output (int b) end
              init prompt('lowB ready to go') end
              cleanup remove prompt end
              remove prompt; b:=5
end
tool LowC = input Inputc end
              output (int a,b) end
              init prompt('lowC ready to go') end
              cleanup remove prompt end
              remove prompt; a:=-4 ; b:=-5
end
init prompt('%') end
cleanup remove prompt end
remove prompt;
r := p+q
end

```


In the following explanation of this example it is assumed that there are no other active tools in the (surrounding) program. When input-output tool 'High' becomes active, its init section is executed: a '%' is printed to prompt the user. Subsequently tools 'LowA' and 'LowC' become active. Their init sections are executed. As a result both tools display a message, in indeterminate order, to the effect that they are ready to go. After this moment only sources 'Inputa' and 'Inputc' become eligible for input matching. Because we impose sequential ordering on incoming inputs, either 'Inputa' or 'Inputc' will be accepted. Suppose 'Inputa' is the first to arrive (if 'Inputc' arrives following 'Inputa,' it is thrown away in the next matching step by the parser): 'Inputa' matches the input rule of 'LowA.' First the cleanup section of 'LowC' is executed resulting in removing the prompting message related with this tool. Tool 'LowC' now terminates, and the tool designator sequence starting (and ending) with 'LowC' is discarded. Subsequently the tool body of 'LowA' is executed, as a result the prompt message is removed and the output parameter 'a' gets the value 4. Tool 'LowA' is inactivated and, according to the input rule of tool 'High,' 'LowB' becomes active, resulting in the prompting message 'low B ready to go.' Now the parser waits for 'Inputb.' After it arrives the prompt message for 'LowB' is removed, and the output parameter 'b' gets the value 5 in the tool body of 'LowB.' At this time the entire input rule of tool 'High' has been matched. As a consequence the tool body of 'High' is executed, resulting in cleaning up the remaining prompt messages, adding the parameters 'p(=a), and 'q(=b),' and assigning the result of this addition (9) to output parameter 'r.' This completes and deactivates tool 'High.' If instead of 'Inputa,' 'Inputc' had occurred, IOT 'High' would have produced -9 as a result.

E. Prefix and Posttest

So far we only have discussed and shown regular input expressions. In order to add power, the IOT model contains two isolated constructs through which executable parts of a tool may exert influence, albeit carefully controlled, on the input specification.

A tool occurring as an operand in an expression, or the expression itself, may be preceded by a *prefix* function, equivalent to a guard [6]. The output parameters of a tool designator may be subjected to a test, the *posttest*. Prefix and posttest functions are evaluated by the input expression parser.

The prefix function may be in-line or defined elsewhere. It must return a Boolean value, and it must not have any side effects. Its syntax is

prefix: Tool expression

The semantics for the prefix function is that the tool expression can *only* become active when the prefix is true. If the prefix evaluates to false the expression in effect becomes empty. If the empty expression constitutes the entire input expression, the tool possessing this input rule fails. Otherwise, if an empty designator occurs in the sequence this sequence is no longer a candidate sequence for matching. As a result it is discarded, which may result in failure of higher level tools.

The syntax of the posttest or postfix function is

Tool: postfix

As with the prefix, the postfix function must return a Boolean value and not have side effects. As an example assume a tool that only accepts a letter 'E' from a keyboard. It uses a basic tool 'Key' that we assume produces a single character as output:

```
tool Key_E = input Key(char c) : c='E' end
..
end
```

A postfix consisting of a simple logical expression may also be written in shorter form, as illustrated in a variant of the previous tool 'Key_E':

```
tool Key_E = input Key(char c='E') end
..
end
```

A slightly more complex example is a tool that only accepts (and returns) characters corresponding with numerical key strokes on an alphanumeric keyboard, by checking whether the character returned by basic tool 'Key' falls in the range of 0 to 9, inclusive:

```
tool Numeric = input Key(char c∈{'0'.. '9'}) end
..
output (c) end
echo(c)
end
```

The semantics of the posttest is that if the tool, designated by the operand subject to the posttest, has completed, the test on the output parameters is evaluated. In the case of a failing tool due to an unsuccessful test, the tool, after cleanup, terminates when an active alternative exists with a successful test. In this case the sequences containing the tool designator with the unsuccessful test will be discarded (compare Section II-C). If no successful alternative exists the failing tool(s) and the descendants, following cleanup and termination, are activated again. If the test succeeds the tool designator in the input sequence is matched.

The introduction of prefix and posttest functions provide a formidable amount of power for the IOT model. They increase the power of the languages specifiable by input expressions far beyond regular languages. This is especially welcome because any realistic interactive discourse will certainly go beyond regular languages. For example, in the following expression the alternative that becomes active depends on the prefix:

```
test1 : A + test2 : B + test3 : C
```

Alternatives may also be indicated with an 'otherwise' option, as follows:

