Spring 5-1-1993

A Formal Definition of Control Semantics in a Completely Visual Language ; CU-CS-673-93

Wayne V. Citrin
University of Colorado Boulder

Michael Doherty
University of Colorado Boulder

Benjamin G. Zorn
University of Colorado Boulder

A Formal Definition of Control Semantics in a Completely Visual Language

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

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

Contact: Wayne Citrin
Department of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, CO 80309-0425

Tel: (303)-492-1688
E-mail: citrin@cs.colorado.edu

A Formal Definition of Control Semantics in a Completely Visual Language

Abstract

Visual representations of programs can facilitate program understanding by presenting aspects of programs using explicit and intuitive representations. To explore this idea, we have designed a completely visual static and dynamic presentation of an imperative programming language. Because our representation of control is completely visual, programmers using 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 prove that using our language to model common high-level constructs is semantically sound. This paper is an expanded version of [5]; we present a revised and more complete treatment of the language's graphical semantics, introduce the concept of canonical configurations, and discuss the representation of function return values.

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 [3] and [7] 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 designing visual languages to be understood in this way 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 have been in the underlying textual language. In fact, it may be 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 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 equivalent 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. If a properly designed completely visual language is based on an textual language, however, it should allow programmers to use visual reasoning to construct their programs, 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, up until now no formal semantic framework has existed for describing, investigating, and reasoning about completely visual languages.

The work presented below explores the use of the completely visual paradigm as a visual notation for imperative programs. We present one possible visual notation and assign it a syntax and semantics. The chief contribution of this paper is the formulation of a rigorously defined formal semantics for the language, expressed in a completely graphical form. Besides being the first use of a completely visual formal semantics known to us, the existence of such a semantics provides a method for reasoning about visual programs, and we prove a number of simple results concerning programs written in our language.

In addition, we demonstrate the correspondences between program structures in our language and equivalent structures in a textual imperative language. The existence of such correspondences suggests that conventional compiler technology may be employed in the implementation of a properly designed completely visual language.

This paper is an expanded version of [5]; we present a revised and more complete treatment of the language's graphical semantics, introduce the concept of canonical configurations, and discuss the representation of function return values in our language.

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 [4].

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 any visual programming language.

The purpose of this paper is to describe the formal semantics of control constructs in VIPR, and certain aspects of VIPR's visual representation will not be addressed here. For example, expressions and assignments are represented textually in this paper. Likewise, the representation of data, data types, the environment, and the store are textual.

2 Related Work - Completely Visual Programming Languages

One of the earliest languages based on graphical transformation rules was BitPict [8], in which a program is a set of rules used to transform patterns of pixels on a grid. If a pattern of pixels on part of BitPict's grid matches the precondition pattern of one of the rules, that pattern is altered to conform to the postcondition pattern of the rule. A number of fairly sophisticated animations have been designed using this system.

A pair of similar languages, ChemTrains [2] and Vampire [11], expanded the graphical transformation model by adding additional graphical primitives and spatial relations. Vampire extended the model still further by adding textual attributes and manipulations to the graphical objects. The power of these systems was illustrated by the applications that were constructed with them: ChemTrains has been used to implement a number of complex simulations, including an ecosystem simulation and a communications protocol simulation [1], while Vampire has been used to reimplement a number of previously-built visual programming systems in a fraction of the time that the initial systems were implemented.

Graphical transformation systems are close to, but not quite, completely visual programming languages. They enforce a separation between state (the current graphical program configuration) and program (the set of graphical transformation rules themselves). Completely visual languages, as originally defined by Kahn [10], make no distinction between state and program. Any completely visual program contains a current state and enough of the program to enable the program to run to completion. (That is, previously executed program constructs may be lost, and the program might not be able to be restarted, but any program snapshot contains sufficient information to run the program to completion from the point at which the snapshot was taken.)

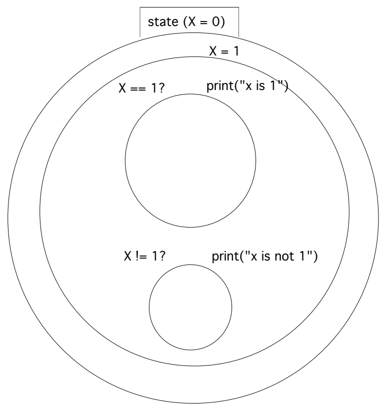
The most significant visual language whose semantics is based on completely visual principles is Pictorial Janus [10], which was originally designed to model the execution of the constraint logic programming language Janus [13] but whose execution semantics may be derived from graphical rules applied to the visual representation. For more information on Janus and Pictorial Janus, the reader is referred to the appropriate references.

Although a program in Pictorial Janus maps directly into a program in textual Janus, a user with no knowledge of the underlying Janus language may write and understand a Pictorial Janus program simply in terms of graphical transformations, which may be more intuitive. Because of this, Kahn has proposed that Pictorial Janus and similar languages may be useful for rapid prototyping and program "sketching," and for technical communications with individuals who are not sophisticated programmers [9].

3 Informal Syntax and Semantics of VIPR

In this section, we present an overview of the basic principles of VIPR control semantics: sequencing, conditional execution, and substitution. In addition, we introduce the important notion of a state object.

Figure 1 shows a typical simple program written in both VIPR and in C. It illustrates the notions of sequencing, conditional execution, and state. One should first note that every VIPR program contains exactly one state object. In addition to containing the values of the variables, the state object is used to specify the locus of control. The current locus of control in any executing VIPR program is the interface between the outer ring to which the state object is attached, and the ring (or, as we shall see, one of the rings) nested immediately within it. This is the feature that gives VIPR its sequential, imperative quality. A computation step occurs when the two rings merge, and the action associated with the inner of the two rings is performed, possibly altering the state. In figure 1, the first execution step is to merge the ring labeled with the assignment statement 'X=1' with the outermost state ring, thus causing the value of X to be changed to 1. Note that atomic statements such as assignment statements and I/O statements that do not alter the flow of control are represented textually. Also note that every ring is considered to be associated with an action. If the action is omitted, a ring's action is considered to be the null statement that does not alter the state. Actions are represented as labels on the upper right portion of a ring.



```
int x;
void main()
{
   x = 1;
   if (x == 1)
      printf("x is 1");
   else
      printf("x is not 1");
}
```

Figure 1: Introductory VIPR program and C equivalent

When a ring merges with the outermost state ring, a new ring or rings come into position immediately inside the state ring. Thus, sequencing is represented by nesting of rings. One command following another command is represented by the ring representing the second command being nested immediately inside the ring representing the first command. In figure 1, for example, immediately following the execution of the assignment statement, a pair of rings moves into place immediately inside the state ring. This is a conditional construct (which is described in more detail below), and is the next statement to be executed.

VIPR implements conditional execution through guarded commands. Thus, each ring is considered to possess a guard, which is generally represented as a label in the upper left portion of the ring, and which is expressed as a Boolean expression. If a guard is omitted, it is assumed to be **true**. Thus, after the assignment statement is executed in figure 1, two rings become candidates for execution. Each ring has a guard associated with it. The guards are evaluated, and one of the rings whose guard evaluates to true is selected. All other rings are eliminated. What happens when more than one guard evaluates to true is left unspecified. This allows for nondeterminism; to represent deterministic constructs, care must be taken that only one guard evaluate to true at any given time.

The third basic VIPR principle is substitution. This feature allows the representation of procedure calls and returns, iteration, and gotos. Figure 2 represents the principle of substitution. If the outermost ring within the state ring contains no action, but possesses an arrow pointing to another set of rings, the ring is replaced by the structure that is the target of that arrow. In section 5, we will show how this construct allows modeling of procedures and iteration.

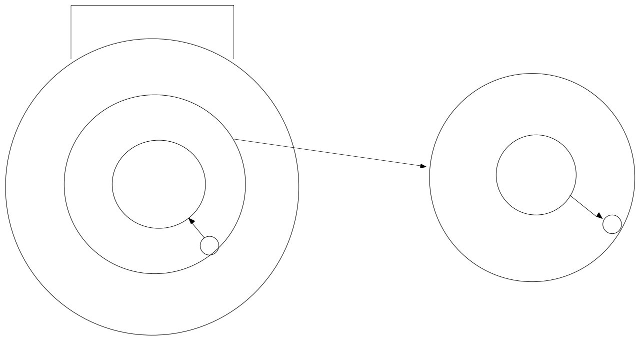


Figure 2: Use of arrows for substitution

It should also be noted that substitution also applies to structures (in this case, smaller rings) attached to the substitution ring and the target ring. In this case, corresponding small rings attached to the larger rings on each end of the substitution arrow (such as the ones in figure 2) are matched, and their contents (which may be labels or arrows) are bound. This allows us to model parameter passing, return continuations, and function return values. Section 4.2 gives the formal semantics of substitution, and section 5 gives some examples of how it may be used in modeling parameter passing, return continuations, and function return values.

Any snapshot of an executing VIPR program contains both the current state of the program and a graphical representation of the remainder of the program. Thus, VIPR programs may be halted at any point and a graphical snapshot saved away. The snapshot may later be retrieved and executed, and the resulting program will run to completion from the point at which it was halted. Figure 3 illustrates this principle by showing snapshots of the program of figure 1 after each execution step.

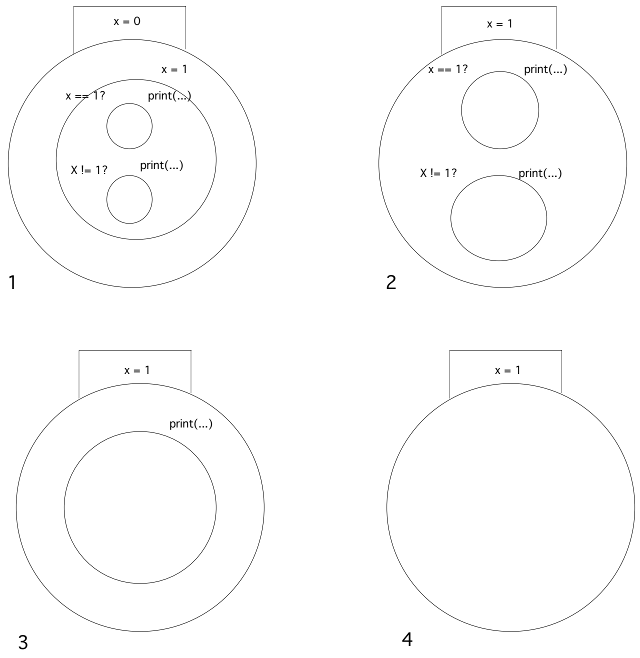


Figure 3: Dynamic execution of a VIPR program

4 Formal Semantics of VIPR

4.1 Description of Approach

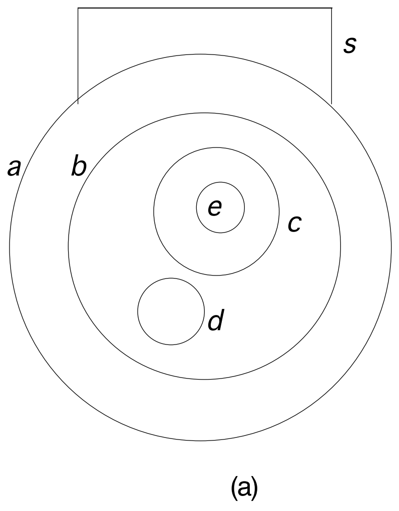
In this section, we describe the semantics of VIPR using Milner-style operational semantics[12]. We describe the language semantics through a set of allowable computation steps over machine configurations. These machine configurations are graphical configurations and a state, and the computation steps are described as graphical transitions and state transformations. As we will see, this technique has the flavor of graphical transformation languages.

