Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-94-5

1994-01-01

BoxGraph: A Two-Dimensional Visual Computation Model

Takayuki Dan Kimura and Timothy B. Brown

Traditional computation models such as Turing machines, lambda-calculus, Markov's normal algorithms, are not suitable models for visual programming languages because they are all based on one-dimensional text strings and visual programming uses two-dimensional graphic diagrams. We propose a two-dimensional computation model, called Boxgraph, that requires no text. The syntax of the model consists of nested boxes connected by arrows, and the semantics consists of dataflow and the concept of consistency. The expressive power of the model is demonstrated by constructing representations of a binary full adder, the Fibonacci function, and the GCD function. The model, with a small extension... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Kimura, Takayuki Dan and Brown, Timothy B., "BoxGraph: A Two-Dimensional Visual Computation Model" Report Number: WUCS-94-5 (1994). *All Computer Science and Engineering Research.* https://openscholarship.wustl.edu/cse_research/357

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/357

BoxGraph: A Two-Dimensional Visual Computation Model

Takayuki Dan Kimura and Timothy B. Brown

Complete Abstract:

Traditional computation models such as Turing machines, lambda-calculus, Markov's normal algorithms, are not suitable models for visual programming languages because they are all based on one-dimensional text strings and visual programming uses two-dimensional graphic diagrams. We propose a two-dimensional computation model, called Boxgraph, that requires no text. The syntax of the model consists of nested boxes connected by arrows, and the semantics consists of dataflow and the concept of consistency. The expressive power of the model is demonstrated by constructing representations of a binary full adder, the Fibonacci function, and the GCD function. The model, with a small extension to make it a practical visual programming language, has been implemented using the Overon operating system running on a SPARCstation equipped with a pen-tablet and udner the PenPoint operating system running on an EO-440 pen computer.

BoxGraph: A Two-Dimensional Visual Computation Model

Takayuki Dan Kimura and Timothy B. Brown

WUCS-94-05

January 1994

Laboratory for Pen-Based Silicon Paper Technology (PenLab)
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

BoxGraph:
A Two-Dimensional Visual Computation Model

Takayuki Dan Kimura and Timothy B. Brown

Laboratory for Pen-Based Silicon Paper Technology (PenLab)
Department of Computer Science
Washington University
St. Louis, MO 63105, USA
phone: (314) 935-6122, fax: (314) 935-7302, tdk@wucs1.wustl.edu

Abstract

Traditional computation models such as Turing machines, λ-calculus, Markov's normal algorithms, are not suitable models for visual programming languages because they are all based on one-dimensional text strings and visual programming uses two-dimensional graphic diagrams. We propose a two-dimensional computation model, called Boxgraph, that requires no text. The syntax of the model consists of nested boxes connected by arrows, and the semantics consists of dataflow and the concept of consistency. The expressive power of the model is demonstrated by constructing representations of a binary full adder, the Fibonacci function, and the GCD function. The model, with a small extension to make it a practical visual programming language, has been implemented using the Oberon operating system running on a SPARCstation equipped with a pen-tablet and under the PenPoint operating system running on an EO-440 pen computer.

1. Introduction

Traditionally computational processes are specified by one-dimensional text strings. Visual programming, in contrast, uses two-dimensional diagrams and icons for specifying such processes. For many computer users, especially those who can draw but cannot type well, visual programming is an attractive alternative to traditional computer programming. Since using a mouse to create diagrams is rather awkward, difficult, and time consuming for novices, the potential advantages of visual programming have not been fully appreciated. Recent advances in pen-based interfaces have changed the situation. The drawing and composition of a visual program has become easier and more natural [1].

A variety of visual programming languages have been proposed, developed, or commercialized [6]. Recent visual programming languages use dataflow as their semantic base and directed graphs as their syntactic base. Each node represents an operation and each arc represents a data communication path. Even though a variety of visual programming languages have been introduced, there are still many unsolved syntactic, semantic, and pragmatic research problems in the area of language design. Because no formal 2D grammar (similar to the phrase structure grammar for textual languages) has been developed, a mechanical parsing of visual

programs is not yet a reality [4]. Even though dataflow is now more widely used than control-flow as a semantic base for visual languages, there is still some question as to which semantic base is most suitable. One important pragmatic problem is finding visual abstraction mechanisms which make complex diagrams easy to understand, the so-called scaling-up problem of visual programming [8].

Historically, two-dimensional notations and languages have been used freely in mathematics and in symbolic logic. For example, Frege [3] introduced notations based on binary trees to represent structures of various judgements. Peirce's system of existential graphs [14] is a diagrammatic system for first order logic. It is similar in syntax to the model presented in this paper. More recent related work is in Harel's concept of Higraphs [5].

