Control Constructs in a Completely Visual Imperative Programming Language

Wayne Citrin
Department of Electrical and Computer Engineering
Campus Box #425
University of Colorado, Boulder 80309–0425

Michael Doherty and Benjamin Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

CU-CS-672-93 September 1993

University of Colorado at Boulder

Technical Report CU-CS-672-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Wayne Citrin
Department of Electrical and Computer Engineering
Campus Box #425
University of Colorado, Boulder 80309–0425

Michael Doherty and Benjamin Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

Control Constructs in a Completely Visual Imperative Programming Language*

Wayne Citrin
Department of Electrical and Computer Engineering
Campus Box #425
University of Colorado, Boulder 80309–0425

Michael Doherty and Benjamin Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

September 1993

Abstract

Visual representations of programs can facilitate program understanding by presenting aspects of programs using explicit and intuitive representations. We have designed a completely visual static and dynamic representation of an imperative programming language. Because our representation of control is completely visual, programmers of this language can understand the static and dynamic semantics of programs using the same framework. In this paper, we describe the semantics of our language, both informally and formally, focusing on support for control constructs. We also illustrate how simple programs written in this language will look both statically and dynamically. Our representation makes explicit some parts of program execution that are implicit in textual representations, thus our programs may be easier to understand.

1 Introduction

1.1 Motivation

Although visual programming languages have been a subject of research for at least thirty years, they have failed to make an impact on programming language design in proportion to the enthusiasm of the investigators in the field. Although various reasons have been advanced for this (see [2] and [5] for a discussion of problems with visual languages), one problem is that most proposed visual languages have simply been visual overlays of textual languages. Semantics of such visual languages can only be understood through reference to the semantics of the underlying textual languages. While this may sometimes allow certain patterns to become apparent (loop constructs may be visible as loops, for example), the act of constructing programs, and the reasoning behind it, is just as difficult as it would

*This material is based upon work supported by the National Science Foundation under Grant No. CCR-9208486

have been in the underlying textual language. In fact, it may have been even more difficult, because the programmer must be aware of the underlying semantics and the new visual syntax. This extra burden results in the lack of a compelling reason to adopt visual versions of existing textual languages.

There are two ways to address this problem. The first is to avoid it entirely by not basing visual languages on pre-existing textual languages. By doing this, however, one would sacrifice the extensive pre-existing body of compiler and optimization technology for a given language. The second approach is to design the visual presentations so that they may be understood in isolation from their underlying textual semantics. In such situations, a visual program and its constructs could be understood in terms of a purely visual semantics: a program could be understood and constructed through the use of visual reasoning. Relatively few visual languages have been proposed that display such properties; they are known as completely visual languages, and base their semantics on graphical transformation rules. Although some of these languages are based on an underlying textual representation, it is not necessary to understand them in such terms. A properly designed completely visual representation of a textual language could allow programmers to use visual reasoning to construct their programs (some individuals would probably prefer such an approach), yet use conventional compiler technology to produce efficient compiled code.

One problem with the design of completely visual languages is that their semantics is not completely understood. In particular, no formal semantic framework exists for describing and investigating completely visual languages.

The work presented below explores the use of the completely visual paradigm as a visual notation for textual languages. We present one possible visual notation and assign it a syntax and semantics. We do not claim that our notation is the best one, or that its use is an improvement over the equivalent text. We simply wish to provide a framework for further development of this model.

1.2 Overview

In the paper below, we define control structures for a completely visual language; that is, a language in which the execution semantics derive from graphical rules applied to the visual representation of the current program state (state in this case meaning both the current values of variables and the current program continuation). Note that by completely visual, we do not mean a programming language that contains no text; for certain aspects of programs, we believe that text may be more appropriate. The language we describe, called VIPR (Visual Imperative PRogramming language), is based roughly on the C programming language. We are also considering object-oriented features in VIPR; they are discussed in a companion paper [3]. The language discussed in this paper is intended to be a starting point for

further research in completely visual programming languages; the specifics of the language will likely change over time.

This paper focuses on control constructs in VIPR. We first present an informal description of sequential control, conditional and unconditional branches, and procedure invocation. An important aspect of our work is that our representation is simple enough that we can formally define its semantics. To our knowledge, such a formal definition has not been given to a visual imperative programming language. In this paper, we outline the formal semantics of the constructs introduced and refer the interested reader to a companion paper for more details [4].

After defining the semantics of our language, we illustrate our ideas by presenting a more complex example program in the language. This example, a bubble sort program, serves to illustrate that the static and dynamic program representation use the same framework and shows how simple programs are represented.

The design of VIPR is an ongoing effort and certain aspects of VIPR's visual representation will not be addressed in this paper. For example, expressions and functions that return values are represented textually in this paper whereas the language will eventually contain a completely visual representation of these based on expression continuations. Likewise, the current representation of data, data types, the environment, and the store are textual. Finally, while we have defined the representation and semantics of parameter passing, we omit the details in this paper because they do not significantly add to the discussion of control.

While many aspects of VIPR has been defined both informally and formally, the language is still in its infancy. As such, we do not have a working compiler for the language that translates and executes pictorial representations. Our current implementation strategy has been to create an interpreter for a textual version of VIPR (very similar to C) and use that interpreter to generate visualizations of both static and dynamic VIPR programs. We see the translation of VIPR programs as important and necessary research goal, and we intend future work to focus on the problems associated with such translation.

2 Related Work

2.1 Completely Visual Programming Languages

The research that comes closest to our own is other work in completely visual programming languages. Although much visual language work has concentrated on visual models of basically textual languages, in which the semantics of the visual language derive from the semantics of the underlying textual language (and not from the graphical properties of the visual representation), a few visual languages have been designed whose semantics derive entirely or predominantly from graphical rules. The most

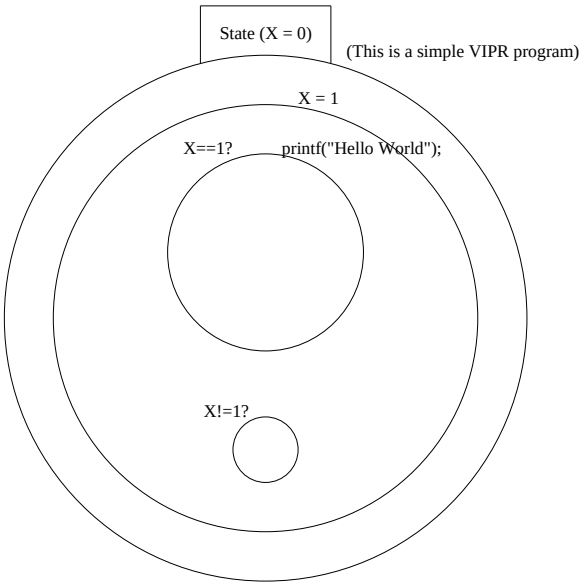
significant such language is Pictorial Janus [10], which was originally designed to model the execution of the constraint logic programming language Janus, but whose execution semantics may be derived from graphical rules applied to the visual representation. Kahn described such languages as "completely visual."