In Milner's original approach, the program portion of the machine configuration was a textual representation of the remainder of the program. Because text is inherently ordered, and pattern matching is well-understood, the actual mechanics of rule application, particularly the matching of program and computation step conditions and the consequent transformation of the match program to conform to the computation step result, was straightforward. In the case of VIPR, where the programs are two-dimensional diagrams, and the computation steps are transformation rules, presentation of the rules in a manner that is at the same time formal, intuitive, and concise, was much more difficult.

In order to specify graphical transformations, we use a new graphical formalism denoting a computation rule, which consists of a pair of graphical templates separated by a $\triangleright$ symbol. If the current program snapshot conforms to the template to the left of the $\triangleright$, then after the computation step represented by the rule, the snapshot will conform to the template on the right of the $\triangleright$. Every element of the current program configuration must correspond to some element (circle, arrow, or variable) of the left-hand template of a computation rule for that rule to be applicable. Since our semantics allows for nondeterministic programs, it is possible for a rule template to match a configuration in more than one way, although, as we shall see, only one rule may match at any given time.

Graphical elements in the rules are circles (actually, any closed shape except for the distinguished state object), the state object, and arrows; the spatial relationships between graphical elements in the rules that are considered relevant to pattern matching are containment, connectivity (between arrows and circles), and tangency (between circles). When several circles are tangent to a given circle, a clockwise relationship holds between the circles, and one of the tangent circles (the one closest to the top) is considered the first circle from the standpoint of the clockwise relationship. Section 4.4 discusses the simple rules of syntax in VIPR.

In addition to the graphical elements themselves (circles, arrows, state object), the rules specifying computation steps may contain meta-objects, including graphical variables, dotted arrows, and label variables. Capital letters represent graphical variables, and denote all circles in the current program configuration with the given containment and tangency characteristics relative to the other graphical elements in the rule (or, more precisely, to the graphical elements in the program configuration to which the graphical elements in the rule correspond), as well as all arrows connecting them. For example, if figure 4a represents a program configuration, and figure 4b represents the left-hand template of a computation rule, then the two state objects s correspond, as do the pairs of circles labeled a and b. In addition, the graphical variable R in figure 4b corresponds to circles c, d, and e.



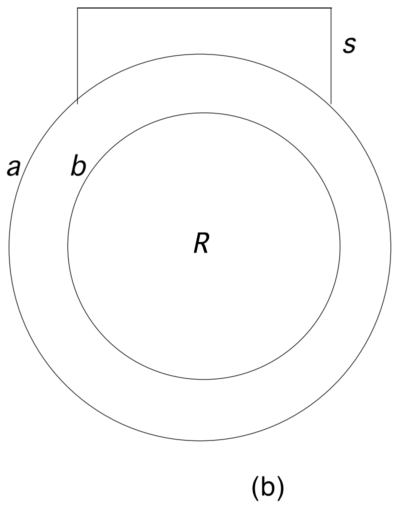


Figure 4: (a) Possible rule configuration
(b) Corresponding rule template

Dotted arrows denote zero or more arrows in the configuration connecting the elements to which they are attached. If a dotted arrow is bold, then the elements corresponding to the graphical variable the arrow enters must all be reachable from some graphical element in each of the graphical variables from which the bold dotted arrows originate, and each bold dotted arrow denotes the set of all arrows in the configuration that connect the elements at the source end of the dotted arrow to the elements at the destination end. (Reachability will be defined momentarily.) If the arrows are not bold (in this paper, we call such arrows light), reachability constraints do not apply. For example, in figure 5, all the elements corresponding to variable R_3 are reachable from both R_1 and R_2, while all elements in R_4 are reachable from R_1 and not R_2, and it is possible, but not necessary, that there be arrows from R_3 to R_4, because the connecting arrow is light. As a notational convenience, a pair of dotted arrows connecting two graphical objects or variables in both directions may be denoted by a double-headed arrow.

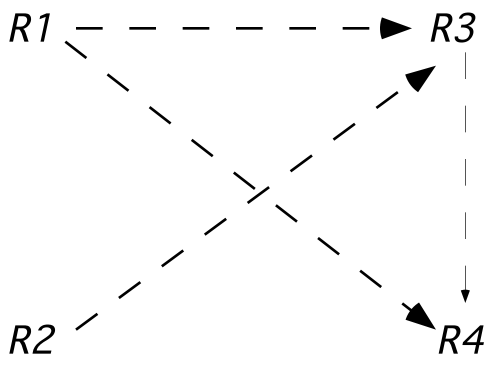


Figure 5: Connectivity with graphical variables and bold dotted arrows

If a dotted arrow or a graphical variable appears on both the right and left sides of a transformation rule, it represents the same collection of objects in each case. If an arrow (dotted or not) appears on both sides of a transformation rule, it is assumed to connect the same elements on each side of the rule. If a variable that appears on the left-hand side of a rule appears two or more times on the righthand side, the set of objects represented by that variable is duplicated as many times as the variable is duplicated. In addition, all arrows entering and leaving the variable, and the connection points, are duplicated on each set of duplicated objects.

Closed figures may possess textual labels. which reside at either of two distinguished sites (the guard site and the action site) on each ring. The guard and action sites are conventionally placed in the upper left and upper right quadrants of a circle, respectively. In our semantic rules, textual labels may be represented by label variables, which are represented by the lower case strings **guard** and **action**.

The state object may also be labeled — in this case, with a representation of the current values of program variables. Milner's operational semantics also represents state outside the framework of pattern matching, so we do not feel that our decision to do so is a drawback from the standpoint of formal semantic specification.

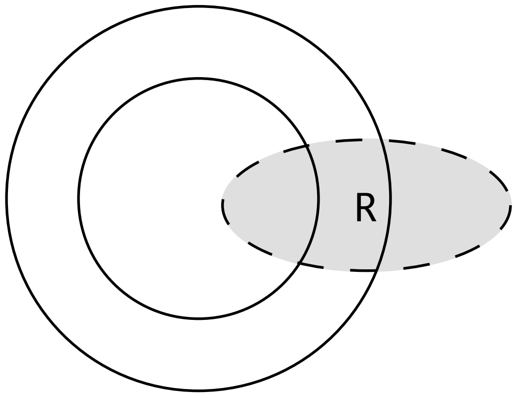
In order to simplify the specification of the rules, we assume that all program configurations on which the rules act are canonical, that is, all circles in such configurations are reachable from the state circle, or simply reachable. Reachability is specified by this simple inductive definition:

- The outermost circle, attached to the state object (the state circle) is reachable.
- Any circle immediately contained within a reachable circle is itself reachable.
- Any circle connected by an arrow from a reachable circle is itself reachable.

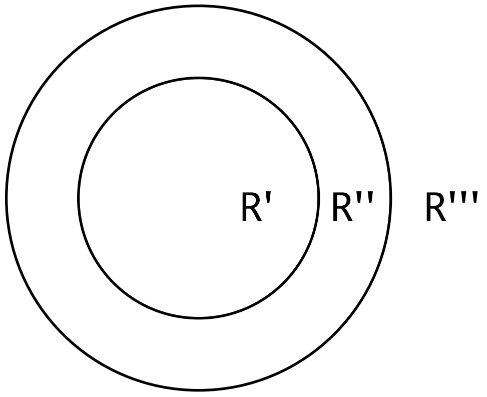
In practice, if a circle is reachable, then it is potentially executable through merging with the state circle as described in section 3. (Our definition of potentially executable is purely graphical, and disregards possible constant values in guards. For example, the **else**-part of a conditional construct whose guard is always **true** is considered potentially executable.) The requirement that diagrams be canonical is not a real restriction since any diagram may be made canonical by removing unreachable elements. This does not affect the meaning of the program configuration since the removed elements would not have been executed in any case.

We shall show later that each computation rule preserves canonicity, and that consequently any configuration derived from a canonical initial configuration is itself canonical. This result will validate our earlier assertion that rules only act on canonical configurations.

The above descriptions may seem complex, but in practice they are quite intuitive and easy to understand, as the rules below will show. The same rules could be described textually through multisets of elements and spatial operators, but the graphical approach seems more intuitive. Although the graphical formalism is powerful, it is not sufficient for certain specialized constructs, particularly those concerned with specifying the recursion rule. We introduce a small amount of extra notation in our specification of the recursion rule in section 4.2. Also, due to the need to cover the entire configuration, the number of light dotted arrows present in rules increases exponentially as the number of graphical elements increases. We could reduce this complexity by eliminating the requirement that the entire configuration be covered by the template and introducing negative conditions into the template, but this introduces additional semantic complexities. We address this problem by introducing a form of graphical shorthand that allows us to refer to several sets of graphical elements at once. This allows us to reduce the number of dotted arrows, and additionally allows a form of disjunction that allows us to reduce the number of rules required. For example, in the figure below:



the shaded region marked R refers to the three regions R', R'', and R''' below



and an arrow out of (or into) a shaded region (R, above, for example) represents an arrow out of (resp. into) any combination of constituent sets of graphical elements. (In the example above, it would represent an arrow out of or into R', R'', or R''', where the or relationship is not exclusive.) Bold, light, dotted, and double-headed arrows have the same meaning as previously described.

As an example of workings of the graphical transformation rules, consider the rule given in figure 6, representing the rule of sequencing. This is a fairly simple rule but exercises many of the features of our scheme. The rationale behind the rule will be explained in the next section.

The precondition of the rule, to the left of the ▹ symbol, contains a state object tangent to a ring. Another ring is contained inside the first ring. The latter ring has two labels, guard and action, at the appropriate label sites. These labels are actually variables, and stand for any guard and any action, respectively.

The rule precondition contains three variables: R, R', and R". R denotes all circles inside the circle labeled with a guard and an action, as well as all arrows between them. R' denotes all circles outside the state circle and reachable from elements of R (because of the bold dotted arrow from R to R'). Similarly, R" denotes all circles between the labeled circle and the state circle reachable from R. It is possible that the configuration may include arrows between R' and R" in either direction; these are denoted by the light dotted arrows. (Note that, for syntactic reasons, there may be no arrows from R' to R, or from R" to R. This is explained in section 4.4.) Also because the template must cover all the elements in the configuration, there may be no other circles immediately inside the state circle and no circles unreachable from R. (Note that the light dotted arrows, while required to make the rule perform properly, do not really contain any information on the "meaning" of the rule. As the number of elements in the rule increases, the number of dotted arrows increases exponentially, and makes the rules difficult to read. Therefore, all subsequent rules will be given without most dotted arrows; the complete rules are provided in an appendix.)

When a rule's precondition template is matched, the configuration is transformed to conform to the postcondition template. Thus, any object in the configuration corresponding to a precondition object that is not present in the postcondition must vanish, and objects that are in both the pre- and postconditions are unchanged. Similarly, collections of graphical objects represented by variables either vanish (if the corresponding variable is in the precondition but not the postcondition), are unchanged (if the variable is present in both parts of the rule), or are copied (if a new instance of the variable appears in the postcondition); a collection corresponding to a variable is never partially altered. Placeholders representing labels are either altered or left unchanged depending on the changes to them between the pre- and postconditions. Graphical elements in the postcondition but not in the precondition are added to the configuration when the rule is applied.