```
test1 : P, Q
```

meaning: if test1 is true then 'P' else 'Q.' This expression can only fail when 'Q' fails. More than one prefixed term may be present, but only one else clause may be given; all terms are

separated from one another by means of the ',' operator. This operator again has been introduced for reasons of convenience, it can be entirely defined in terms of the selection ('+') operator, e.g., for the example above as follows:

$\text{test1:P, Q} \equiv \text{test1:P} + \sim \text{test1:Q}$

Prefixes are frequently used in combination with the '\$' operator to provide a conditional looping construct, as follows (with optional parentheses):

(prefix: Tool) \$

meaning: while 'prefix' is true, 'Tool' can be activated. If 'prefix' becomes false the repetition ends. This construct is called conditional repetition. It can be shown that any input expression with a range operator ('*') in it, may be replaced by a conditional repetition and a slightly different tool body (i.e., including the setting of a Boolean variable), to produce the same output parameters. The conditional repetition is very powerful in combinations such as

(test1:A + test2:B) \$

If the tool bodies of both 'A' and 'B' influence 'test1' and 'test2,' then this expression may accept any number of 'A's (including zero) interleaved with any number of 'B's. As a special case this could be used to provide the context-free language $a^n b^n$. Three prefixed tools would allow a context-sensitive language such as $a^n b^n c^n$, as demonstrated in the following example:

```

tool AnBnCn = input (alpha:As + beta:Bs + gamma:Cs) $ end
               output (string sentence) end

init na,nb,nc; bool alpha,beta,gamma;

tool As = input Key(char c='a') end
         echo(c);
         sentence:=sentence+c; na:=na+1;
         beta:=true
end
tool Bs = input Key(char c='b') end
         echo(c);
         sentence:=sentence+c; nb:=nb+1;
         alpha:=false;
         if nb=na then beta:=false; gamma:=true fi
end
tool Cs = input Key(char c='c') end
         echo(c);
         sentence:=sentence+c; nc:=nc+1;
         if nc=nb then gamma:=false fi
end
init alpha:=true; beta:=gamma:=false;
         sentence:=''; na:=nb:=nc:=0
end
end

```

F. The Escape Tool

At different tool levels in an input-output tool program the programmer may define a so-called escape tool. It uses the reserved tool keyword **escape**, but behaves in most other re-

spects like a normal tool. This tool allows interruption of (the processing of) an input string which is erroneous according to the user, but has been accepted by the parser so far. After the cleanup sections of all active tools have been executed the parser will backup to the beginning of the level at which the escape tool was defined. That is, it will restart at the beginning of the input rule of the tool immediately surrounding the escape tool. Alternatives at a higher level no longer play a role. This implies that the tool(s) designated at the beginning of the present input expression will be activated again. In the tool body of an escape tool the programmer may provide code to undo the undesired changes of state. We shall illustrate the appearance and operation of the escape tool by means of the following example:

tool Z = **input** X + Y **end**

string buf;

tool X =

input ltr['a'];ltr['b'];ltr['c'] + ltr['d'];ltr['e'] **end**

char sym;

escape tool = **input** Key(sym='@') **end**

buf:=''

end

tool ltr [char s] = **input** Key(sym=s) **end**

buf:=buf+s

end

end X

tool Y = ...

:

end Y

init buf:='' **end**

end Z

The escape tool is local to tool 'X,' so it can only be active when 'X' has been activated. In our example 'Y' is at first an alternative to 'X.' Tool 'ltr' stands for a family of tools. A particular actual instance parameter signifies the letter to be read. This letter is concatenated to a buffer 'buf.' Suppose tools 'ltr['a']' and 'ltr['b']' have completed. Also assume these two tool completions uniquely determine the alternative 'X' instead of 'Y.' If at that moment the '@' key is depressed, the input rule of the escape tool is matched. Its tool body is executed, resulting in emptying 'buf,' after which it completes. The cleanup section of 'ltr['c']' and 'X' are executed, and the parser backs up to the beginning of the input rule of 'X,' initializing the tools 'ltr['a'],' and 'ltr['d']. 'Y' no longer plays a role: to return to the beginning of tool 'Z' an escape tool at the same level as 'X' and 'Y' would be needed. Notice that any side effects resulting from the completion of tools 'ltr['a']' and 'ltr['b'],' or from the init or cleanup sections of 'X' and 'Y,' will remain in force if not explicitly undone.

In a single IOT several escape tools may be active simultaneously at different tool levels. In this way one may fine-tune the amount of backing up. For the parser the escape tool is more or less considered as a permanently active alternative

('+' operator) to every single operand in the input expression of the immediately surrounding tool.

Although it may be considered not part of the IOT model, it appears advisable to enter escape tool mode by means of an escape trigger. This trigger is defined as a basic escape tool such that its input stimulus is unique and unambiguous for a particular programming environment. For instance the 'ESC' key of a keyboard could be used for this purpose. In practice this would rule out its use for other purposes (input rules).

G. Discussion

In the IOT model basic input-output tools play the role of terminal symbols, similar to terminals in a grammar. The basic tools thus constitute a set of primitives upon which all other input-output tools build. The exact nature of basic input-output tools and how many basic tools exist, is irrelevant for the IOT model. These tools may correspond to physical devices, or to one or more devices wrapped in a layer of software. The set of basic tools also depends on the application surroundings. In an interactive graphics environment the basic set would presumably be different from the basic set in a process control context. Notice also that the functional level of the basic tools in one environment may differ sharply from that level in a different environment. Compare for instance hardware interrupts and commands (in a command language processor). They may both act as primitives, though in rather different surroundings. An interface between input devices and basic input-output tools is presented in [14].

Notationally, the input expressions of IOT are related to the path expressions introduced by Campbell and Habermann [5]. In general path expressions are used to specify the flow of use of abstract objects. As such they play a crucial role in mediating resource control. The input rules in a tool program specify the flow of use of tools; syntactically the rules specify the input language. The input rule of a tool can be considered as the right-hand side of a special kind of grammatical production rule, the tool itself as the left-hand side. Higher level tools correspond to nonterminal symbols, basic tools to terminal symbols. The highest level tool (the program) corresponds to the start symbol of the grammar. The instance and output parameters can be considered as attributes (affixes) of the terminals and nonterminals. IOT appears to be closely related to a subclass of affix grammars [9], namely the class of grammars which can be parsed top-down from left to right in one pass.

Hence, input tool program can be considered as a very special kind of a parser generator. The main difference between a parser thus created and a normal (generated) parser is that the parsing method is especially geared towards an interactive environment. When during the parsing process more alternatives are possible all these alternatives have to be evaluated before the next input token can be matched against a terminal symbol. Only in this way all the corresponding tools can be determined and activated. Activation of all alternatives is important since a tool may have a prompt specified in its init section which normally tells the user that the tool is an outstanding alternative. For a good dialogue it is important that

all outstanding tools have a possibility to introduce themselves to the user.

Another unusual aspect of the generated parser is the handling of errors. Thanks to the interactive environment an incorrect input token can just be thrown away. The parser simply reactivates the previously activated tools in order to let these tool prompt the user again. This is one way for the user to find out that he has done something wrong, so that he can do better the next time.

There is some similarity between the IOT model and a stimulus-response model as used in psychology. The input rules of input-output tools in hindsight also show a more than superficial resemblance to hierarchic control languages over production systems as employed in artificial intelligence [13]. Simple IOT's, in particular tools triggered by a single stimulus, may be compared with interrupt handlers in operating systems. IOT's span a wide range of applications. On the one hand, they may be used as a hardware description language, e.g., at the microcoding level. On the other hand, they are very flexible constructs to be used in high-level command languages. Some examples in the intermediate range will be presented next.

III. APPLICATION OF INPUT-OUTPUT TOOLS

In the following we present examples showing the application of IOT's. In particular, the hierarchic build-up of tools from simpler tools is shown. In addition, examples of increasing complexity and power demonstrate the use of several of the allowed operators, as well as the use of prefixes and post-tests. The examples also purport to show the programming style that is fostered by the IOT facilities.

A. Keyboard

This tool builds on the basic tool 'Key,' and uses a buffer to collect the characters of the string typed in from the keyboard:

```

tool Keyboard = input Keys * Return end
               output (string buf) end

char sym;
tool Keys    = input Key(sym≠CR) end
               output ( ) end
               buf := buf+sym # concatenate sym to buf #
end
tool Return = input Key(sym=CR) end
               output ( ) end
end
init buf := ' ' end
end

```

The input rule of 'Keyboard' specifies by means of the range ('*') operator that it wants zero or more inputs from tool 'Keys,' concluded by one input from tool 'Return.' Tool 'Keys' specifies it wants input from basic tool 'Key,' but only if the character is not a carriage return (indicated by the symbol CR). When tool 'Keyboard' is activated its init section is executed, clearing the buffer. Then both tools 'Keys' and 'Return' are activated (note the range operator following 'Keys'). The parser now awaits input from basic tool 'Key.' If

such input is received the parser executes the posttests. In this case one of the posttests must be true, but if a situation arose where none of the posttests succeeded, the input would be discarded. If the posttest of 'Keys' succeeds, the input rule of 'Keys' is matched and consequently the tool body of 'Keys' is executed: 'sym' is concatenated to the buffer. This process would continue, collecting all incoming characters into buffer 'buf,' until eventually a CR is typed in. This action would satisfy the posttest of tool 'Return,' complete 'Return' and because of its completion cause the matching of the input rule of tool 'Keyboard.' The latter tool in turn would complete and be deactivated. The output rules in tools 'Keys' and 'Return' are not required, they merely pass an empty parameter list to tool 'Keyboard.' They are included for illustrative reasons only.

B. Rubber Band

In freehand drafting on a computer graphics display device it is often desirable to draw a line by first fixing the beginning point, and then trying several positions for the end point. Every try is visually illustrated by a temporary line. This facility is called "rubber banding." The next tool provides this facility. When active it puts a cursor in a predefined location on the screen. The user moves the cursor to the beginning point of the line. When this point has been determined he depresses button 1 (a programmable function key). He subsequently moves the cursor to a temporary end point for the line. The tool follows the cursor movements with a maximum line drawing rate of approximately 10 per second. When finally satisfied, the user pushes a button (here number 2) which fixes the endpoint. In addition to the drawing actions the tool returns the coordinates of beginning and end point to the user in the user coordinate system, the limits of which are passed to the tool through range variables 'xr' and 'yr.' The tool 'Rubberband' uses the basic tools 'Button' and 'Clock,' as well as the tool 'Locator' defined in Section II-A. Tool 'Button' represents the family of programmable function keys of which in this case instances 'Button[1]' and 'Button[2]' are used. Tool 'Clock [time]' is a timer for an interval of length 'time'; it generates an input event when the interval expires:

```
mode range = [1:2] real;
tool Rubberband [range xr,yr] =
  input Beginpoint(real xb,yb); Endpoint(real xc,yc) end
  output (real uxb,uyb,uxe,uye) end
  bool line present;
  tool Beginpoint = input Locator(real x,y) * 1..$Button[1]
    end
    output (x,y) end
  end
  tool Endpoint = input Band(real x,y) * 1..$Button[2] end
    output (x,y) end
    tool Band = input Clock[0.1]; Locator(real x,y) end
      output (x,y) end
      if line present then delete line fi;
      move(xb,yb); draw(x,y); line present := true
    end
```

```
remove cursor
end
init line present := false;
insert cursor (0,0)
end
cleanup remove cursor end

# scale to user coordinates #
uxb := xb*(xr[2]-xr[1])+xr[1];
uxe := xe*(xr[2]-xr[1])+xr[1];
uyb := yb*(yr[2]-yr[1])+yr[1];
uye := ye*(yr[2]-yr[1])+yr[1];
```

end

C. Integer Numbers from a Keyboard

This tool reads in a signed or an unsigned integer string from a keyboard and returns the integer representation of the string. Error checking and conversion is completely self-contained:

```
tool Number =
  input Digit * Return + Sign; Digit * 1..$ Return end
  output (int integer) end

  char c; string xalpha;
  proc convert = (string numerical) int :
    begin # conversion code #
    end

  tool Sign = input Plussign + Minussign end
    tool Plussign = input Key(c='+') end
    end
    tool Minussign = input Key(c='-') end
      xalpha := '-'
    end
    echo(c) # echo to user #
  end

  tool Digit = input Key(c ∈ {'0'.. '9'}) end
    xalpha := xalpha+c; # append c to xalpha #
    echo(c)
  end

  init xalpha := '' end

  integer := convert(xalpha)
end
```

This is an example of a hierarchy of three tool levels. At the highest level we note that tool 'Number' is satisfied by either zero or more digits (unsigned number), or a sign followed by at least one digit (signed number). In both cases this is followed by a carriage return from tool 'Return' as defined in Section III-A. Notice that a single CR satisfies the tool. On the second level, the tool 'Sign' is constructed from the tools 'Plussign' and 'Minussign.' Their operation is not identical: 'Plussign' reads the plus sign but does not do anything with it, while 'Minussign' puts the minus sign in the (still empty) string 'xalpha.' Once a sign has been accepted, at least one digit (by way of tool 'Digit') should follow. Every digit read

is appended to 'xalpha.' Only characters accepted are echoed to the user ('echo' statement).

When the outermost input expression (of tool 'Number') is satisfied the corresponding tool body is executed. Here this means that the completed string 'xalpha' (possibly empty) is converted to an integer.

The advantage of the tool approach is here particularly evident. Most of the input handling, including checking for illegal characters (which are thrown away) is done by the parser. Because of it, the tools and subtools need only have simple bodies. The conversion procedure 'convert' can be much simpler than in traditional programming: the string argument it receives is by definition correct. As far as the tool user is concerned this tool really reads an integer number and presents the value to him.

If the programmer wants to exert more control over the incoming characters, tool 'Number' should be designated such that instead of having the parser do the testing for valid characters, the tool bodies should do this. This would allow, for instance, the tool to issue error messages to a user who enters incorrect symbols repeatedly. Alternatively, we could leave the input rules alone, and keep a count of tool activations in the init section. When a certain number of reinitializations would be exceeded, a detailed prompting message could be generated.

D. Pressure Control

For the next example we adapt a program, originally written in Pearl [7], controlling two pressure meters, to our IOT model. The original program was what we might call a traditional solution to this problem, at least as far as input handling is concerned. One task uses the read-out to control the one pressure by opening or closing a valve. The other task (in the same program) informs the (in all likelihood human) plant operator by means of a console message, whenever the second pressure gets too low or too high.

We use two basic tool families, 'Clock' and 'Pressgauge.' 'Pressgauge' is a meter measuring the pressure in a certain channel, 'chan.' It returns a read-out when sampled:

```

tool Control = input Job1 & Job2 end

  tool Pressure [int chan] =
    input Pressgauge[chan] (dial) end
    output (real val) end
    int dial; real a1 = ..., b1 = ...;
    # get reading and convert #
    val := dial * a1 + b1
  end

  tool Job1 = input (Clock[1]; Task1) $ end

    tool Task1 = input Pressure[1] (value) end
      real value;
      if value > 2 then close valve
      elif value < 1 then open valve
      fi
    end
  end

```