Pictorial Janus is unique in that a snapshot of the dynamically executing program contains a complete copy of the static program being executed. Kahn's definition of completely visual languages included this requirement, but if we relax the definition to allow languages where the state and the program are separate, another class of languages may be considered completely visual: the graphical transformation languages. In these languages, a program consists of a set of before/after pairs of diagrams. State consists of a set of graphical entities and their relationships. If the "before" part of a before/after pair matches part of the state, the state is transformed to conform to the "after" part of the pair. BitPict [7] is one of the simplest of such languages, in that its before/after pairs are simple pixel patterns. ChemTrains [1] and Vampire [12] allow more complex visual entities and relations, and also permit variables in the transformation rules.

Completely visual languages of both types have the advantage that the user does not need to understand two different (albeit related) sets of semantics, the semantics of the visual model and that of the underlying textual model, but can understand and write programs with knowledge of only the visual semantics. Our work differs from other work with completely visual languages because we are defining a representation of a simple imperative language with state, variables, and common control structures such as `if`, `while`, and `goto`.

2.2 Relation to Existing Visual Imperative Languages

VIPR is a departure from other visual imperative languages, most of which are based on a control flow graph model, and which are not completely visual, but rather rely on the semantics of some underlying textual language. Pict [8] is the archetypal language in this model. Pict was designed to allow the user to do everything "visually," but in this case, "visually" meant through the use of icons representing computation steps. The language itself was a representation of flow charts, where each flow chart symbol was replaced by a Pict icon. The language was found to be popular with beginning programmers, who were convinced that they were not programming, but the authors found limitations in the kinds of programs that could be implemented with the system, and they also found that the amount of screen space required to implement even trivial programs was prohibitive. Other similar systems that refine the design of visual equivalents of textual imperative languages include C^2 [11] and PECAN [13].




```
/* This is a simple VIPR program */
int x;
void main ()
{
    x = 1;
    if (x == 1) {
        printf("Hello World");
    }
}
```

Figure 1: VIPR Hello World Program and C Equivalent

3 Informal Syntax and Semantics of VIPR

3.1 Overview of VIPR

VIPR is a statically-typed, imperative programming language. Its appearance and semantics are based roughly on Kahn's Pictorial Janus [10], which is a completely visual language that supports constraint logic programming. VIPR is different from Pictorial Janus in that its underlying model of computation is a familiar imperative language with procedures. Formally, the VIPR language described here is very similar to Tennent's Simple Imperative Language [14].

Because VIPR's semantics are imperative, programs, both static and dynamic, include a graphical "state" object that participates in every operation. Whereas this state object is implicit in textual languages, it is explicit in VIPR. As mentioned, the purpose of this paper is not to explore visualizations of this state information, and as such, the internals of the state object will be represented textually in the remainder of this paper. Likewise, while we are currently working on a visual representation of language expressions, in this paper we present expressions textually.

To introduce VIPR, we include in Figure 1 a "Hello World" program both in VIPR and its equivalent in C. This example serves to illustrate many important aspects of our language. First, program statements are represented as circles. Thus, there are circles for the assignment statement and the print statement. Next, sequential execution is indicated by nesting circles inside one another. Thus, because the `printf` statement circle is inside the assignment circle, the semantics are that it executes after the assignment.

Finally, the example shows that conditionals are indicated by including two possible circles inside the same circle. Each of these circles indicates a possible branch and must be guarded by a condition (indicated by text on the left of the circles ending with a question-mark).

Throughout the examples presented, we will write the actions associated with a statement as text on the upper right-hand side, outside the circle with which the action is associated. Predicated guards on statements are written as text on the upper left-hand side outside the statement being guarded. Comments in VIPR appear in parentheses. Note that both the action and the guard are optional (e.g., the outmost circle in Figure 1 has neither, as does the `else` part of the conditional). If the guard is missing, a true guard is assumed. If the action is missing, then a null statement is assumed.

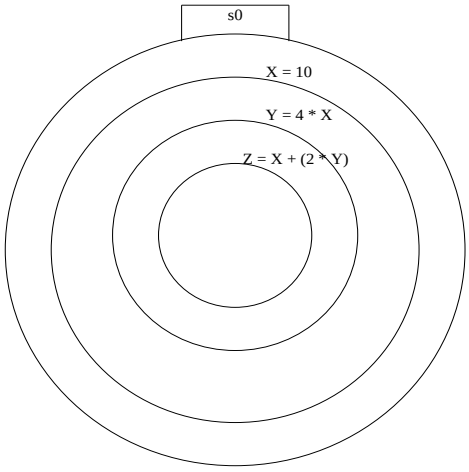
Intuitively, execution of a VIPR program goes as follows. A state object is present in the outermost of the nested rings. Each execution step involves merging the outermost two rings of the diagram. When rings merge, the action associated with the ring being consumed takes place on the state. Thus, rings may not merge unless a state object is present. When more that one ring is present when a merge is about to happen, the guards are evaluated in the current state and the guard evaluating to true determines what ring will merge. Other nested rings with false guards disappear and are not evaluated. Execution terminates when there are no further nested rings.

One of the interesting aspects of our representation is that not only does it have a two-dimensional representation, but that the two-dimensional representation suggests a corresponding three-dimensional interpretation. Programs may be visualized as pipes containing branches and merge points, and execution is equivalent to moving through a particular path down the pipe. The two-dimensional representation can then be interpreted as a stylized perspective drawing of the view down the pipe, and the dynamic view of execution can be interpreted as a trip through the pipe. Equivalently, a VIPR program may be envisioned as a three-dimensional flow chart with the two-dimensional representation being a view down the flow chart in the direction of the control flow.