Therefore, using the figure 6 as the example, if the precondition of the rule is satisfied, the ring with the guard and action is removed (since the ring is not part of the postcondition), and the action is applied to the current state, denoted by the placeholder label s, leading to a new state. The collections of objects associated with the variable R is unchanged. Similarly, objects associated with variables R' and R" are unchanged, and the arrows connecting them are also unchanged.

One interesting aspect or our semantic technique is that it represents the previously mentioned class of graphical transformation languages exemplified by BitPict [8], ChemTrains [2], and Vampire [11]. Unlike VIPR, where a snapshot of an executing program contains both program state and all the remaining instructions, these languages provide for a graphical configuration representing the program state and a separate set of graphical rules representing the program. Typically, running a program in one of these languages consists of observing the changing program state. In the context of a language like ChemTrains or Vampire, a snapshot of a VIPR program may be considered the graphical program configuration, and the set of semantic rules presented in the next section may be thought of as the program. Thus, a language like ChemTrains or Vampire may be used as a metaprogramming environment for the specification and implementation of VIPR.

4.2 Semantic Specification

The first VIPR rule is the rule of sequencing, which is given in figure 6. The rule states that if the outermost inner ring (that is, inside the outer state ring) has a guard that evaluates to **true**, and there is no arrow connecting the ring from the outside (such an arrow would suggest that some other ring would come first according to the substitution rule), the ring is removed and the state is altered in accordance with the action. The remainder of the rule suggests that the ring has some (possibly empty) contents, and these contents may connect to other elements inside or outside the outer ring. (The template does not include any arrows from R” to R; they would be syntactically incorrect. See section 4.4 for a further discussion of this.)

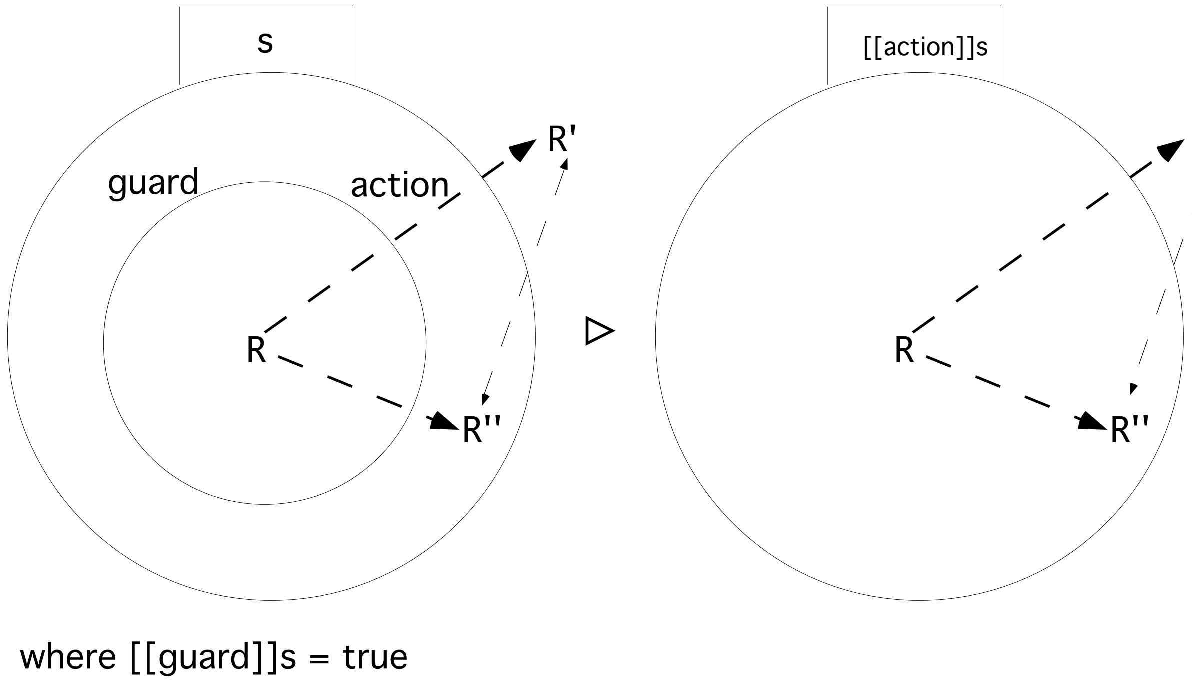
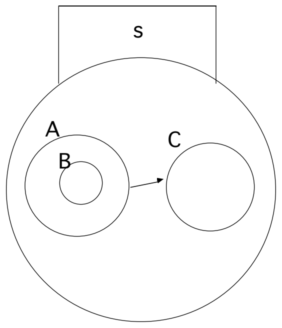


Figure 6: Rule of sequencing

The rule of sequencing is designed to suppress execution of any ring that has an arrow connecting to it from outside. The rationale behind this is that arrows define an ordering, and that anything pointing to an object must execute before the object itself executes. Thus, in the diagram



object A is a candidate for execution according to the rule of sequencing, but object C is not.

The second VIPR rule is the rule of selection, shown in figure 7. When there is more than one ring immediately contained within the outer state ring (such that, similarly to the rule of sequencing, none of the rings are contacted by an arrow coming from the outside), exactly one such ring whose guard evaluates to true is chosen, and the remainder are discarded. The rule of sequencing applies to the result. The rule also specifies that any constructs connected to the removed construct should also be removed, unless they are also connected to the retained construct. (R2 denotes the removed construct, and R2' denotes constructs reachable from the removed construct but not also from R and which are also removed.) This is done to preserve canonicity.

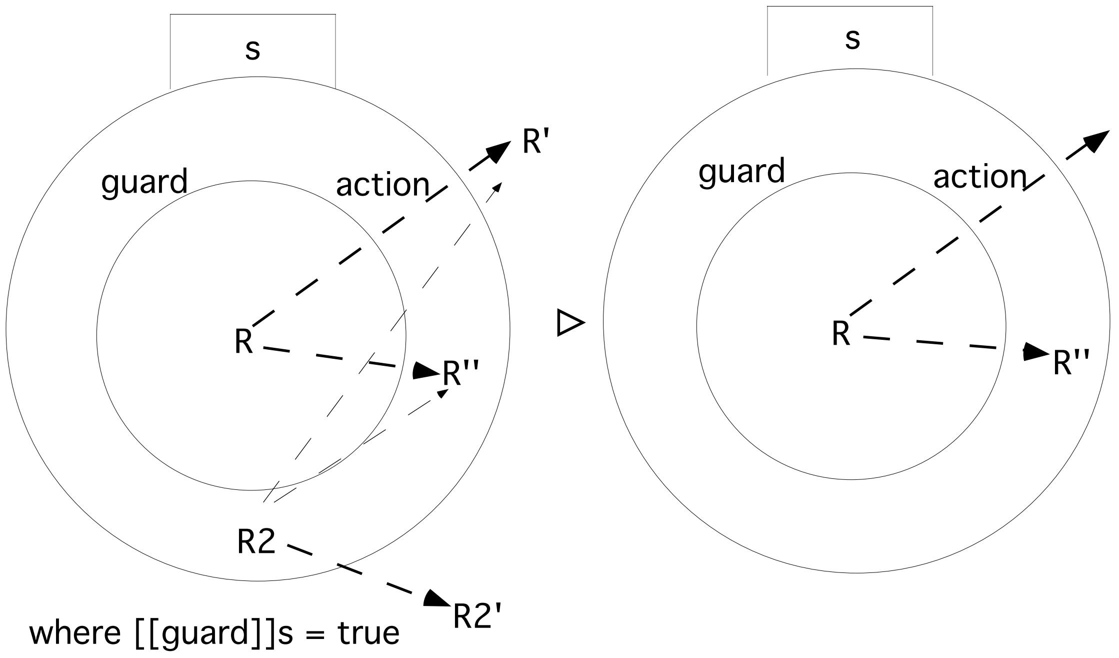


Figure 7: Rule of selection

The third rule, given in figure 8, is the rule of recursion. Although the rule only applies to direct recursion, other types of recursion may occur, although they are handled by the substitution rules. The rule states that if the outermost executable ring is contacted by an arrow coming from inside, we must copy the construct in place of the ring where the arrow originates before we may proceed. This rule allows us to execute a loop an unlimited number of times. If we did not have this rule, the rule of sequencing would eliminate the ring before we had a chance to copy it. In the graphical specification of the rule of recursion, we see where the basic graphical specification method given in section 4.1 falls short. In specifying the transformations connected with the recursion rule, we wish to indicate that a variable corresponding to a group of graphical elements contains a particular graphical element, and to indicate that, as part of the transformation, this element is replaced. In figure 8, the dotted and cross-hatched circle labeled R represents a collection of elements including the specific ring from which the arrow originates. This ring contains cross-hatching in the opposite direction. In the right-hand part of the rule, that inner ring has been replaced by another copy of R, indicated by cross-hatching in the same direction as the ring that was replaced. Note also that the new dotted arrows in the postcondition template are copies of the original dotted arrows in the precondition (and which also survive in the postcondition).

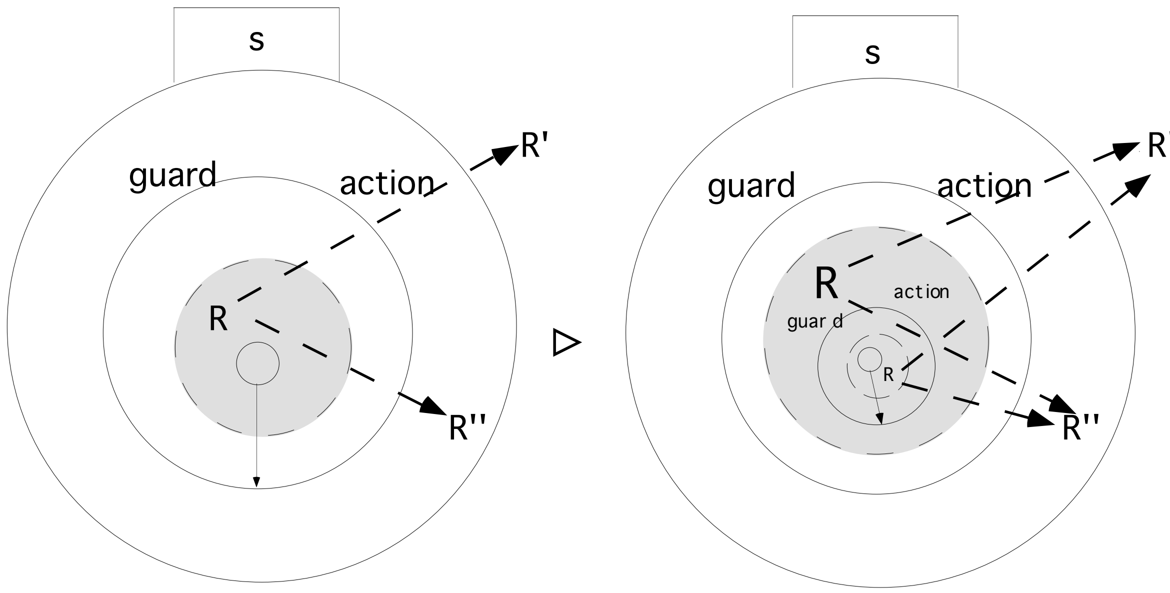


Figure 8: Rule of recursion

The fourth rule (figure 9), is the rule of substitution. This rule is used to model procedure calls and gotos. The rule states that, if the outermost ring has an arrow connecting it to some other structure, a copy of that structure (including its contents) is substituted for the original ring. Any contents of the original ring are copied into the new structure. In addition, if there are small rings on the inside of both the original ring and the destination ring, and there are equal numbers of such small rings, the two sets of small rings, together with their contents and any connection arrows, are copied into the new construct. The small rings are matched beginning with the two small rings closest to the top of their respective circles (and to the right, if no small circle is immediately at the top) and corresponding small rings are matched in a clockwise direction. This, as we will see in section 5, and in the last two rules, is how parameter passing and return continuations are modeled.