```

tool Job2 = input (Clock[30]; Task2) $ end

  tool Task2 = input Pressure[2] (value) end
    real value;
    if value > 3.5
      then display msg (Time, 'uplim=', value)
    elif value < 0.5
      then display msg (Time, 'lowlim=', value)
    fi
  end
end

```

This tool takes the following actions. Once a second read the pressure in channel 1, open or close a valve depending on this reading. Every 30 s measure the pressure in channel 2, and report to the operator when certain limits are exceeded (the function 'Time' puts a time-stamp in the message).

Due to the '&' operator the input rules for 'Job1' and 'Job2' are simultaneous candidates for matching. This matching may occur in an interleaved way. 'Clock[1]' and 'Clock[30]' are different tool instances and represent therefore different interval timers. The assumption in this example is that tool body execution time is small with respect to 1 s.

E. Displaying the Function $Y = \text{Sqrt}(X)$

Input-output tools may also be flexible constructs to be applied when during a normal course of actions a special situation arises in which user intervention is desirable or even necessary. Such a situation is illustrated in the following example. It depicts a program segment which interactively displays a function 'y=sqrt(x),' using data entered by a user from a keyboard. If the user errs and keys in a negative number the system will prompt him for corrective action. Either he types in 'Y' upon which the system takes the absolute value of the negative number, or he types 'N' and the system will discard the number. Tool 'Ysqrtx' completes after some 'Button' on a function keyboard has been hit. The tool uses the tool 'Number' defined in Section III-C, as the equivalent of a basic tool:

```

tool Ysqrtx =
  input (Func; error; Correct) * Button[k] end
  real x,y; bool error;

  tool Func = input Number (int val) end
    init error := false end
    x := val;
    if x < 0 then error := true
    else y := sqrt(x); display point(x, y)
    fi
  end

  tool Correct = input Accept + Reject end
    char c;
    tool Accept = input Key(c='Y') end
      x := -x;
      y := sqrt(x);
      display point(x,y)
    end
  end

```



```

    tool Reject = input Key(c='N') end
    end
    init prompt('x<0; Want abs(x)?') end
    cleanup remove prompt end
    remove prompt
  end
end

```

F. Menu Handling

The final example shows a facility to fill out forms by means of a frame containing questions to be displayed on a screen. These so-called menus find wide application in screen oriented dialogues because they guide the user through an application and at the same time they significantly decrease the possibility of user errors. For this application the displayed frame looks like:

REGISTRATION FORM	
Name:	
Age group:	* Adult * Minor
Occupation:	
* Form Done	* End Program
* Cancel	

For the purpose of this application the name and age group are always required. If the age group is adult then the occupation is also required, for nonadults the occupation is optional. Form completion ends when menu option 'Form Done' is selected. Subsequently, the information entered is written to a file, the form is cleared, and the screen is ready for the next form. If a mistake is made while working on a form the 'Cancel' option may be selected to restart. Processing ends when the option 'End Program' is selected. Selection is done by means of a lightpen, represented by the basic tool 'Pick' which returns the string selected. This example also uses the tool 'Keyboard' defined in Section III-A:

```

tool Registration = input Frame * End end

tool Frame = input (Name & Age group);
               adult: ( Occ;Done) ,
               (<Occ> Done)
  end

string str, name, occupation, group; bool adult;

tool Name = input Keyboard (string buf) end
  name := buf
end

tool Age group = input Grownup + Young end
  tool Grownup = input Pick (str='Adult') end
    group := 'A'; adult := true
  end

```

```

    tool Young = input Pick (str='Minor') end
      group := 'M'; adult := false
    end
  end
  tool Occ = input Keyboard (string buf) end
    init occupation := '' end
    occupation := buf
  end
  tool Done = input Pick (str='Form Done') end
  end
  file(name,occupation,group); clear form
end Frame;