3.2 Control Constructs in VIPR

3.2.1 Sequential Control

The most basic kind of control is sequential control, which is implicit in every textual imperative programming language. In VIPR a sequence of sequential statements is represented by nested circles. Figure 2 shows three sequential C assignment statements and their equivalent in VIPR. Execution of these assignments progresses from the outermost ring inward, as illustrated in Figure 3. The outermost ring combines with nested rings in succession and each combination results in a modification to the state. Thus, the effect of assignment on the state, implicit in textual languages, is explicit in VIPR. This




```
x = 10;
y = 4 * x;
z = x + (2 * y);
```

Figure 2: Static Representation of Sequential Statements in VIPR

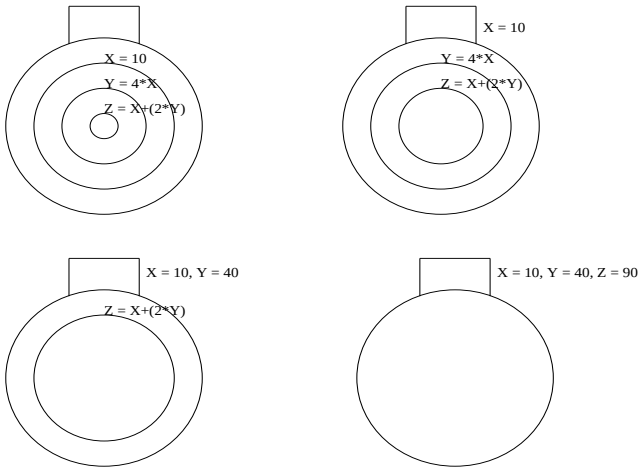
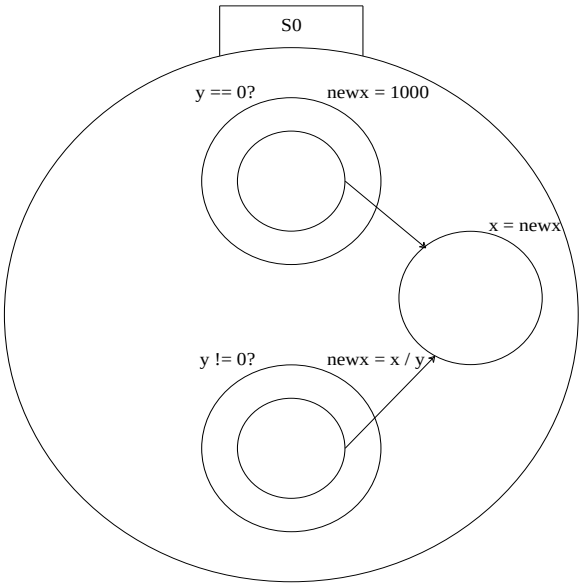


Figure 3: Dynamic Representation of Sequential Statements in VIPR




```
if (y == 0) {
   newx = 1000;
} else {
   newx = x / y;
}
x = newx;
```

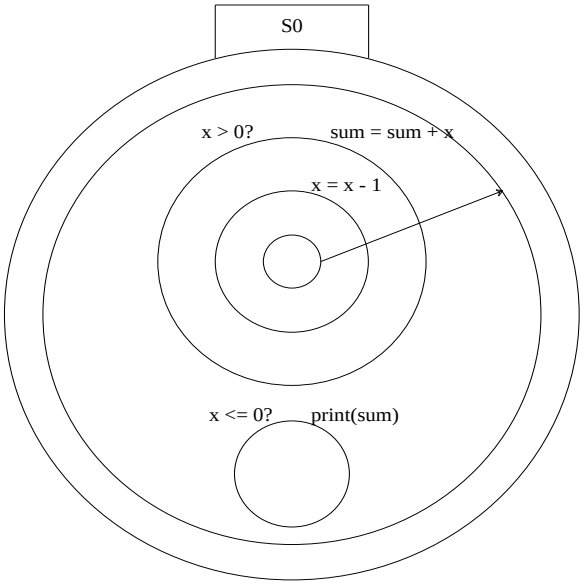
Figure 4: Representation of a Conditional Statement in VIPR

example also further illustrates the perception that VIPR execution represents traveling down a tube—as each statement is executed, its disappearance gives the viewer the perception that is has been passed by.

3.2.2 Conditional and Unconditional Branching

Conditional branch structures are represented in VIPR by adding an optional guard clause to each statement in the program. For example, in the "Hello World" program, the two branches of the `if` statement were represented by two circles with guards of `X==1` and `X!=1`. The guard mechanism is very general and any number of alternative guarded statements can appear inside a circle. Because there is no linear order to these guards, their semantics are that they may be evaluated in any order and at most one of them can evaluate to `true`. If none of the guards are true, the program is in error. Thus, we support conditional semantics very similar to Dijkstra's guarded-if construct [6]. With these semantics, `if-then`, `if-then-else`, and `case` statements can be easily represented. `if-then-elsif-else` semantics simply requires nested `if-then-else` conditionals.

Figure 4 contains the static VIPR representation of a C `if` statement where an assignment follows both branches of the conditional. This example also illustrates the use of arrow notation in VIPR. First, note the guards attached to the two statements that are conditionally executed, just as we saw in the "Hello World" example. Next, note the circles nested inside these rings. These rings have no associated actions but are necessary because they are used as anchors for the arrows leading to the statement following the `if`. The arrow notation in VIPR is a simple substitution rule that says whatever circle the




```
while (x > 0) {
   sum = sum + x;
   x = x - 1;
}
print(sum);
```

Figure 5: Representation of a `while` Loop in VIPR

arrow points to (the arrow's target) can be substituted for the circle that is the source of the arrow. Thus, the "x = newx" statement could semantically be substituted into both branches of the conditional. While these semantics are also true in the case of the textual `if` statement, the ability to substitute is implicit, whereas in VIPR "substitutability" is obvious and explicit.

One may ask why the "x = newx" assignment is not executed as one of the guarded alternatives in the outer ring. The rule for circles appearing as the target of an arrow, where the arrow is pointing to the outside of the circle, is that they are will not be executed unless the source of the arrow has already been executed. For an example where the arrow points to the inside of a circle, see the next section. The arrow in VIPR is exactly equivalent to the familiar `goto` statement of textual languages. Thus, any form of unconditional branching is possible in VIPR using the arrow notation.

3.2.3 Iteration

With the semantics of conditional execution and the arrow substitution rule, the representation of a `while` statement follows immediately. Figure 5 illustrates the VIPR code for a while loop that sums the numbers from 1 to x and prints them. The most interesting thing about this example is that loops require the arrow to point to an enclosing circle instead of an external circle.