Note that, in order to preserve canonicity, if, after performing the copy operation and eliminating the substitution arrow, there are no longer any arrows pointing to the element copied, we must eliminate the original element. Otherwise, it must remain. Therefore, we provide two versions of figure 9, given as figures 9a and 9b. Also note the use of the region/disjunction construct described in section 4.1.

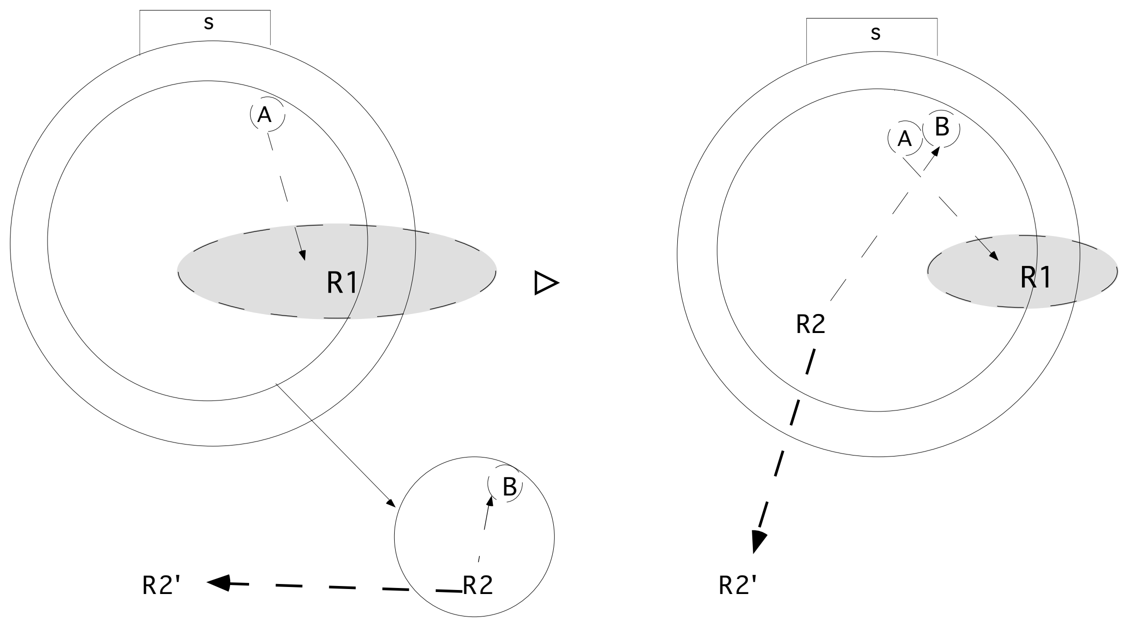


Figure 9a: Rule of substitution (I)

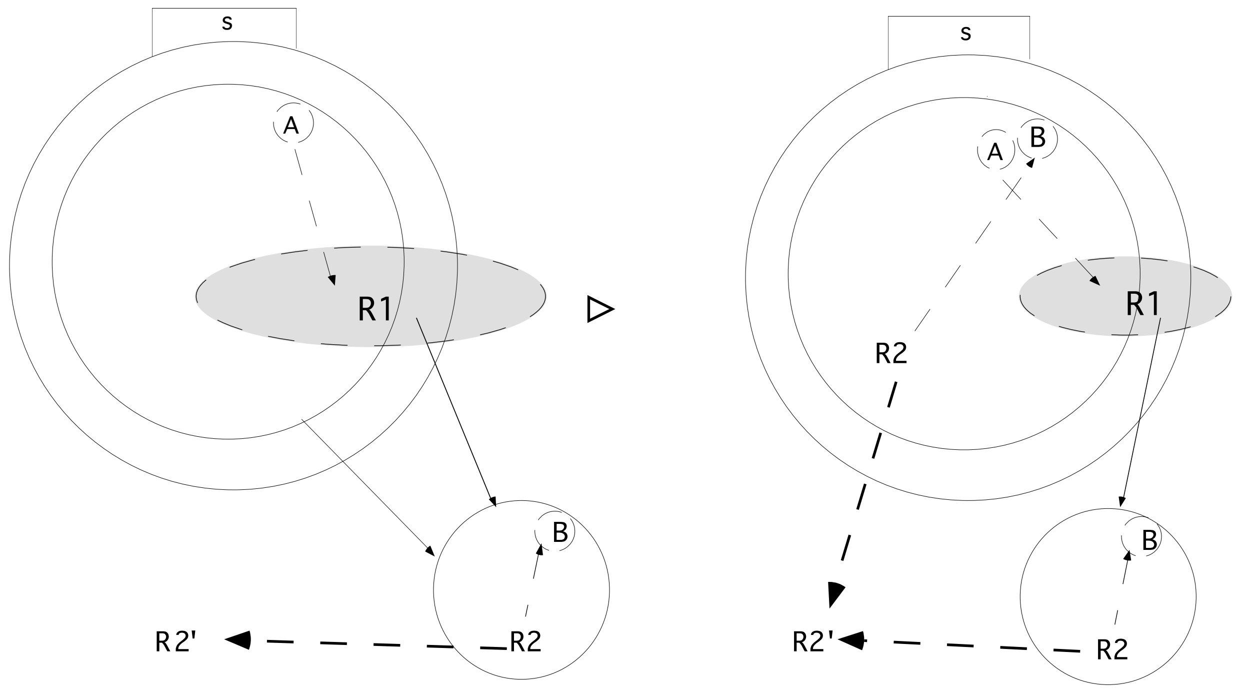
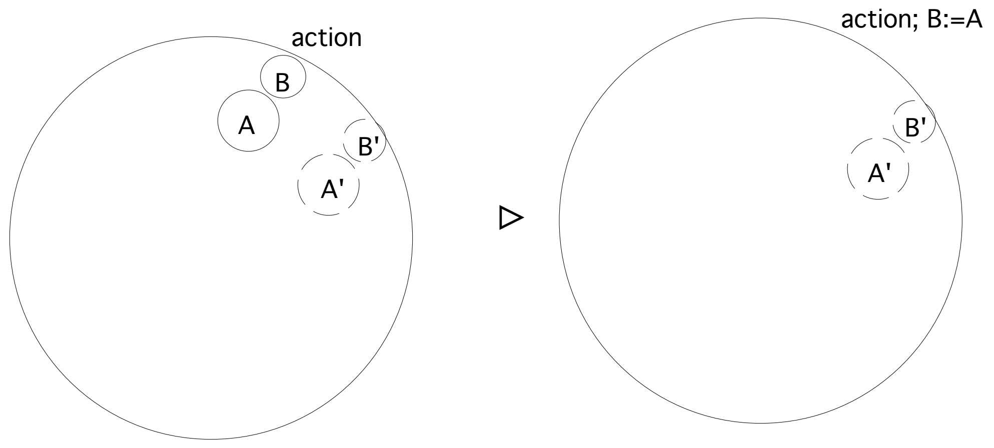


Figure 9b: Rule of substitution (II)

The fifth and sixth rules are rules of parameterization. The first rule of parameterization (figure 10) states that when a value is passed to a parameter, the argument value is assigned to the parameter. Thus, parameters in VIPR are passed by value. The rule binds parameters in a clockwise order, so that if there is a parameter pair at the most clockwise position, the parameter and the argument are bound. The rule will be repeatedly applied until there are no more parameters to bind. The dotted circles in the rule denote zero or more pairs of rings that are tangent to each other, along with their respective labels. Note that we have not specified how scoping is accomplished. Currently we simply assume that variables are appropriately and automatically renamed on substitution. Clearly, a more formalized approach will eventually have to be taken.

The reader will note that the parameterization rules are written in a slightly different style from the previous rules, in that they cover only a part of the configuration. This was done for readability; the complete rules are given in the appendix. However, since substitution may only occur if it involves the ring immediately within the state ring (as specified in the rule of substitution), parameter binding may only occur at that point, too, and the context of the parameterization rules is unambiguous.



where B is identifier and A is expression

Figure 10: First rule of parameterization

The second rule of parameterization (figure 11) states that arrow chains are combined into a single arrow. As we shall see in the next section, this is how procedure returns are modeled.

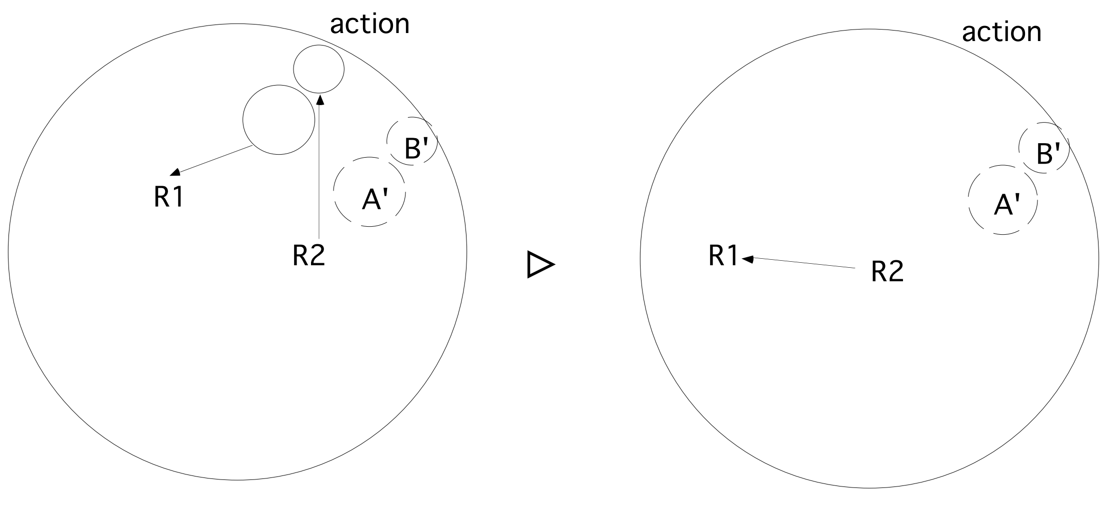


Figure 11: Second rule of parameterization

Finally, the meaning of an entire VIPR program is the state that results from some initial configuration when the program may be reduced to a state ring with nothing inside it. In other words, if A $\triangleright^*$ B means that configuration A derives configuration B in zero or more computation steps, and {A}s denotes the meaning of configuration A with initial state s, then

$$\{A\}s = \begin{cases} s', & \text{where the derivation of figure 12 holds} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

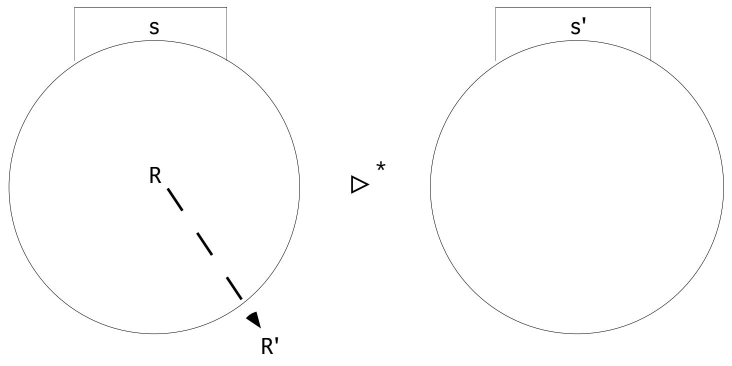


Figure 12: Termination of a program

We note that, although there is deliberate ambiguity in the way in which the rule of selection template partitions the elements of the configuration to which it is applied (thereby allowing nondeterminism), it is easy to see that there is no ambiguity concerning which rule to apply to a given configuration, since every rule's template possesses a topological property that distinguishes it from all other rules. Consequently, for any given initial configuration, there is always exactly one execution sequence of rules, assuming that the nondeterminism of the selection rule is always resolved in the same way.