tool End = input Pick (str='End Program') end
  clear screen
end

escape tool = input Pick (str='Cancel') end
  clear form
end

init display form; display menu end

end

```

When tool 'Registration' is activated, the execution of the init section puts a blank form at the top of the screen and a menu at the bottom. All items in the menu may be selected by means of the lightpen. The 'Name' and the 'Occupation' items on the form have to be filled out by keying in some string, the 'Age group' item is selected with the lightpen, hence the form also contains a menu. From the input rule of tool 'Frame' one sees that name and age group are required before a possible occupation is entered. Name and age group may be entered in an interleaved way. For instance, one may partially key in the name, then select the applicable age group option, followed by the completion of the name. The age group option selected determines the setting of the switch (prefix) 'adult.' If it is true occupation is required followed by the 'Done' option, otherwise the occupation is optional. If the entire input rule of 'Frame' is matched the tool body of 'Frame' is executed: the filing operation and clearing the form. The entire sequence of actions may now be repeated for a next form or the program may be terminated by means of tool 'End.'

This example also shows the use of the escape tool. It becomes active as soon as tool 'Registration' has been initialized. When the input rule of the escape tool is matched (option 'Cancel') processing is interrupted and restarts at the beginning of the input rule of tool 'Registration.' If 'Frame' and 'End' had possessed init sections these would be re-executed. Selecting the escape tool only backs up to the beginning of the input rule of the tool immediately surrounding the escape tool, in this case the tool 'Registration.'

IV. CONCLUDING REMARKS

A. Libraries and Simulation

The use of a library facility in combination with the input-output tool language facilities enhances the power of the IOT model. The library could serve as a depository for tools that

are of general applicability. A good example is the input tool 'Number' defined in Section III-C. In many applications a tool such as this one appears useful.

A library could also serve to further standardization on a single set of logical input devices. One could have a standard collection of locally used logical devices in the system tool library while at the same time having special versions of some logical devices in one's private tool library. This combination appears to offer standardization while simultaneously retaining flexibility [14], [20].

An additional, very interesting application of a library is to simulate basic input tools that are desired but not part of the available interactive equipment. Libraries provide a flexible solution to this problem. The system tool library may provide a standard set of simulated tools which can be overridden by simulations defined in the user's program or library [14], [20].

B. Ambiguity

It is possible that an incoming input token causes more than one active input rule operand to be matched: the tool program is (locally) ambiguous. A simple example of such a situation is given by two active IOT's with input rules 'B1;X' and 'B2;X,' respectively, where tools 'B1' and 'B2' have identical input rules but (in general) different tool bodies. It is in general not possible to postpone the execution of the tool bodies of 'B1' and 'B2,' because they may produce values needed in 'X.'

This case could in principle be detected at compile-time. If a library facility is present we could run into the same situation. It would be more difficult to diagnose because it represents an external compilation or binding-time ambiguity, but this may be solved by supplying sufficient information about all hierarchic tool names and where they occur in input expressions.

A far more complex case is run-time ambiguity, usually resulting from prefixes or postfixes with the same truth value, or from actual instance parameters with the same value. It seems that in this situation at compile or bind-time only a worst case analysis can be made by considering all possible combinations of prefix, postfix, and instance parameter values. At best this procedure results in a warning list of 'possible' ambiguities, because the semantic code may be such that in actuality ambiguity does not arise.

The general adage is not to abort a running interactive program when an ambiguity occurs. It may only occur locally and later be resolved, or happen in a branch of an IOT that is not used during this run. It may even be that the ambiguity is intentional. For instance, circumstances with any operands that have to be parsed simultaneously (with selection, interleaving or range operator), could be used to deliberately introduce nondeterminism into the system.

Therefore, ambiguous sequences are in practice maintained as long as necessary and possible, while the tool bodies and output rules are executed. If an active tool has an input expression equal to the left part of another active input expression then the shortest rule only is considered to be matched when the corresponding input stream has arrived.

C. Implementation

Because we do not wish to add to the proliferation of programming languages, even though we have stated in the beginning that there are only a few languages for interactive programming proper, we have chosen to embed the input-output tool facilities in some host language. This still leaves open the choice of an implementation language. At the moment there exist two major implementations.

1) *Ctool Implementation*: The first implementation is called 'Ctool' [14], [15]. It is a subset of IOT in the form of an extension of the Unix language 'C' [16]. User programs written in this superset are converted to 'C' code by means of a preprocessor. This preprocessor builds a parse tree which is added to the application program. Also added is a parser which, at run-time, tries to match incoming input tokens against the input sequences specified. When a match is successful the parser causes the appropriate tool body to be executed, after which control is returned to the parser to handle additional input. There is one general restriction in the 'Ctool' implementation. The absence of abstract data structures in the language 'C' dictates that no two instances of a tool may exist at a time. In our case instances may be equated with parameterized tools, consequently 'T[parm1] + T[parm2]' is not allowed, although 'T[parm1];T[parm2]' is acceptable, because in the latter case the occurrences of 'T' cannot be active at the same time.