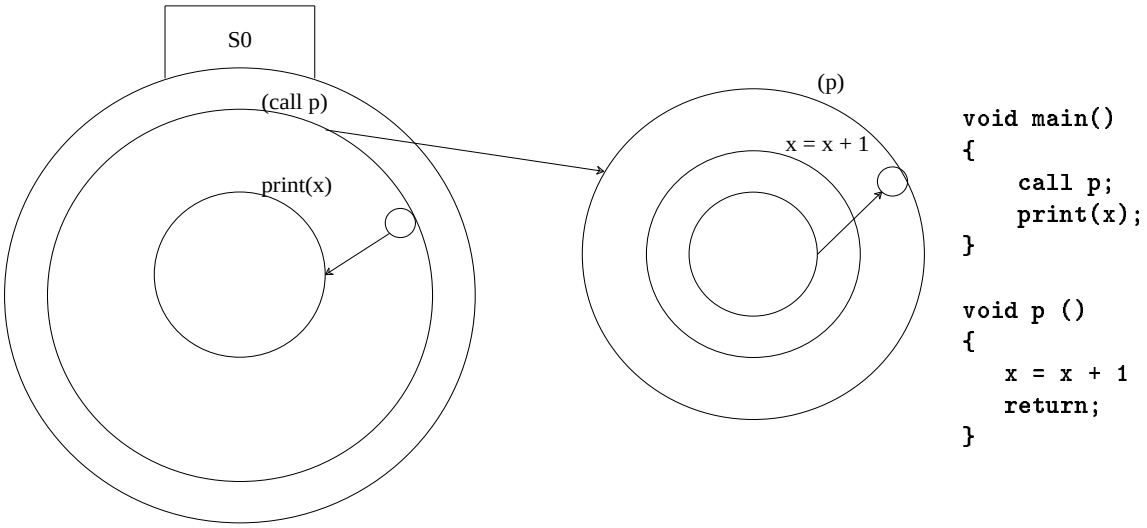


Figure 6: Procedure Call and Return in VIPR

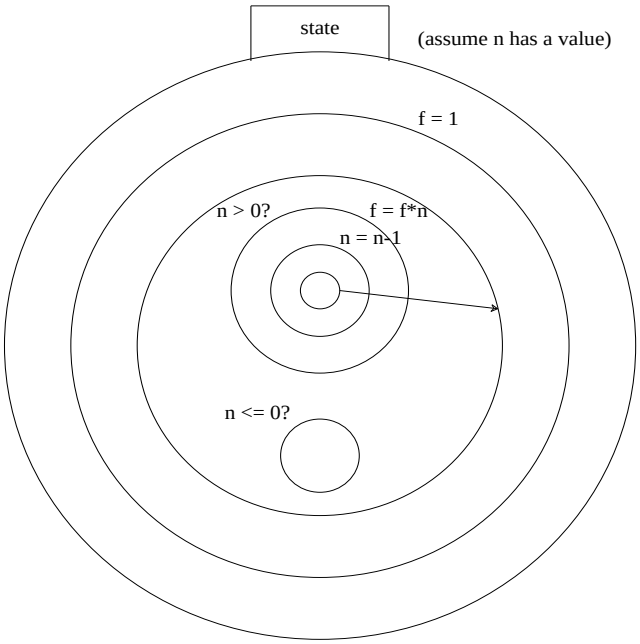
3.2.4 Procedure Call and Return

Procedures are sets of nested rings outside the main procedure's rings (see Figure 6). Procedure invocation in VIPR is a slightly modified version of the `goto` arrow. A procedure call is a ring that possesses an arrow pointing to the procedure being called. Every ring representing a procedure call must also indicate where the continuation of the procedure will be. Thus, procedure continuations, implicit in textual languages, are explicit in VIPR.

To understand our representation, consider the C code in Figure 6. In this example there are two procedures, main, which contains a call to procedure p, and p, which simply increments a global variable and returns. In VIPR, the procedure call ring contains a smaller circle attached to the inside of the ring, connected by an arrow to a circle representing the print statement that is executed upon return. Thus, the small attached circle is used to indicate a special parameter passed to every procedure indicating what statement to execute directly upon return.

Each procedure definition ring (e.g., the one for p in the figure) has a corresponding small attached circle representing the continuation formal parameter. Any ring representing a return statement has an arrow leading from it to the continuation formal parameter, which at call time will have been combined with the caller's continuation and thus leads to the statement to be executed on return. Note that using this notation, the statement doing the call does not use any names (the "`(call p)`" notation is simply a comment). Likewise, procedures do not require names (again the "`(p)`" is just a comment).

As mentioned above, the graphical mechanism of procedure call and return is a special case of the semantics of the goto arrow. Because there is an arrow from the calling ring to the procedure construct,




```
/* assume n has a value */
f = 1;
while (n > 0) {
    f = f * n;
    n = n - 1;
}
/* f contains n! */
```

Figure 7: Iterative Definition of Factorial in VIPR and C Equivalent

the procedure construct is substituted for the call ring. The commands to be executed on return are substituted for the return ring because of the chain of arrows running from the return ring to the small continuation parameter ring on the inside of the procedure ring, and then from the small continuation argument ring on the procedure call ring to the commands to be executed upon return.

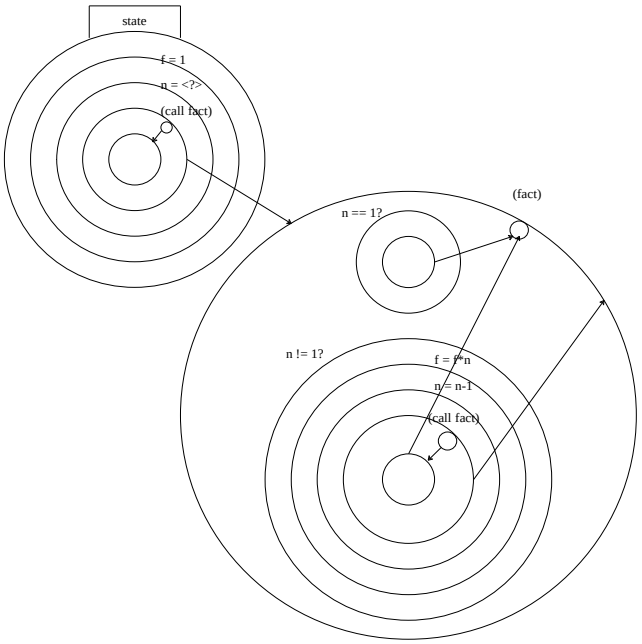
3.2.5 Iterative and Recursive Definitions of Factorial

To conclude this section, we present both an iterative and recursive definition of a program that computes the factorial of some number (n) that has been given an initial value in the state. In Figure 7, we present the iterative version of the factorial program both in C and in VIPR. In Figure 8, we present the recursive version of the factorial program both in C and in VIPR.