4.3 Canonicity

We mentioned earlier that we assume that the rules only operate on canonical configurations; that is, there are no unreachable clusters of elements floating around in the configuration that must be accounted for in the various transformation rules. It is clear that the initial configuration can be made canonical by simply removing all unreachable graphical elements. It remains to be shown that every intermediate configuration derivable from a canonical initial configuration by the transformation rules is itself canonical, and that this can be expected at each application of a rule. It suffices to show that the rules preserve canonicity; that is, given a canonical input configuration, each rule will produce a canonical output configuration. We will demonstrate this property for the rules of sequencing and selection, but the same techniques can be used to prove the result for the other rules.

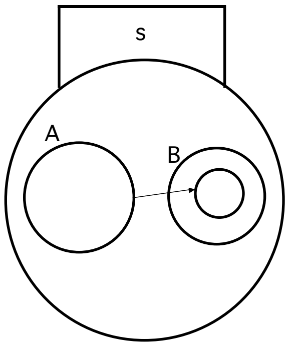
For the rule of sequencing, assume that the input configuration is canonical. That means that each of the elements in R' and R" is reachable, and in fact, the arrows from R to R' and R" "carry" this reachability. If the sequencing rule is applied, R, R', and R" still exist, as do all the arrows from R to R' and R". Thus, all the elements in R' and R" are still reachable, and the resulting configuration is canonical.

For the rule of selection, again assume that the input configuration is canonical. Thus, R, R', R", R2, and R2' are all reachable, and the bold dotted arrows represent the configuration arrows that guarantee that R', R", and R2' are reachable. After the rule is applied, R2 and R2' are removed, but R, R', and R" are still present. R is clearly reachable, and the bold arrows from R to R' and R" represent a set of arrows from some circles in R to elements in R' and R" that cause each element of these latter sets to be reachable from R. Thus, all the elements of R, R', and R" are reachable, and the resulting configuration is also canonical.

4.4 A Note on Syntax

It is not our intention in this paper to specify formally the syntax of VIPR programs; numerous other papers have discussed the specification of two-dimensional syntax. Informally, the syntax of VIPR is quite simple, and can be observed from the examples seen so far. There is a unique state object, attached to the outside of a circle, and not contained inside any other circle. Circles may be contained within other circles, or may be outside one another, but may have no other relations, except that two circles may be tangent to each other. Arrows must begin and end in a circle.

One syntactic rule that may not be obvious concerns the way in which arrows may connect circles: an arrow may not cross a circle (other than the state circle) from the outside to the inside. Therefore, the following diagram is syntactically invalid:



There are two reasons for this rule, both semantically related. The first is that to allow such diagrams would greatly complicate the formal semantic specification, since a new notational convention would be needed to indicate that some, but not all, the elements of B are reachable from some elements of A. The old notation of the bold, dotted arrow does not cover this case. The second reason is that allowing such arrows would affect our definitions of reachability and canonicity and would invalidate some of our canonicity results. For example, in the above diagram, is the outer ring of group B reachable? According to the rules of sequencing and selection as currently stated, that ring will never be executed. According to our definition of reachability, though, that ring is reachable. Since our definitions of reachability and canoniicity are otherwise serviceable, we choose to explicitly rule out this unusual case.

Not allowing such structures is no great loss, since they are analogous to jumps into blocks and other nested program structures in conventional textual languages. Such constructs are semantically problematic, and allowing them invalidates invariant properties of program structures that would otherwise hold. Consequently most languages with structured control disallow such jumps, and so do we.

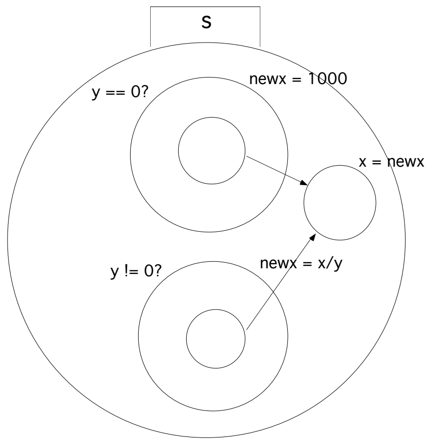
5 Modeling High-Level Imperative Constructs in VIPR

The primitives provided by VIPR, and the set of transformation rules, may be used to model a large number of higher-level constructs that are found in conventional imperative languages. These primitives could also be used to model new constructs that may not appear in any language. We show how a number of conventional constructs may be modeled in the sections below.

5.1 Conditional and Unconditional Branching

Conditional branch structures are represented in VIPR through use of the rule of selection. To model conventional deterministic conditionals, we must make sure that the guards of the candidate rings are constructed so that exactly one must be true when the construct is executed. For example, in the program in figure 1, 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. If more than one guard is true, the result is non-deterministic. If none of the guards are true, the program’s meaning is undefined, since it will never reach a terminating configuration according to the transformation rules of the language. Thus, because we allow for non-determinism, we support a somewhat different conditional semantics than Dijkstra's guarded-if construct [6]. With our semantics, and the proper programming discipline, 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 13 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 the rule of substitution to join two execution paths. First, note the guards attached to the two statements that are conditionally executed, just as we saw in the Figure 1 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. 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.

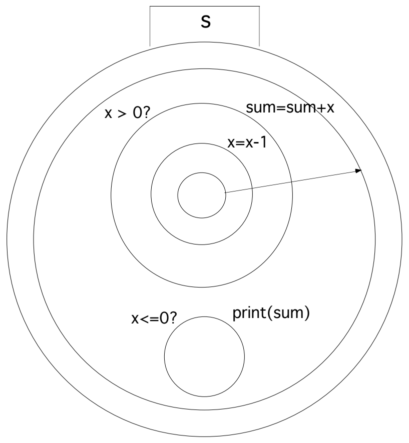


```
if (y == 0)
   newx = 1000;
else
   newx = x/y;
x = newx;
```

Figure 13: Representation of a conditional statement in VIPR, and its C equivalent

5.2 Iteration

With the semantics of conditional execution and the rules of substitution and recursion, the representation of a while statement follows immediately. Figure 14 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, thus denoting an embedded recursive structure. We also note that most formal semantic treatments of iteration employ a similar recursive model.



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

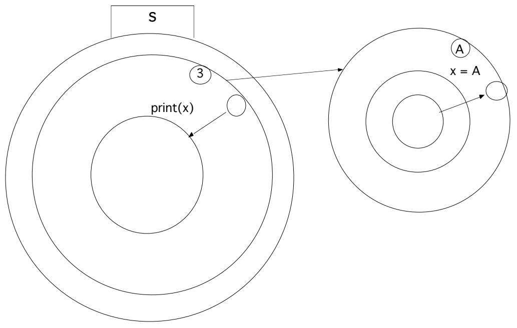
Figure 14: Representation of a while loop in VIPR, and its C equivalent

5.3 Procedure Call and Return

Procedures are sets of nested rings outside the main procedure's rings (see Figure 15). To model procedure invocation in VIPR, we employ the rule of substitution, and the two rules of parameterization. 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.

Parameters are passed through small auxiliary rings, which appear in corresponding positions in the procedure call ring (for the arguments) and the procedure definition ring (for the formal parameters). The return continuation itself may be thought of as another kind of parameter. Also, since a return statement is simply another type of procedure call or parameterized goto, our model allows function return values to be passed back within the general parameter framework.

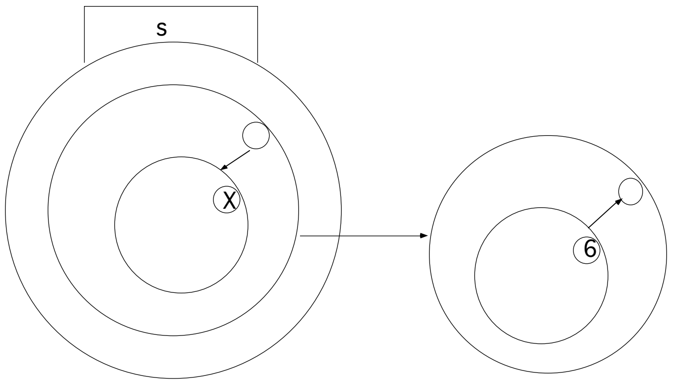
To understand the visual representation of procedures, consider the example in figure 15. In this example, there are two procedures: main, which contains a call to procedure p, and p, which assigns a parameter value to a local variable and returns. In VIPR, the return continuation is represented as an arrow attached to one of the small parameter rings. The corresponding return statement is represented by a ring with a substitution arrow pointing to the return continuation parameter. Dynamically, this results in a chain of arrows from the return statement to the statement to be executed after the procedure call's return. The second rule of parameterization consolidates this chain into a single substitution arrow.



```
void main()
{
   p(3);
   print(x);
}

void p(A)
int A;
{
   x = A;
   return;
}
```

Figure 15: Procedure call and return in VIPR

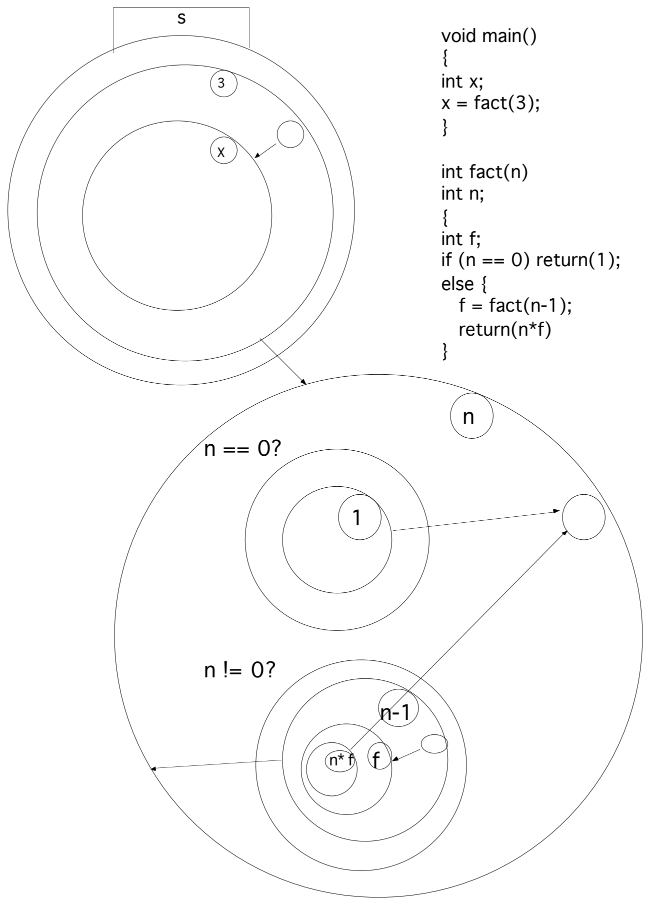
Figure 16 shows a simple function call and the manner in which values are returned.



```
int f()
{
return(6);
}

main()
{
int x;

x = f();
}
```

Figure 16: Function call with return value

Finally, figure 17 presents a VIPR program that models recursive factorial. It demonstrates the way in which result values may be returned by functions.