In this paper we construct a computational model that serves as a semantic base for visual programming languages such as Show and Tell [8]. Our model, Boxgraph, is based on transformations of configurations of nested boxes on the two-dimensional plane. Traditional computation models, such as Turing machines, λ-calculus, and Markov's normal algorithms, are not suitable for visual programming languages because they are all based on transformations of one-dimensional text strings. The notions of concatenation, deconcatenation, string matching and string substitution are fundamental components underlying these models. Extension of these concepts to two-dimensional space is not easy nor natural.

We have two goals in the construction of our model. First, we try to minimize the number of concepts in the model and yet make it computationally complete, i.e., '*make it as simple as possible, but not simpler.*' Second, we try to make the model as close to a practical visual programming language as possible. With a small extension of the Boxgraph model to include an abstraction mechanism for iteration, a simple but practical language, Simple Hyperflow (SHF), has been designed and implemented using the Oberon operating system [12] running on a SPARCstation with a pen-tablet. SHF is also a part of Hyperflow [9,10], a visual shell language for a children's workstation implemented on an EO-440 pen tablet running the PenPoint operating system [2].

We present our computation model in several stages. First, in the next section, we define a part of the model that is equivalent to Propositional Logic. Then, in Section 3, we define the Recursive Boxgraph model that is computationally complete. Section 4 defines SHF by introducing an iteration construct into the Boxgraph. In Section 5, two SHF programs, one for the Fibonacci function and the other for the GCD (Greatest Common Divisor) function, are constructed, and it is shown that Fibonacci and GCD are visually 'inverse' of each other.

2. Basic Boxgraph

The Boxgraph model of computation consists of a set of diagrams in the two-dimensional plane and a set of transformation rules. A *boxgraph* is an acyclic directed graph of hierarchically nested boxes. The primitives are boxes, arrows, and nothing else. Particularly, no text is needed.

Figure 1 and 2 show examples of boxgraphs. Figure 1 represents the logical functions AND,

OR, and XOR, where the logical values, True (T) and False (F), are represented by a double box without arrows and a single box without arrows, respectively. Figure 2(a) shows the boxgraphs before and after the computation of F = AND(F,T), and Figure 2(b) shows the same for T = AND(T,T). During the computation a box may become *inconsistent*, when two different values try to occupy the same box causing a conflict. This is shown by shading the inconsistent box. When a box becomes inconsistent, all the arrows incident with the box become non-existent for the remainder of the computation. A box can be either *open* or *closed* in terms of limiting the scope of inconsistency propagation. An open box, framed by dotted lines as in Figure 1(c), allows the inconsistency flow out to its surrounding environment, i.e, when an open box becomes inconsistent, the smallest box containing it also becomes inconsistent. A closed box framed by solid lines, does not propagate inconsistency to its environment. The concept of inconsistency will be defined in more exact terms later.

A box partitions the 2D space into two parts, the interior and the exterior of the box. For example, consider the boxgraph of Figure 3(a). The box A divides the boxgraph into its interior Figure 3(b), and its exterior, Figure 3(c). The interior defines the functionality of the box and the exterior defines the usage of the box. A different usage of the same box **A** is shown in Figure 3(d). The boxgraph of Figure 3(e) represents the same computation as Figure 3(a) while the graphic layouts are different.

The location, the size, and the shape of a box is insignificant. Indeed a box can be replaced by any simple closed curve. For example, Figure 3(f) is isomorphic to Figure 3(a). Even though our implementations of the model allow boxes only, we use both boxes and other shapes in this paper, as later in Figure 6(a). Similarly, the size and the shape of an arrow is insignificant. The location of an arrow, however, is significant in terms of which boxes the arrow intersects. Figures 4(a) and 4(b) are two different boxgraphs.

No two boxes may intersect with each other, but an arrow may intersect with other arrows and with boxes. Two boxes of different hierarchical levels can be connected by an arrow that crosses layers of box boundaries, as long as no cycle is formed among the boxes of the same hierarchical level. The graph in Figure 5(a) is not a boxgraph because boxes **A** and **B** are on the same hierarchical level and form a cycle with the line segments **b** and **d**. On the other hand, the boxgraph of Figure 3(a) is acyclic because the four boxes of Figure 3(c) are acyclic and the interior three boxes in the box **A** of Figure 3(b) are also acyclic. Note that if we were to consider box **A** to be at the same hierarchical level as the three interior boxes in Figure 3(b), then the four boxes form a cycle by a path from **A** to itself through the three boxes. In order to avoid such unwanted cycles, we consider, by definition, the arrows connecting a box to its interior boxes to be at the the same level as the exterior boxes, therefore they do not count when testing the interior boxes for cycles.