4 Formal Semantics of VIPR

4.1 Description of Approach

In this section, we will describe a restricted version of VIPR that models the Simple Imperative Language (SIL). There are several advantages to this approach. The SIL is a subset of most conventional imperative languages, so by modeling it we demonstrate that a large subset of the control constructs of these




```
void main()
{
    f = 1;
    n = <?>; /* assign n */
    fact();
    /* f contains n! */
}

void fact()
{
    if (n == 1) return;
    else {
        f = f * n;
        n = n - 1;
        fact();
    }
}
```

Figure 8: Recursive Definition of Factorial in VIPR and C Equivalent

languages may be modeled in VIPR. Second, the semantics of the SIL are well understood and accepted (see [14] for a more detailed discussion of the semantics of the language), so it will suffice to describe the syntax of the VIPR constructs and give the equivalent SIL constructs. A formal denotational and operational description of the general VIPR language is beyond the scope of this paper, but may be found in [4]. In addition [4], contains a proof that the semantics of the constructs presented below, interpreted as general VIPR constructs, are equivalent to the semantics of the Simple Imperative Language as conventionally defined.

4.2 A Note on Syntax

For the syntactic specification of VIPR, we employ a formalism known as *relational grammars* [15], which are a class of two-dimensional grammars. Relational grammars differ from conventional grammars in that the righthand sides of productions in the latter are simply strings of terminals and non-terminals, where the only spatial relationships—textual ordering and adjacency—are implicit; righthand sides of relational grammars consist of a multiset of terminals and non-terminals, along with a set of spatial relations among the elements of the multiset. Two-dimensional grammar productions are generally textual, although they may be illustrated by graphical specifications. The graphical specifications alone, however, are generally ambiguous and cannot be used in a grammar without additional annotations.

Another difference between conventional grammars and two-dimensional grammars is that conventional grammars used for syntax are context free, while two-dimensional syntactic descriptions are often context sensitive. In textual languages, program units are generally contiguous pieces of text, and constructs that span non-adjacent units, such as the connection between the declaration of an identifier and its use, are specified through non-syntactic methods. In visual languages, on the other hand, such relationships are often made explicit through a graphic representation (a definition of a procedure and a call to the procedure may be connected by an arrow, for example), and there is no compelling reason to separate out such graphical constructs from a syntax. In relational grammars, particularly the subclass known as *picture layout grammars* [9], context sensitivity is modeled by allowing the spatial relations on the righthand side of a rule to reference objects that are not part of the corresponding multiset. These objections are known as *remote objects*. In the textual version of the grammar rules, remote objects are italicized; in the graphical versions of the grammar rules, they are either italicized or printed with dotted lines.

It should be noted that it is difficult to parse two-dimensional grammars efficiently. This is not a concern for us as we are using the two-dimensional grammar as an abstract syntax. We assume that the graphical input has been parsed, and that a tree or graph conforming to the syntactic description has already been derived.

4.3 Formal Definition

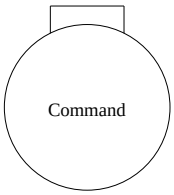
Figures 9 and 10 give the formal syntax of the SIL subset of VIPR in graphical form, along with the equivalent SIL constructs. Figure 14 in Appendix A gives the equivalent two-dimensional syntax rules in textual form. Note that [C] denotes the semantic meaning of the construct C. The particular valuation function to be used (Command, Procedure, etc.) should be evident from the context.

Rules 1, 2, and 3 describe the large-scale structure of a program. A program consists of a main procedure and a possibly empty set of procedures. Each of these entities is disjoint; they do not overlap.

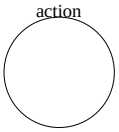
The main procedure is unlike the others in that it contains a distinguished state object (as specified in rule 4). Since execution can only take place in the presence of the state object, execution must begin with the main procedure. As we have discussed earlier, control is transferred to other procedures by copying the called procedure into the main procedure.

A main procedure contains a command, which may be any of the commands described in rules 5 through 9. (Rule 10, describing the **return** statement, is not applicable to main procedures since it requires the presence of a small ring representing the continuation parameter passed to procedure, and the main procedure has no continuation parameter.)

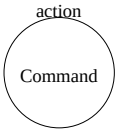
Syntax	SIL equivalent
1) Program -> Main Proc-Set	[Main] ; [Proc-Set]
2) Proc-Set -> Proc Proc-Set	[Proc] ; [Proc-Set]
3) Proc-Set ->	
4) Main ->	[Command]



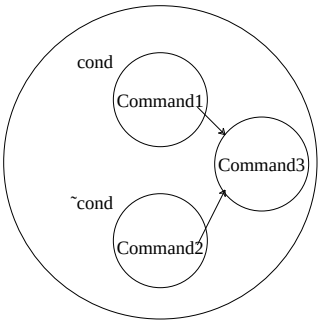
5) Command ->	action



6) Command ->	action; [Command]



7) Command ->	if cond then [Command1] else [Command2]; [Command3]



8) Command ->	while cond do [Command1] ; [Command2]

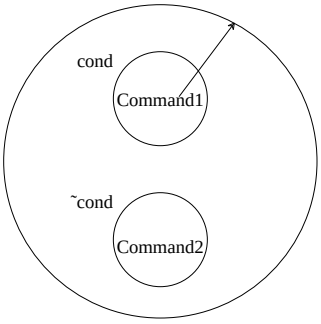


Figure 9: Graphical Formal Syntax of SIL Subset of VIPR

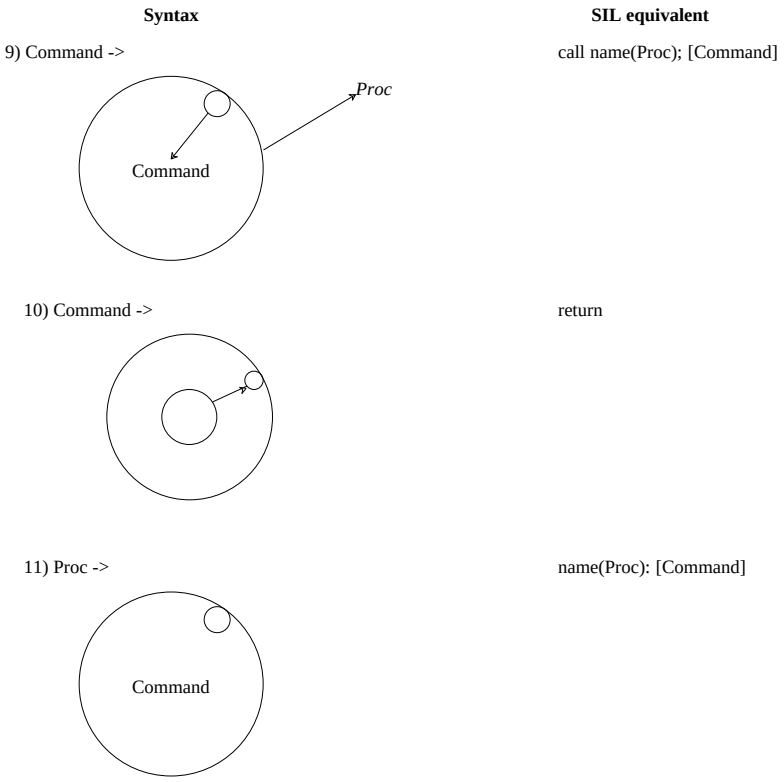


Figure 10: Graphical Formal Syntax of SIL Subset of VIPR (continued)