An industrial implementation of the same IOT subset has been completed in Fortran ('Ftool'), another one is in progress in APL.

2) *Wtool Implementation*: The second implementation is a full implementation of the IOT model. It is called 'Wtool' and it is based on the languages Pascal and Modula-2 [22]. This implementation features a preprocessor and a parser in pretty much the same way as the 'Ctool' implementation does. The preprocessor compiles, depending on a user option, a Wtool program either into Pascal or Modula-2. The preprocessor accepts a subset of the Modula-2 language, supplemented with the IOT structuring concepts. The subset of the Modula-2 language was chosen such, that automatic compilation into either target language would be possible. The parser is written using the same subset of the Modula-2 language, and may therefore be run through the preprocessor to either yield a Pascal or a Modula-2 program.

The sequence of events during the execution of a tool program requires a certain degree of (possibly simulated) parallelism, since the '&,' '+,' and '*' operators require both operands to be parsed simultaneously. The coroutine concept is sufficiently powerful to achieve the required amount of parallelism. Unfortunately, no coroutine facilities are available in a standard Pascal implementation. However, it proved fairly easy, following the suggestions of Kriz and Sandmayr [10], to implement the routines 'NEWPROCESS' and 'TRANSFER' for various Pascal systems as they exist in the Modula-2 'SYSTEM' module.

Another important observation is, that the parse graph of a running tool program can only be created dynamically, since the number of instances of a tool depends on the values of the

tool parameters. The compiled tool program must therefore contain routines to dynamically generate the (sub)graph corresponding to that tool upon request of the parser. This necessitates a facility to link code to data structures, since the parser cannot know all (user written!) tools by name. The Modula-2 procedure types provide an excellent solution to this problem, but again some additional (but rather trivial) routines had to be written for the Pascal implementation.

This implementation of IOT is fully portable in its Modula-2 phenotype. The installation of the Pascal version requires a few man-days to implement the coroutines and the replacement for the procedure type facility.

D. Other Work

Work related to the handling of input at more sophisticated levels than customarily done, is reported in [11], [17], [1], and [8]. Lafuente and Gries [11] have extended Pascal to allow display screen layout and to use special display variable attributes such as 'selectable,' 'key-in,' etc. Their model also provides facilities to hierarchically build frames out of subframes. More relevant in the context of the present paper, they use a so-called 'behavior rule' in which the required input is specified together with its constraints. Ordering of input is not so clear. In principle, input is unordered, any actual ordering is controlled implicitly via the constraints. Lafuente and Gries have also studied graph-oriented methods to allow compile-time checking of ambiguities and inconsistencies in this rule [12]. Shaw [17] uses flow expressions containing powerful operators to describe the flow of control of modules which could, but need not be interaction modules. Although his flow expressions seem similar to our input expressions, they are not the outer layer of an object. There is thus no equivalent of tool initialization, cleanup, and tool body execution, and there is no information returned, other than in the modules themselves.

Anson [1] describes objects which are triggered by incoming simple signals (no expressions), perform some action, followed by the emission of an, in principle, undirected signal. This signal may be captured by any other object specifying it as a trigger. His model seems to be quite closely related to data flow models. Hopgood and Duce [8] present a preliminary model derived from techniques in artificial intelligence [13]. The model is based on rules in a production system. The rules are triggered when a set of stimuli is present. If the latter is the case an action is performed and an (optional) stimulus generated. This model appears to be related to Anson's model.

The IOT model shows considerable intersection with the above models. Compared with existing flow-of-control and data flow approaches, our model is a hybrid: on the one hand it can be used (with trivial input expressions) as a data flow model, on the other hand, one tool could do all the algorithmic action needed (including many parser functions), thus presenting a flow-of-control appearance.

E. Conclusion

We have offered a novel conceptual model for input and output in interactive and real-time program environments. It is called the input-output tool model. It provides the programmer with facilities to hierarchically construct high-level input-

driven primitives, which we have called input-output tools. These tools include a tool body representing the semantic action taken when a specified input expression has been satisfied by physical or virtual input actions. This semantic action is responsible for the progress of the program. In particular it takes care of the preparation of the output part of interaction. Complete application programs may be built in layered form from these 'interaction modules.'

The IOT model has clear advantages over traditional interactive programming practices, because

- it offers an input specification language more powerful than regular languages,
- it fosters, almost forces, structured programming,
- it is superior to the subroutine approach to input: the programmer only has to *specify* his inputs and does *not parse* them as in most current programs; the parser is generated by the system,
- it allows interleaving and parallelism in entering inputs,
- in addition to input type checking the parser may also handle value checking of input,
- abstract (or logical) and simulated devices are easily definable,
- input-output tools of sufficiently high-level may be considered as input command procedures and hence may be used as entities in a user dialogue language.

Further research is underway and partially complete in the following areas: extension of the tool model to communicating parallel tool processes [14], [19] (the input tool process model, ITP), extension of the input rule operations to simple pattern recognition operators, backtracking and ambiguity prevention (this might include a classification of tools in terms of their side effects), and allowing dynamic (user-definable) input rules at run-time.