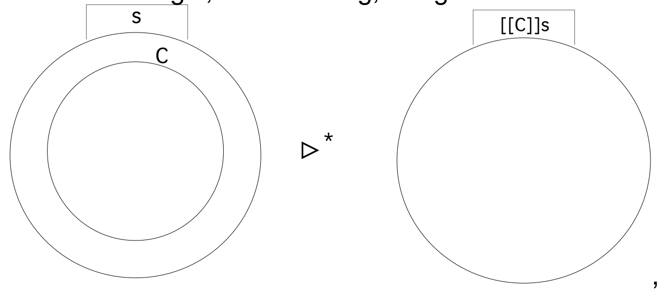
```
void main()
{
int x;
x = fact(3);
}

int fact(n)
int n;
{
int f;
if (n == 0) return(1);
else {
   f = fact(n-1);
   return(n*f)
}
```

Figure 17: Representation of a recursive factorial function

6 Equivalence

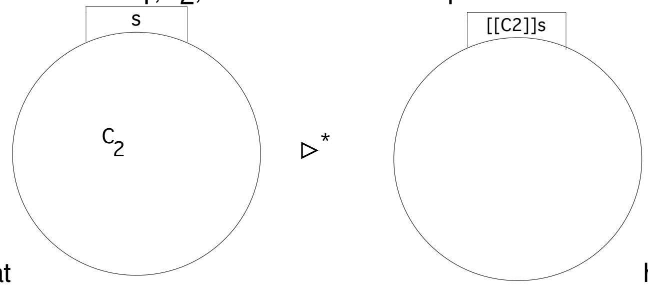
It is fairly straightforward to prove that the operational semantics of the high-level VIPR constructs described in section 5, employing the transformation rules described in section 4, are equivalent to the denotational semantics of the high-level constructs as conventionally described. We will perform such proofs for sequencing, conditionals, and iteration. We use structural induction in our proof.

The base case is the single, state-altering, assignment command. If C is such a command, then the derivation

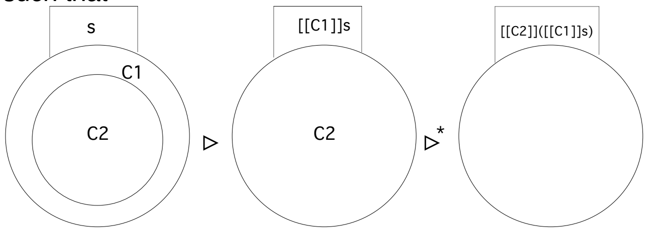


, along with the definition of the meaning of the program clearly indicates that the operational meaning of the left-hand configuration is [[C]]s.

For the sequential construct C_1;C_2, if we assume that C_1 is a unit statement, and C_2 is a compound

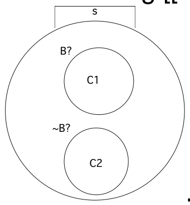


statement such that holds, we can clearly see that the



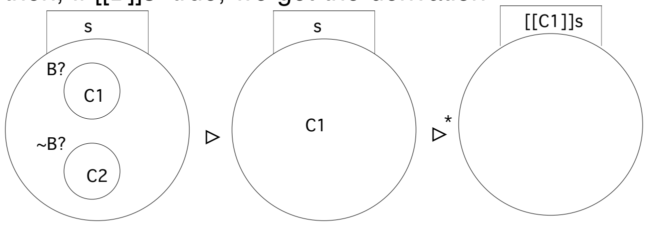
derivation is valid, and the meaning of the left-hand structure (corresponding to C_1;C_2) has the meaning [[C_2]]([[C_1]]s), which is the same as [[C_1;C_2]]s.

For the conditional statement **if** B **then** C_1 **else** C_2, we begin with the following configuration, where we can make the induction assumption that the configuration C_1 has meaning [[C_1]] and the configuration C_2 has meaning [[C_2]]:



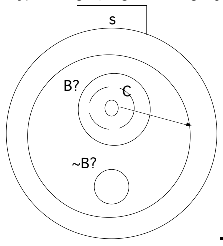
.

then, if [[B]]s=true, we get the derivation

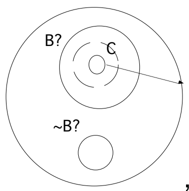


which indicates that the meaning of the left-hand configuration is [[C_1]]s when [[B]]s is true,. Similarly, when [[B]]s is false, the meaning of the construct is [[C_2]]s. This is exactly the meaning of the if-then-else construct.

Finally, we examine the while-do construct. **while** B **do** C may be represented in VIPR as the



configuration . If we assign the name C' to the subconfiguration



, through our understanding of the if-then-else construct and the substitution arrow, we can say that C' = **if** B **then** C;C'. This may be shown to be equivalent to **while** B **do** C. See [14] for one proof of this result.

7 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. The most significant contribution of this paper is the presentation of a completely visual formal operational semantics for the control structures of the language, including a powerful and flexible metalanguage for the specification of the semantic transformations. This semantics is simple and is in accordance with the intuitive and informal notions of execution semantics described elsewhere in the paper.

The main drawback to the formalism is the proliferation of arrows representing all the combinations of arrows in the configuration that may be there. Although this is not a problem for a simple rule like the rule of sequencing (rule 6), when the number of graphical elements increases even a small amount, the diagrams become cluttered and difficult to understand due to the exponential explosion in numbers of dotted arrows. We have handled this in the body of the paper by not representing these dotted arrows in most of the rules presented, since they do little to illustrate the meaning of the rules, but rather are there only for correctness. We have also provided some notational shorthand to make the full rules more manageable. The more complex rules in the appendix give the complete version.

Since one goal of semantic design is simplicity of the formalism, we would like to address this problem in the future. One possibility is to show how the complete rules may always be generated from simplified rules like the ones given in the paper. In this way, the simplest rules could always be used while the full and correct rules would be understood.

A second possibility is to eliminate the need for the dotted arrows representing optional rules by removing the requirement that the rule templates cover the entire configuration and that every graphical element in a configuration be accounted for in a rule. If a rule does not have to account for arrows in the configuration, then the arrows would not need to be represented in the rule. The problem with this approach is that there is no way to distinguish between graphical elements that do not happen to be in a particular place and graphical elements that must not be in that place. In the formalism we use in this paper, when we want to specify that the ring to be executed in the rule of sequencing must not have an arrow leading into it, we simply do not show any such arrow: if it is not there in the template, it cannot be in the diagram, if the template is to conform to the diagram. On the other hand, if every element in the configuration need not be accounted for in a rule template, then the fact that the template does not mention an arrow leading into the circle does not mean that there must be no such arrow: to specify this, we need new notation, and it is not clear if that new notation is simpler than what we have now.

As far as we know, there have been no previous attempts to apply graphical techniques to the specification of the formal semantics of a visual language. Similarly, we do not know of any previous attempts to apply visual techniques to the proofs of formal properties of the languages. The fact that the operational semantics are simple and intuitive is encouraging. Even though the underlying textual language might have a complex formal semantics (on the order, say, of C), our experience with VIPR shows that a completely visual language with simple semantics may be derived from it, and we expect that such a language might be simpler to learn and use.

References

[1] B. Bell & W. Citrin (1992) Simulation of Communications Protocols through Graphical Transformation Rules. In: Advanced Visual Interfaces - Proceedings of the International Workshop AVI '92 (T. Catarci, M. F. Costabile & S. Levialdi, eds.) World Scientific Publishing, Singapore, 208-222.

[2] B. Bell & C. Lewis (1993) ChemTrains: A Language for Creating Behaving Pictures. In: IEEE Symposium on Visual Languages Bergen, Norway, 188-195.

[3] F. P. Brooks (1987) No Silver Bullet: Essence and Accidents of Software Engineering. Computer **20**, 10-19.

[4] W. Citrin, M. Doherty & B. Zorn (1994) The Design of a Completely Visual Object-Oriented Programming Language. In: Visual Object-Oriented Programming (M. Burnett, A. Goldberg & T. Lewis, eds.) Prentice-Hall, Englewood Cliffs, NJ, to appear.

[5] W. Citrin, M. Doherty & B. Zorn (1994) Formal Semantics of Control in a Completely Visual Programming Language. In: IEEE Symposium on Visual Languages St. Louis, to appear.

[6] E. W. Dijkstra (1976) A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ, 217 pp.

[7] E. W. Dijkstra (1989) On the cruelty of really teaching computer science. Comm. ACM **32**, 1397-1404.

[8] G. W. Furnas (1991) New graphical reasoning models for understanding graphical interfaces. In: Human Factors in Computer Systems: CHI '91 Conference Proceedings New Orleans, 71-78.

[9] K. M. Kahn (1992) Towards Visual Concurrent Constraint Programming. Technical Report SSL-91-092, Xerox Palo Alto Research Center.

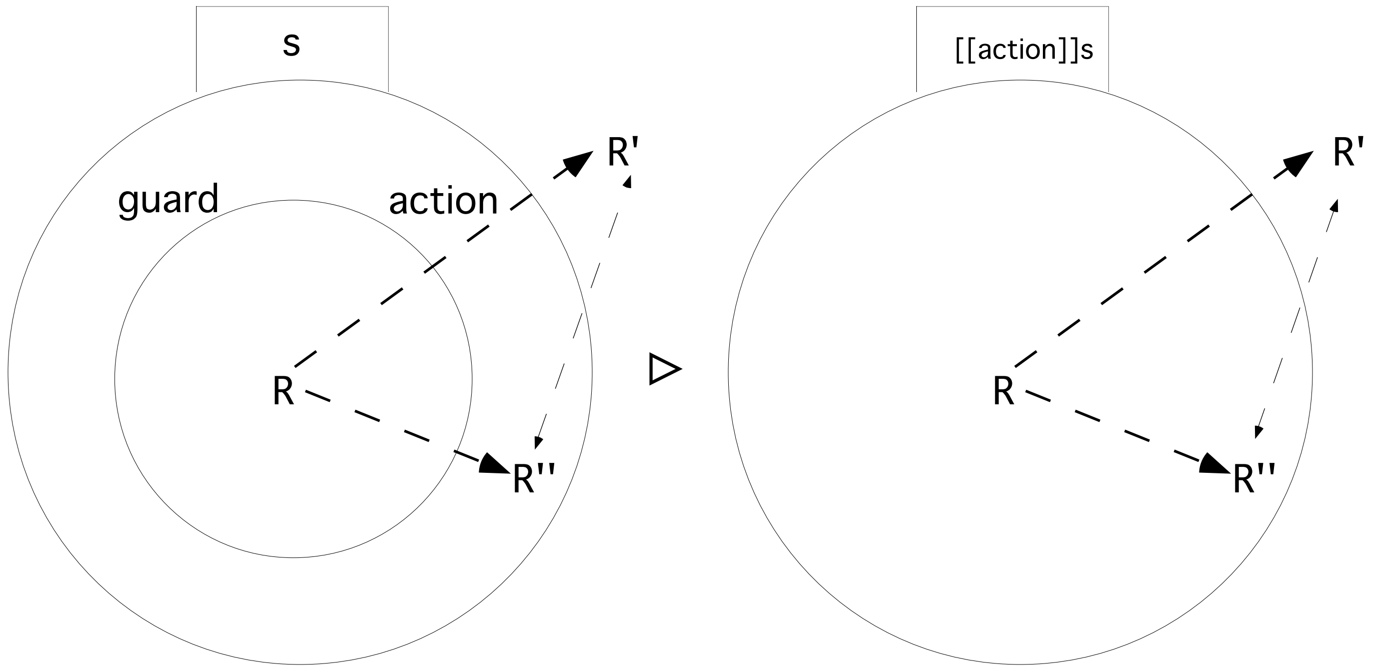
[10] K. M. Kahn & V. A. Saraswat (1990) Complete Visualizations of Concurrent Programs and Their Executions. In: IEEE Workshop on Visual Languages Skokie, IL, 7-15.