Rule 5 describes a single command, generally the command performed just before program termination (since there is no subsequent command contained within it). This command has an associated action, usually an assignment or input/output operation. As described earlier in our informal semantics, the effect of this construct is to merge the ring with the immediately surrounding main procedure ring and perform the associated action.

Rule 6 describes sequential execution. The meaning of this construct is the meaning of the action on the outermost ring, followed by the meaning of the command construct contained within. The execution of the construct is modeled graphically by having the outer ring merge with the surrounding main procedure ring, performing the associated action, and leaving the command contents as the new contents of the main procedure ring, ready to be executed.

Rule 7 denotes a conditional statement. The two alternatives are labeled with conditions, one the negative of the other. If the positive condition holds, the enclosed command, denoting the **then**-part, executes, and if the negative condition holds, the enclosed command, representing the **else**-part, executes. At the end of each alternative, control transfers to the third ring, containing the commands that execute after the conditional statement completes. This joined execution thread is not necessary; execution could continue inside each alternative, in the same way that:

if Expr **then** Stmt1 **else** Stmt2; Stmt3

is equivalent to:

if Expr **then begin** Stmt1;Stmt3 **end else begin** Stmt2;Stmt3 **end**

The format presented here is more readable and better demonstrates programmer's intent.

Rule 8 specifies the structure of a **while** loop. As long as the condition holds, Command1 is repeatedly executed. The repetition is given by the arrow that leads from inside Command1 to the outer ring of the while statement. When the last statement of Command1 is executed, the final statement is replaced by the while command itself. At some point, the negative condition succeeds, and execution proceeds outside the iteration.

Rule 9 models procedure calls. The outer ring with the arrow pointing to the *Proc* object (which is not a part of the Command construct but is rather part of the context) denotes a graphical substitution of the procedure for that ring. The inner Command construct is the return continuation, which is passed to the procedure as a parameter. As we shall see in rules 10 and 11, the return construct will reference this return continuation in such a manner that the return continuation command is substituted for the return command ring. This models call and return in the conventional manner.

Rule 10 represents procedure returns. In the procedure call in rule 9, a reference to commands to be executed after return is passed as an argument. This reference is in the form of an arrow, which also represents graphical substitution. The arrow from the ring in the return construct links to the arrow to the return continuation, thus signaling that the return ring should be graphically replaced by the object it points to: in this case, the return continuation. Note that the outer ring and the small parameter ring are not part of the construct, but are rather part of the rule context, in this case the surrounding procedure.

Rule 11 describes the form of a procedure. It consists of a large ring with a small ring representing the continuation parameter, and some command construct in the interior. The meaning of the construct is the meaning of the command, prefaced with a system-generated procedure name, which is used in calls to the procedure.

5 An Illustrated Example

In this section we illustrate the execution of a VIPR program by showing the static and dynamic representations of a bubble sort function. The textual representation of the bubble sort is shown in Figure 11 and the corresponding static representation is shown in Figure 12.

Figure 13 shows snapshots of the dynamic representation of the bubble sort function at four different steps in its execution. The four snapshots are labeled with the corresponding statement number from

```
    int a[100], n;

[1] void BubbleSort(void)
    {
      int hold, j, pass, exch;

[2]   pass = n;
[3]   do
      {
[4]       exch = 0;
[5]       for (j=0; j < (pass-1); j++)
          {
[6]           if (a[j] > a[j+1])
              {
[7]               hold = a[j];
[8]               a[j] = a[j+1];
[9]               a[j+1] = hold;
[10]              exch = 1;
              }
          }
[11]      pass--;
[12]  } while (exch);
    }
```

Figure 11: VIPR Bubble Sort Function

Figure 11. In the interest of space and simplicity, some text labels have been omitted, and areas that are too crowded are shaded, indicating that there is some lost information. In an actual implementation of VIPR such overcrowding would be handled by culling out all objects that appear below screen resolution. State objects have been omitted for the same reason.

Figure 13(a) shows the dynamic representation just prior to executing the `if` at statement [6]. The shaded circle inside the `j++` statement contains a copy of the `for` statement (excluding the loop variable initialization). This copy was obtained by substitution of the circle pointed to by the `j++` statement in Figure 12.

Figure 13(b) shows the situation when the conditional of the `if` statement has passed and the exchange beginning at statement [7] is about to execute. Substitution of the `j++` statement has occurred.

Figure 13(c) shows the program at the end of the `for` loop (statement [5]). This state may have been reached by executing through the sequential statements that are the outer circles of Figure 13(b) or by taking the alternate branch from Figure 13(a). With the increased resolution we can see that the `j++` statement contains a complete copy of the `for` loop.