The input-output tool model has been used in demonstration-oriented projects with very satisfactory results. Ease of programming is accompanied by efficient execution. So far no degradation with respect to traditional interactive programming parameters, such as response time, has been observed. Presently, the input-output tools are used to build a sophisticated command processor module for a large interactive pictorial information system used in environmental planning and cartography. The ease and flexibility in programming the interaction has led to a system friendly in use and adaptive to the user's wishes.

Note Added in Proof: Since the writing of this paper, interest in language constructs for interaction has been steadily growing. Some of the recent developments, although by no means complete, appear in *Computer Graphics (ACM-Siggraph)*, vol. 17, no. 1, 1983.

ACKNOWLEDGMENT

We gratefully acknowledge the very worthwhile discussions with H. Meijer, who contributed to the input rule concept; J. Stroet, who suggested important improvements; and D. Bergeron; all at the time of the Nijmegen University Computer Graphics Group. We also greatly benefited from a critical scrutiny of the original proposal by members of the Dutch Working Group on Computer Graphics.

REFERENCES

- [1] E. K. Anson, "The semantics of graphical input," *Comput. Graphics (ACM-Siggraph)*, vol. 13, no. 2, pp. 113-120, 1979.
- [2] L. C. Caruthers, J. van den Bos, and A. van Dam, "GPGS--A device independent general purpose graphic system for stand-alone and satellite graphics," *Comput. Graphics (ACM-Siggraph)*, vol. 11, no. 1, pp. 112-119, 1979.
- [3] Core, "Status report of the graphic standards planning committee," *Comput. Graphics (ACM-Siggraph)*, vol. 13, no. 2, 1979.
- [4] G.F.P. Deecker and J. P. Penny, "Standard input forms for interactive computer graphics," *Comput. Graphics (ACM-Siggraph)*, vol. 11, no. 1, pp. 32-40, 1977.
- [5] R. H. Campbell and A. N. Habermann, *The Specification of Process Synchronization by Path Expressions (Lecture Notes in Computer Science)*, vol. 16. Berlin: Springer, 1974, pp. 89-102.
- [6] E. W. Dijkstra, "Guarded commands, nondeterminacy, and formal derivation of programs," *Commun. Ass. Comput. Mach.*, vol. 18, no. 8, pp. 453-457, 1975.
- [7] J. Heger and G. Koch, *Erfahrungen bei der Erstimplementierung eines Pearl-subsets (Lecture Notes in Computer Science)*, vol. 12. Berlin: Springer, 1974, pp. 401-412.
- [8] F. K. Hopgood and D. A. Duce, "A production system approach to interactive graphic program design," in *Proc. IFIP-WG 5.2 Workshop on the Methodology of Interaction*, Seillac. Amsterdam: North-Holland, 1979.
- [9] C.H.A. Koster, "Affix grammars," in *Algol68 Implementation*, J.E.L. Peck, Ed. Amsterdam: North-Holland, 1970.
- [10] J. Kriz and H. Sandmayr, "Extension of Pascal by coroutines and its application to quasi-parallel programming and simulation," *Software-Practice and Experience*, vol. 10, pp. 773-789, 1980.
- [11] J. M. Lafuente and D. Gries, "Language facilities for programming user-computer dialogues," *IBM J. Res. Develop.*, vol. 22, no. 2, pp. 145-158, 1978.
- [12] J. M. Lafuente, "Some techniques for compile-time analysis of user-computer interactions," *IBM J. Res. Develop.*, vol. 24, no. 6, pp. 716-731, 1980.
- [13] N. J. Nilsson, *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga, 1980.
- [14] M. J. Plasmeijer, "Input tools--A language model for interaction and process communication," Ph.D. dissertation, Univ. Nijmegen, 1981.
- [15] M. J. Plasmeijer, J.W.M. Stroet, and J. van den Bos, "An implementation of high-level graphics input tools," in *Proc. Euro-Graphics Conf.*, Bologna, 1979, pp. 101-115.
- [16] D. M. Ritchie, "The Unix time-sharing system," *Commun. Ass. Comput. Mach.*, vol. 17, no. 7, pp. 365-375, 1978.
- [17] A. C. Shaw, "On the specification of graphics command languages and their processors," in *Proc. IFIP-WG 5.2 Workshop on the Methodology of Interaction*, Seillac II. Amsterdam: North-Holland, 1979.
- [18] J. van den Bos, "Definition and use of higher-level graphics input tools," in *Proc. ACM-Siggraph*, Atlanta, 1978, pp. 38-42.
- [19] J. van den Bos, M. J. Plasmeijer, and J.W.M. Stroet, "Process communication based on input specifications," *TOPLAS (ACM Trans. Programming Languages and Syst.)*, vol. 3, no. 3, pp. 224-250, 1981.
- [20] J. van den Bos, "Whither device-independence in computer graphics?," *Int. J. Man-Mach. Studies*, vol. 18, no. 1, pp. 89-99, 1983.
- [21] V. L. Wallace, "The semantics of graphic input devices," *Comput. Graphics (ACM-Siggraph)*, vol. 10, no. 2, pp. 61-63, 1976.
- [22] N. Wirth, *Programming in Modula-2*. Berlin: Springer, 1982.

Jan van den Bos was born in The Hague, The Netherlands in 1939. He received the M.S. degree in mathematics and physics in 1964 and the Ph.D. degree in 1967 from the University of Amsterdam, Amsterdam, The Netherlands.

From 1967 to 1968 he was Research Associate at JILA (University of Colorado). He has been Assistant and Associate Professor at the faculty of the Department of Computer Science, University of Nebraska, until 1976.

In 1974 he joined the University of Nijmegen, Nijmegen, The Netherlands as Head of the Computer Graphics Group. His current research interests include programming language facilities for interactive systems, interactive pictorial information systems, rastergraphics languages, man-computer communication, local area networks, and communicating parallel processes.

Marinus J. Plasmeijer was born in Hengelo, The Netherlands, in 1952. He received the B.S. and M.S. degrees in computer science from the University of Twente, Enschede, The Netherlands, and the Ph.D. degree in computer science from the University of Nijmegen, Nijmegen, The Netherlands, in 1981.

In 1977 he joined the University of Nijmegen at the Department of Computer Science. His current research interests include programming languages, programming environments, man-machine communication, communicating parallel processes, and machine architecture.

Pieter H. Hartel was born in Heemstede, The Netherlands, in 1953. He received the M.S. degree in mathematics and computer science from the Free University, Amsterdam, The Netherlands in 1978.

The next three years he worked for the European Organisation for Nuclear Research (CERN) at Geneva, as a Systems Programmer. In 1981 he joined the Computer Graphics Group of the University of Nijmegen, Nijmegen, The Netherlands. At present he is on the

Faculty of the Department of Computer Science, University of Amsterdam. His main research interests include programming languages, distributed systems, and networks.