[11] D. W. McIntyre & E. P. Glinert (1992) Visual Tools for Creating Iconic Programming Environments. In: IEEE Workshop on Visual Languages Seattle, 162-168.

[12] R. Milner (1976) Program semantics and mechanized proof. In: Foundations of Computer Science II, part 2 (K. R. Apt & J. W. de Bakker, eds.) Mathematical Centre, Amsterdam, 3-44.

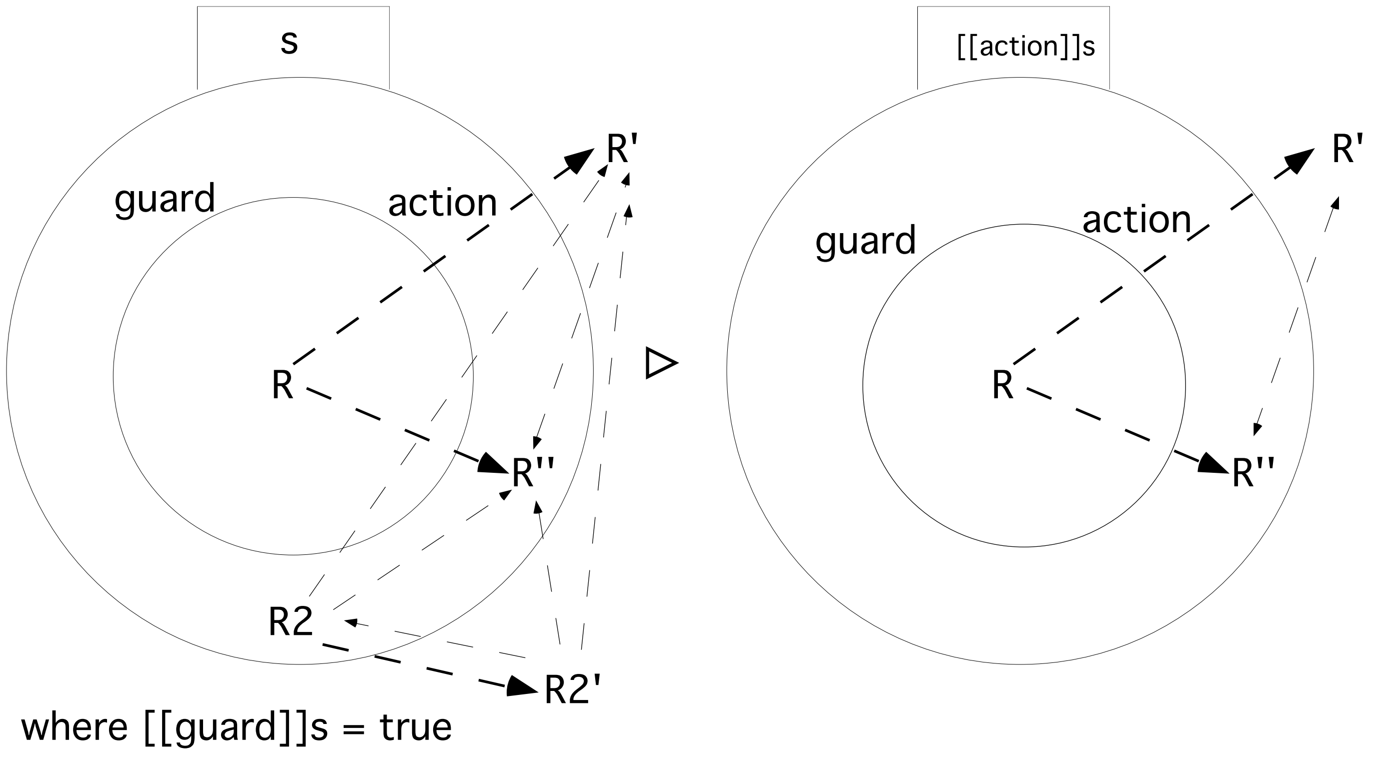
[13] V. Saraswat, K. M. Kahn & J. Levy (1990) JANUS: A step towards distributed constraint programming. In: North American Logic Programming Conference Austin, TX, 431-446.

[14] R. D. Tennent (1991) Semantics of Programming Languages. Prentice-Hall, New York, 236□pp.