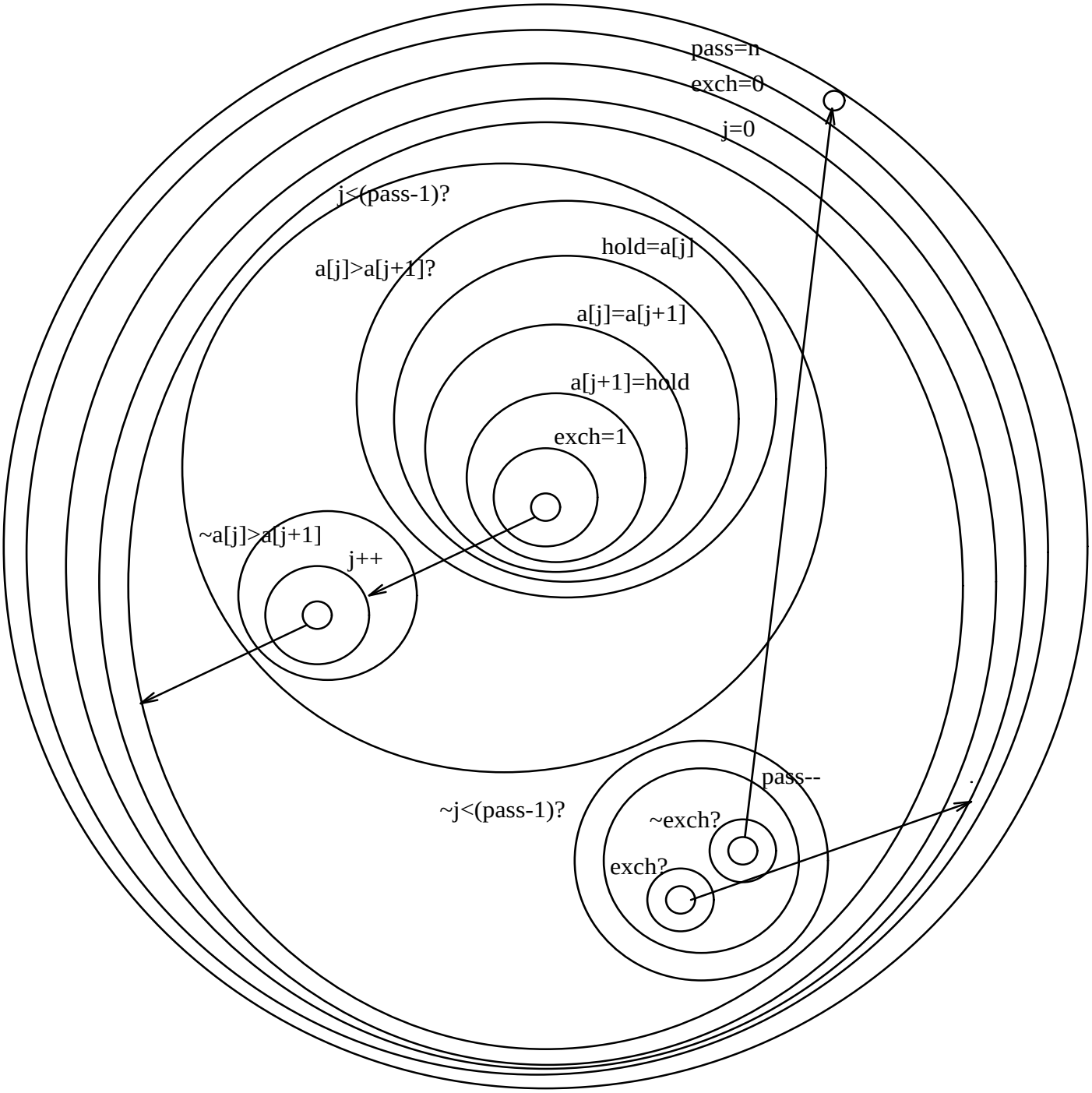


Figure 12: Static Representation of Bubble Sort Function

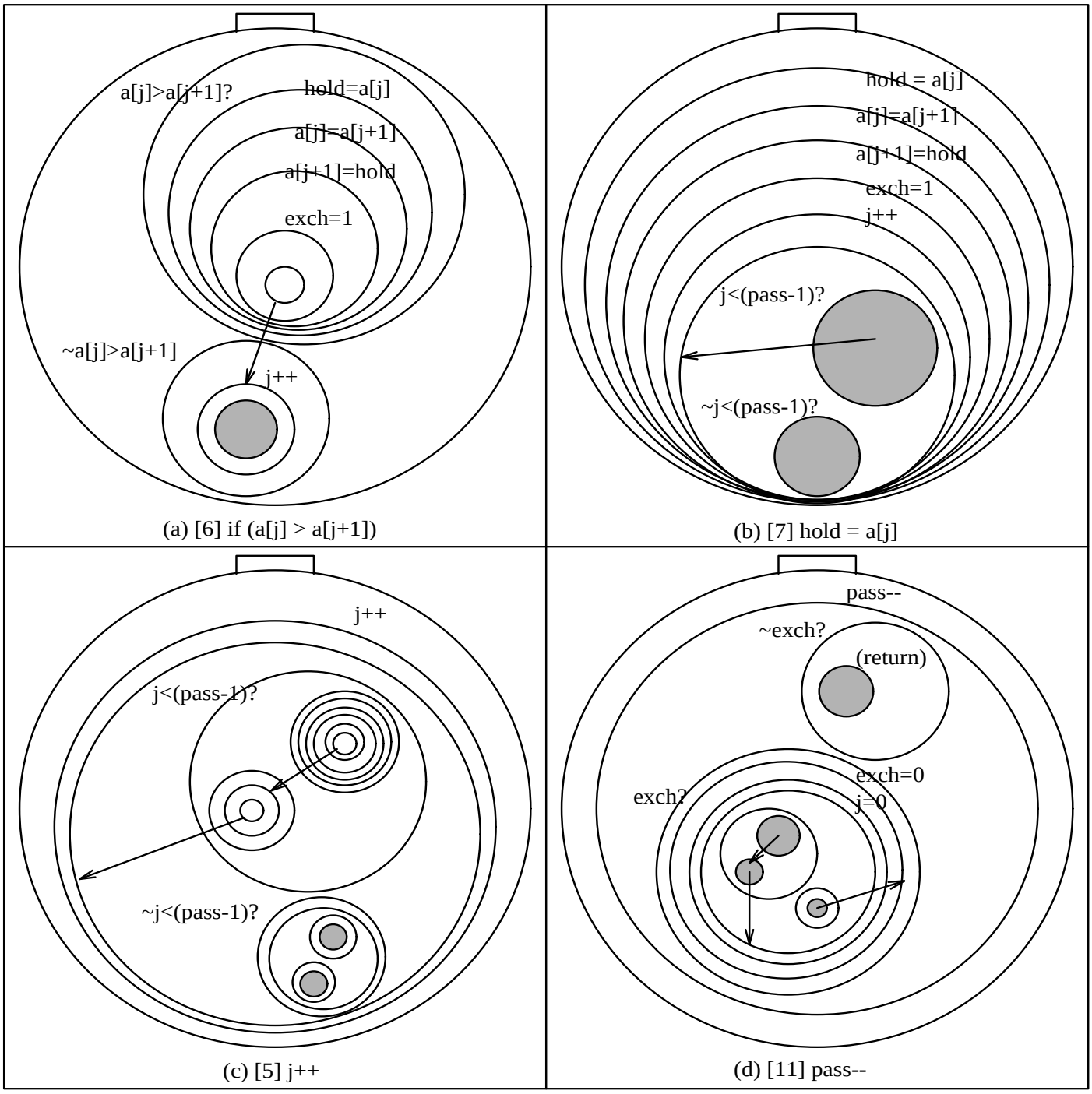


Figure 13: Dynamic Execution of Bubble Sort Function

Figure 13(d) shows the situation at the end of a pass through the bubble sort, where the `exch` flag is to be tested. Here we can see that if an exchange has taken place (`exch == true`) we will execute another complete pass of through the bubble sort. Otherwise, the function will return. Note that at this point, the shaded circle labeled (`return`) would contain the continuation of the program from the point of call of the `BubbleSort` function. The program continuation is obtained through substitution from the continuation parameter of the function.