A boxgraph containing no arrow is called *trivial* and represents a *value* such as T and F in Figure 1. A box may contain either nothing, a trivial boxgraph, or a non-trivial boxgraph, but not both. Figure 5(b) illustrates a non-boxgraph which violates this rule. A box containing a non-trivial boxgraph, such as box **A** in Figure 3(b), is called an *operation* box. Note that a box containing only arrows, as in Figure 5(c), is also an operation box. An empty box or a box

containing a trivial boxgraph is called a *memory* box. In our intended semantics, an operation box receives input values from its exterior and returns output values to the exterior. Arrows which arrive at the box boundary from the exterior are referred to as *in-arrows*. Arrows which start at the box boundary are referred to as *out-arrows*. Establishing the association between the in-arrows and the out-arrows is analogous to parameter binding in traditional programming. The boxgraph model uses the following *positional* binding rule:

> Rank all the in-arrows from the exterior by lexicographical ordering of the (x,y) coordinates of their intersection points with the box. Then do the same for all the out-arrows to the interior. Bind an in-arrow with an out-arrow of the same rank.

The same binding rule applies to in-arrows from the interior and out-arrows to the exterior. For example, in Figure 3(e), arrows **a** and **b** are bound with arrows **c** and **d**, respectively, and arrow **e** is bound with arrow **f**. An operation box defines an environment for the computations carried out by its interior boxes and controls propagation of consistency to its outer environment. In that sense it defines a two-dimensional block structure.

An informal semantics of the boxgraph model can be presented in an imperative (prescriptive) form as follows: A trivial boxgraph represents a *datum* or a *value*. A non-trivial boxgraph represents an *operation*. An arrow represents a data communication path, i.e., dataflow. A value *flows* from the starting box of an arrow to the ending box. The data transfer may be carried out anytime asynchronously. Since no cycle exists, once a box is filled by a value, the value stays there. A memory box becomes *inconsistent* if the incoming value is different from the value already existing in the box, i.e., when a conflict occurs in the box, it becomes inconsistent. When a box becomes inconsistent, all arrows intersecting with the box become ineffective, i.e., no data may flow on those arrows. If an open memory box becomes inconsistent, the smallest operation box containing the open box also becomes inconsistent. If the operation box is also open, then the smallest box containing the open operation box becomes inconsistent, and so on. A computation halts when all empty boxes are filled with values. If a boxgraph has an empty box with no in-arrows, then it is called *free*, otherwise it is called *bound*. There exists no computation for a free boxgraph.

In order to make the above semantics more precise, we now present them in a declarative (descriptive) form. In the declarative semantics, a boxgraph defines a logical constraint among the box contents. A boxgraph is consistent if the content of each box satisfies the constraints imposed by the neighboring box contents. A computation is defined as a process to find the set of values for filling in the empty boxes without violating the consistency of the boxgraph. Formally a boxgraph is *consistent* if there exists an elaboration, otherwise it is *inconsistent*, where an *elaboration* of a boxgraph is an assignment of values, including the null value ($\perp$), to all the arrows such that the constraints imposed by each box as defined below may be satisfied. The concept of elaboration is similar to the concept of interpretation in mathematical logic, where a set of propositions are defined to be consistent if there exists a model (interpretation) for which all propositions are satisfied. As in logic, the consistency of a boxgraph depends on the existence of

an elaboration and not on how the elaboration is constructed. Local constraints imposed by a box on the values of the incident arrows are defined as follows (Figure 6):

Let $A = \{a_1, a_2, \ldots, a_m\}$ be the set of values on the in-arrows to a box, and

$B = \{b_1, b_2, \ldots, b_n\}$ be the set of values on the out-arrows from a box.

(1) When the box is a memory box with c ($\perp$, if it is empty) as its content:

(1.1) If $A \cup \{c\}$ has no non-null value, then $B = \{\perp\}$.

(1.2) If $A \cup \{c\}$ has exactly one non-null value a, then $B = \{a\}$.

(1.3) If $A \cup \{c\}$ has more than one non-null value and the box is closed, then $B = \{\perp\}$.

(2) When the box is an operation box with β as its content:

(2.1) β bound with A and B is consistent.

(2.2) If the box is closed and β bound with A and B is inconsistent, then $B = \{\perp\}$.

If a boxgraph is consistent and an elaboration is found, then the computation of filling the empty boxes can be carried out by transferring the values on the in-arrows into the empty boxes. As an example, for computing T=XOR(F,T), using the boxgraph of Figure 1(c), an elaboration is given in Figure 7. The result of a computation is unique if the elaboration is unique. Since all boxgraph are acyclic, using structural induction on the partial ordering on the boxes, we can prove that if a finite boxgraph is bound and consistent, then its elaboration is unique. A proof is similar to those given in [7] for demonstrating the determinancy of the Show and Tell visual programming language.