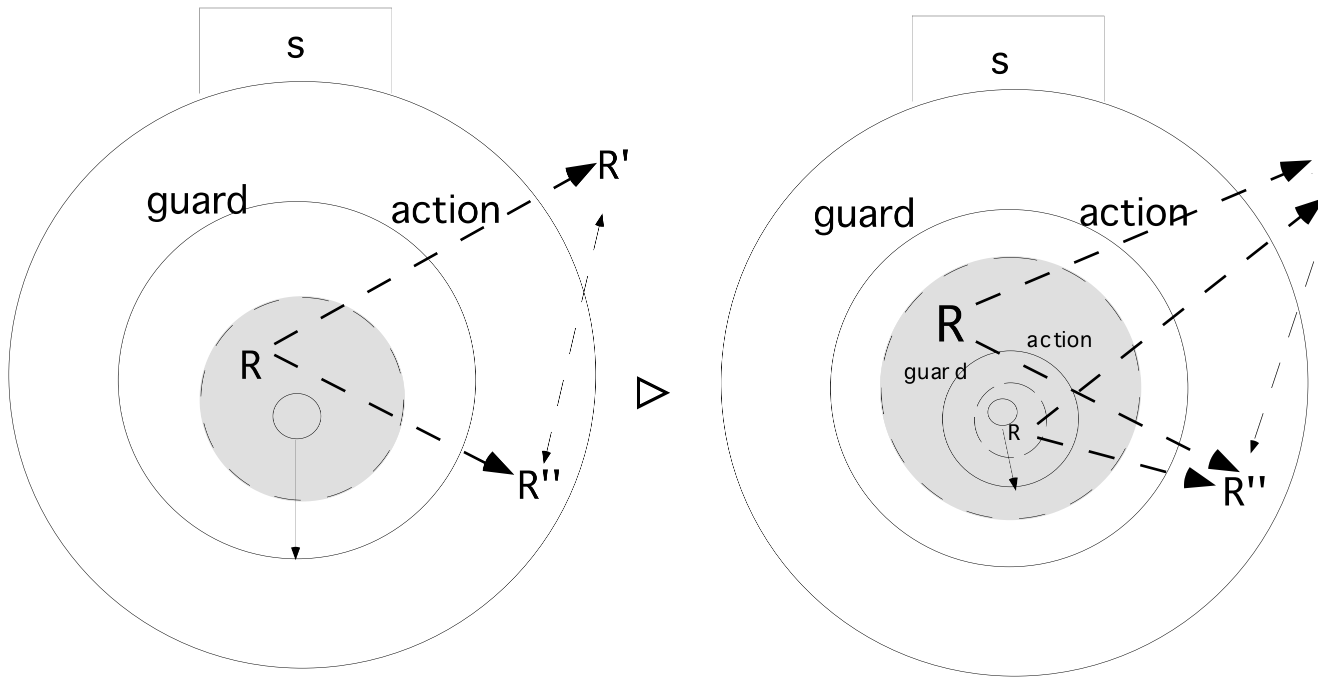


where [[guard]]s = true

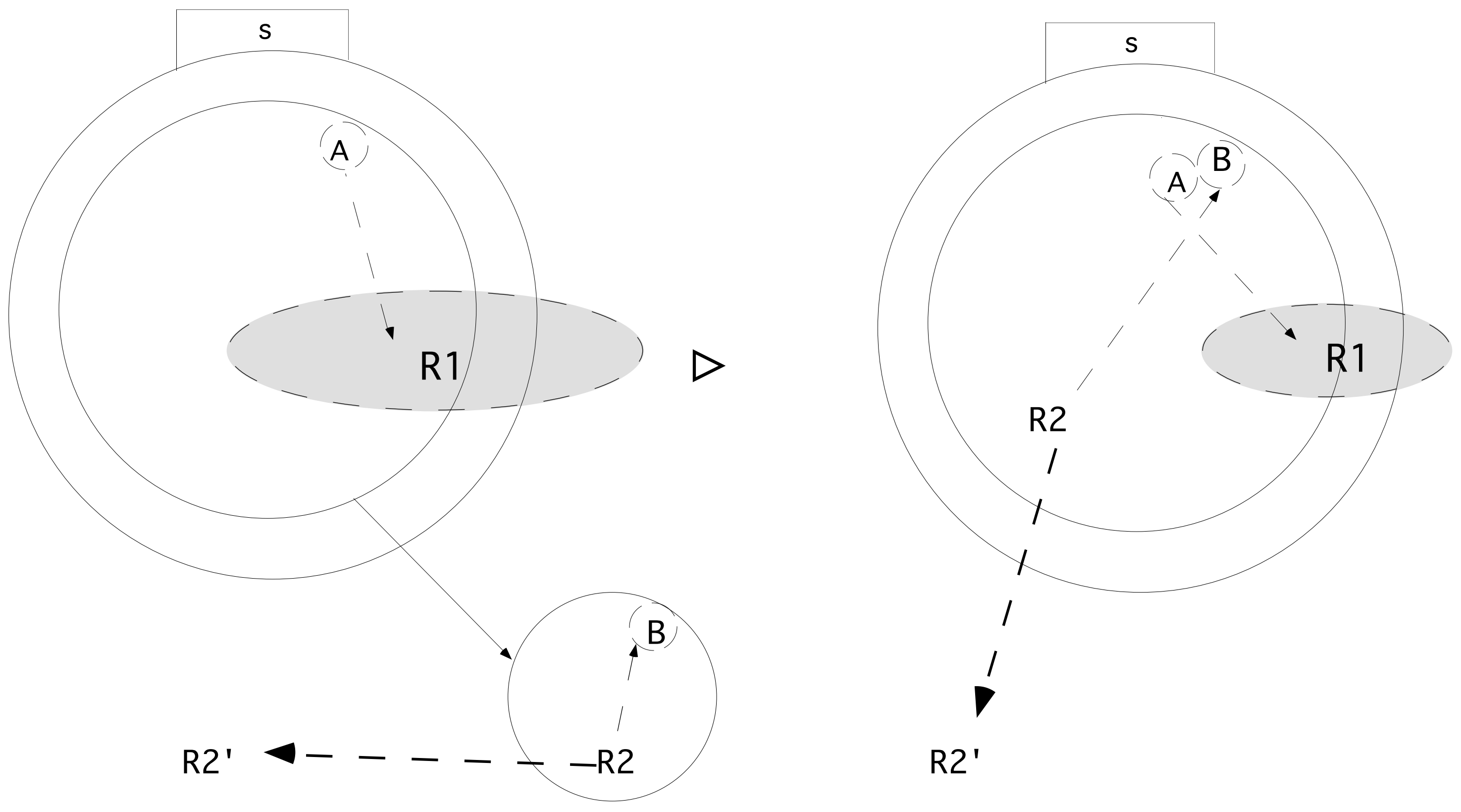
Rule of sequencing



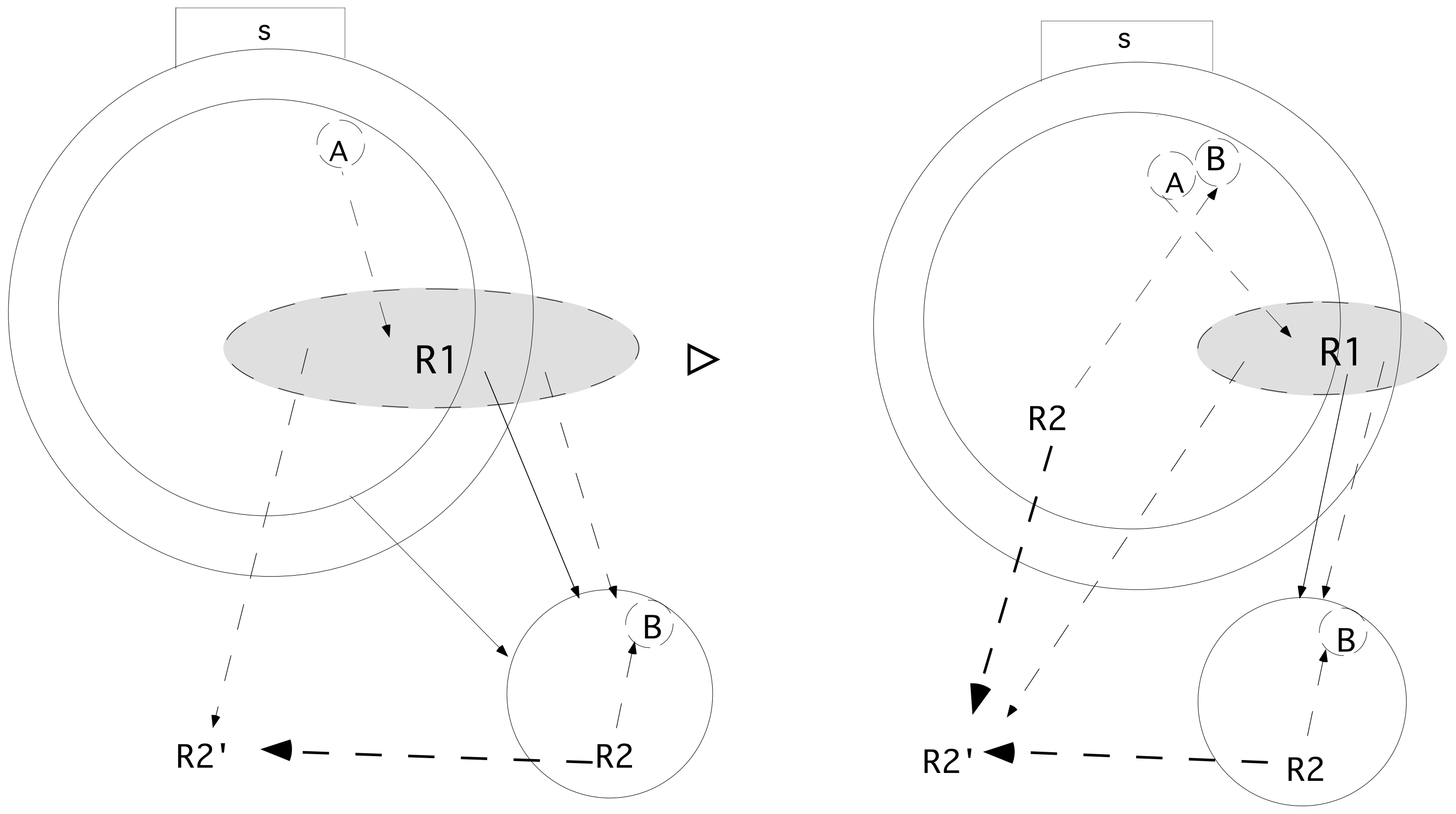
Rule of selection



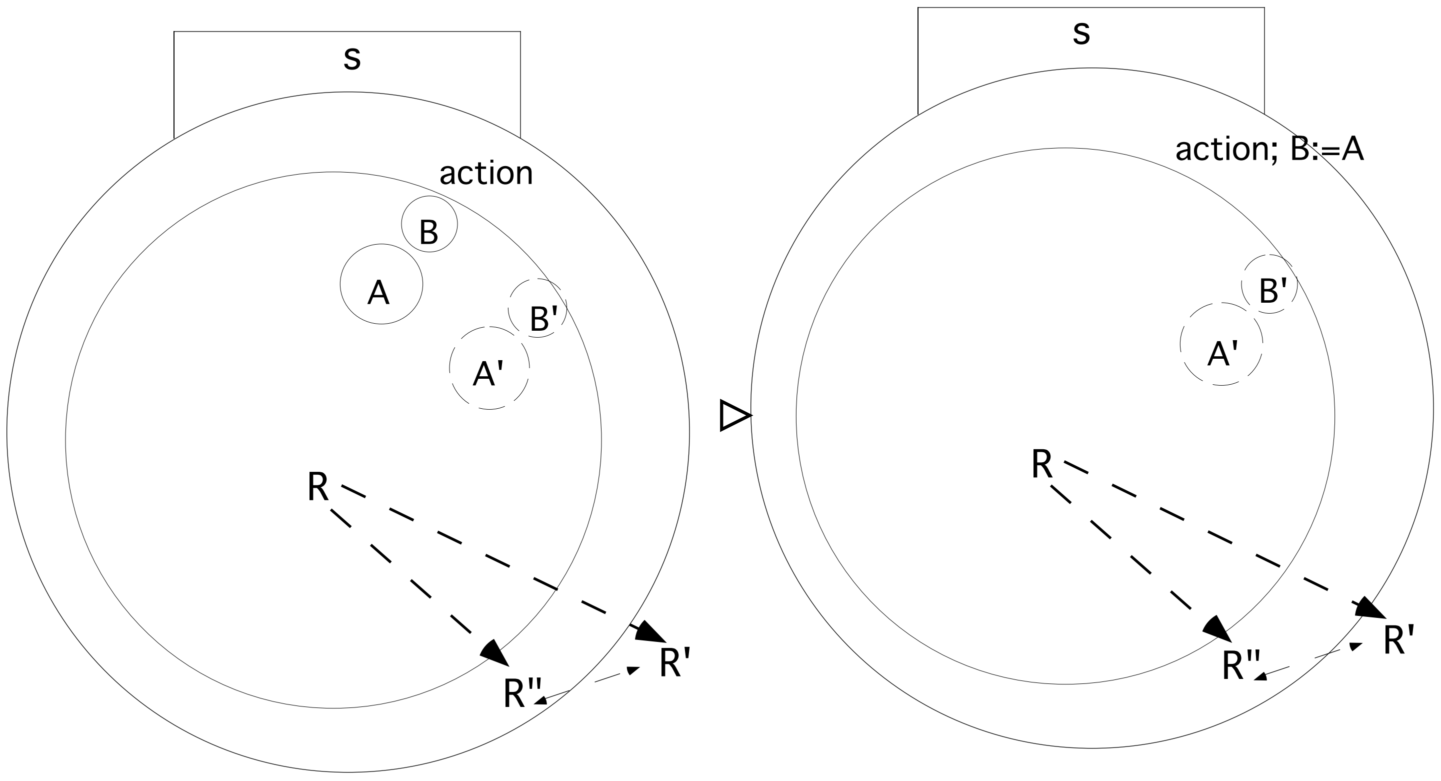
Rule of recursion



Rule of substitution (I)

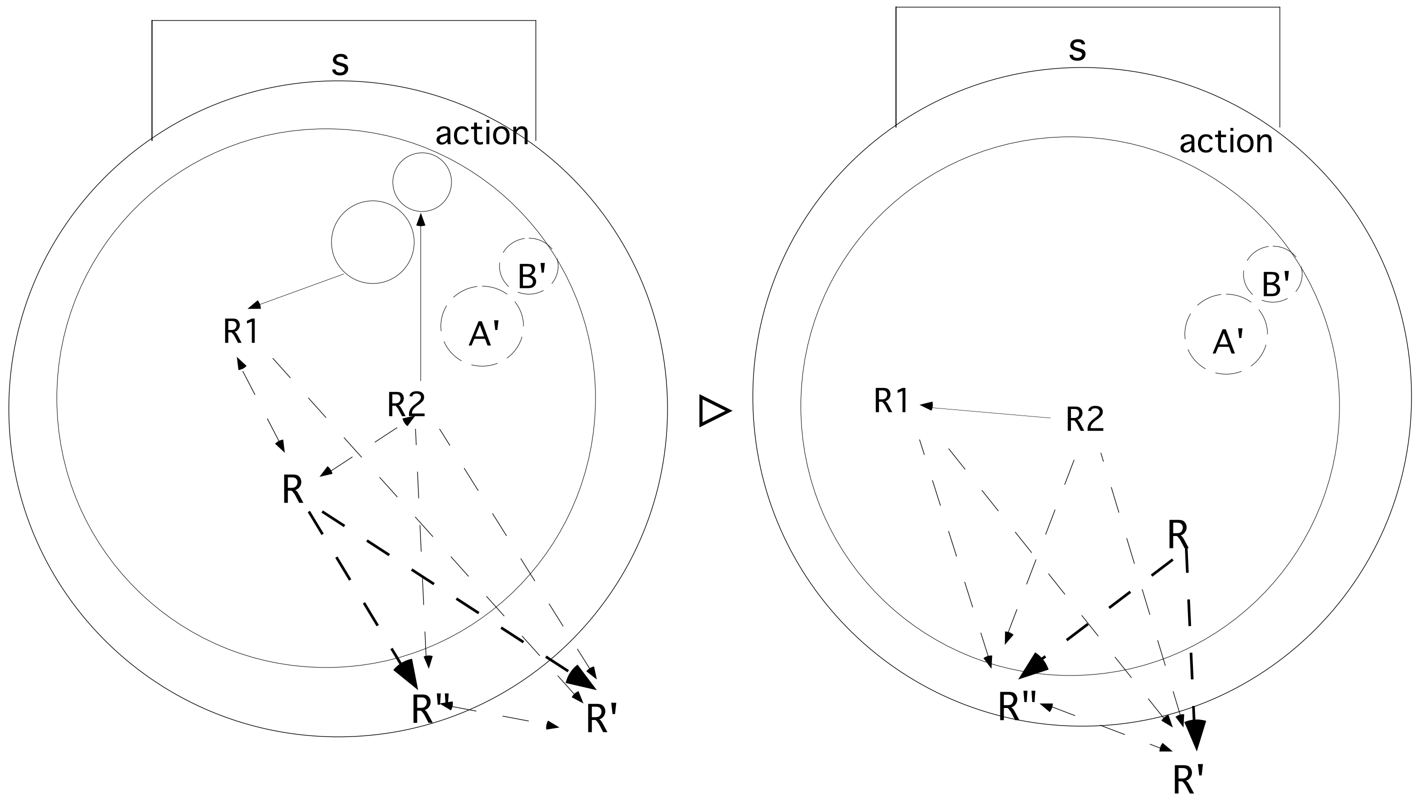


Rule of substitution (II)



where B is identifier and A is expression

First rule of parameterization



Second rule of parameterization