6 Summary

In this paper, we have described the control structures in a completely visual imperative programming language. The execution semantics of our language, VIPR, derive from graphical transformation rules on the visual representation, and as such are independent of any underlying textual language. One advantage of this approach is that a user of the language can understand it without first understanding a textual language.

Because VIPR is completely visual, the static program and its dynamic execution are presented in the same framework. Such an framework, specifically for imperative languages, may be valuable in unifying a great deal of related work in visualizing specific static and dynamic aspects of programs.

Because the execution semantics of VIPR derive entirely from the visual representation, programming language elements that are implicit in textual languages are explicit in VIPR. For example, the presence of a return continuation parameter (e.g., the return address), absent in textual languages, is present in VIPR. The state, which is implicitly passed from statement to statement in textual languages, is explicit in VIPR. We anticipate that because these language elements are explicit in VIPR, programmers may understand them more easily.

Finally, we have defined the semantics of VIPR formally. To our knowledge, ours is the first formal definition of a completely visual imperative programming language. With such a definition, we will be able to reason formally about aspects our our language and prove how it relates to other formally defined textual languages.

VIPR is a young language that is still being designed. The implementation of VIPR is also just beginning. Our current implementation strategy is to build an interpreter for a textual equivalent of VIPR and then translate textual VIPR into the visual representation. Beyond that, we anticipate building a programming environment in which visual VIPR programs could be written and executed. Our goal with VIPR is to attempt to better understand the design, formal semantics, and implementation issues of completely visual imperative languages.

References

[1] B. Bell and C. Lewis. ChemTrains: A language for creating behaving pictures. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 188–195, Bergen, Norway, 1993.

[2] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, pages 10–19, 1987.

[3] Wayne Citrin, Michael Doherty, and Benjamin Zorn. The design of a completely visual object-oriented programming language. In *OOPSLA'93 Workshop on Visual Object-Oriented Programming*, pages 19–35, Washington, D.C., September 1993.

[4] Wayne Citrin, Michael Doherty, and Benjamin Zorn. Formal semantics of control constructs in a completely visual imperative language. Technical Report CU-CS-673-93, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, September 1993. In preparation.

[5] E. Dijkstra. On the cruelty of really teaching computer science. *Communications of the ACM*, pages 1397–1404, December 1989.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[7] G. W. Furnas. New graphical reasoning models for understanding graphical interfaces. In *Proceedings of CHI'91*, pages 71–78, Anaheim, CA, April 1981. ACM Press.

[8] E. P. Glinert and S. L. Tanimoto. Pict: An interactive graphical programming environment. *Computer*, pages 7–25, November 1984.

[9] E. J. Golin and S. P. Reiss. The specification of visual language syntax. In *1989 IEEE Workshop on Visual Languages*, pages 105–110, Rome, ITALY, October 1989.

[10] K. M. Kahn and V. A. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 7–15, Skokie, IL, October 1990.

[11] M. E. Kopache and E. P. Glinert. *Visual Programming Environments: Paradigms and Systems*, chapter C^2: A Mixed Textual/Graphical Environment for C. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[12] D. W. McIntyre and E. P. Glinert. Visual tools for creating iconic programming environments. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*, pages 162–168, Seattle, WA, September 1993. IEEE Computer Society.

[13] S. P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, pages 276–285, March 1985.

[14] R. D. Tennent. *Semantics of Programming Languages*. Prentice-Hall, 1991.

[15] K. Wittenberg. Earley-style parsing for relational grammars. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, Seattle, WA, September 1992.

A Textual Two-dimensional Syntax Rules of VIPR

1) Program → {Main,Proc-Set},
Main **outside** Proc-Set

2) Proc-Set → {Proc,Proc-Set},
Proc **outside** Proc-Set

3) Proc-Set → {}

4) Main → {state,ring,Command},
contents(ring) = {Command},
state **connects-outside** ring

5) Command → {ring,action}
action **on** ring,
contents(ring) = {}

6) Command → {ring,action,Command},
action **on** ring,
contents(ring) = {Command}

7) Command → {$ring_1$,$ring_2$,$ring_3$,$ring_4$,$cond_1$,$cond_2$,
$Command_1$,$Command_2$,$Command_3$,$arrow_1$,$arrow_2$},
contents($ring_1$) = {$ring_2$,$ring_3$,$ring_4$,$arrow_1 arrow_2$},
$cond_1$ **on** $ring_2$, $cond_2$ **on** $ring_3$, **text**($cond_1$) = ˜**text**($cond_2$),
contents($ring_2$) = {$Command_1$},
contents($ring_3$) = {$Command_2$},
contents($ring_4$) = {$Command_3$},
$arrow_1$ **connects** $Command_1$ **to** $ring_4$,
$arrow_2$ **connects** $Command_2$ **to** $ring_4$

8) Command → {$ring_1$,$ring_2$,$ring_3$,$cond_1$,$cond_2$,
$Command_1$,$Command_2$, arrow},
contents($ring_1$) = {$ring_2$,$ring_3$, arrow},
$cond_1$ **on** $ring_2$, $cond_2$ **on** $ring_3$, **text**($cond_1$) = ˜**text**($cond_2$),
contents($ring_2$) = {$Command_1$},
contents($ring_3$) = {$Command_2$},
arrow **connects** $Command_1$ **to** $ring_1$

9) Command → {ring,smallring,$arrow_1$,$arrow_2$,Command},
contents(ring) = {Command,smallring,$arrow_1$},
smallring **connects-inside** ring,
$arrow_1$ **connects** smallring **to** Command,
Proc **outside** ring,
$arrow_2$ **connects** ring **to** *Proc*

10) Command → {ring,smallring,arrow},
ring **inside** $ring_2$,
smallring **connects-inside** $ring_2$,
arrow **connects** ring **to** smallring

11) Proc → {ring,Command},
contents(ring) = {Command,*smallring*},
smallring **connects-inside** ring

Figure 14: Textual Syntax Rules for SIL Subset of VIPR