In order to demonstrate the power of the boxgraph model presented so far, a binary full adder is constructed in Figure 8. With x, y, and c as a Boolean input, the Boolean output values, s and c', are computed by: $s = x \oplus y \oplus c$ and $c' = xy+yc+xc$. In Figure 8(a), a black dot represents an empty memory box. The symmetries of s and c' are illustrated in Figure 8(b) by separate constructions of circuits for s and for c'.

3. Recursive Boxgraph

The computational power of the model presented in the previous section is limited to that of Boolean circuits or that of Propositional Logic. In this section we introduce naming, numbers, the successor operation, and the predecessor operation, into the boxgraph model. We call the extended model a *recursive boxgraph.*

A *name* of a boxgraph is any icon (bitmap) of any size. Any rendering of a text can be used as a proper name of a boxgraph. A declaration of a name and its denotation is represented by a meta-expression connecting the name icon and a boxgraph by an '=' sign. A recursive definition of a boxgraph is allowed; thus a box may contain either nothing, a trivial boxgraph, a non-trivial boxgraph, or a name of a boxgraph. In Figure 1, 'T' and 'F' are used as the names of two trivial boxgraphs. When a box contains the name of an operation box, we equate the frame of the named operation box with that of the box containing the name. For example, if we name the operation box of Figure 3(a) by '$\wedge$' as in Figure 9(a), then Figure 3(b) and 3(c) can be represented by Figure 9(b) and Figure 9(c), respectively. We also use an boxed icon as a name for an operation box.

In the recursive boxgraph model, we represent the set of natural numbers by a tally system

where the number zero is represented by an empty box and the successor of number **n** is represented by a box containing the representation of number **n**. There are two primitive number operation boxes; the successor box and the predecessor box, named by '⇑' and '⇓', respectively. The semantics of the primitives are illustrated in Figure 10. Note that the predecessor box is inconsistent if its input is zero.

The standard set of arithmetic operations are defined in Figure 11 (a) through (g). Recursive definitions of the Fibonacci function and the GCD function are given in Figure 12(a) and (b).

4. Simple Hyperflow

The recursive boxgraph is capable of representing any computable functions, i.e., the model is computationally complete. It also can be used as a visual programming language. However, it is not necessarily a practical visual language, because it has limited abstraction mechanisms for complexity management, i.e., only naming. Naming simplifies representation of hierarchically nested boxgraphs. It provides the capability of *vertical abstraction*. In two dimensional languages there exists another type of abstraction, called *horizontal abstraction*, which was first introduced in the Show and Tell Language [8] as an *iteration* box. We introduce the iteration box into the recursive boxgraph and we call the resulting model a Simple Hyperflow (SHF), because it serves as the semantic foundation of a practical visual programming language such as Hyperflow [9,10].

The iteration box is denoted by an operation box with a thick frame as in Figure 13(a). It represents an abstraction of folding a horizontally spreading boxgraph of Figure 13(b) into one place. Small circles indicate the connecting points of horizontal concatenation of operation boxes. When an iteration box is executed, it is unfolded dynamically until the operation box becomes inconsistent.

A usage of the iteration box and the power of visual abstraction can be demonstrated by constructing an iterative definition of the Fibonacci function, which is much simpler than the recursive definition given in Figure 12(a). Figure 14(a) visually represents the one essential step of computing the Fibonacci sequence. Figure 14(b) gives a boxgraph for computing a segment of Fibonacci sequence, which is a horizontal iteration of an identical boxgraph, enclosed in the dotted gray box. By folding the sequence into a single box, Figure 14(c) represents an unbounded boxgraph for computing the Fibonacci sequence. The execution of Figure 14(c) will not terminate since the content of the iteration box never becomes inconsistent. The boxgraph of Figure 14(d) introduces a condition that causes inconsistency. It computes the largest Fibonacci number less than 1000.

5. GCD = Fibonacci^{-1}

In this section we demonstrate a merit of visual programming. We construct a visual program for the GCD function as a 'visual inverse' of a visual program for the Fibonacci function.

Corresponding to Figure 14(a), the one essential step for computing the GCD is represented by Figure 15(a), where the '-' operation is the symmetric difference defined in Figure 11(c).

Using the same abstraction mechanism as for Fibonacci, a boxgraph of Figure 15(b) can be constructed for GCD. It is an 'inverse' of Figure 14(c) in the sense that the arrow directions are reversed, the vertical position is inverted, and the addition operation is inverted into the symmetric difference operation. The termination condition is also inverted from the upper bound testing in Figure 14(d) to the lower bound testing in Figure 15(c).

The relationship between the GCD and the Fibonacci functions is indicated by the following representations using Guarded Commands:

FIB = $x := 0; y := 1;$
do $x \geq y \Rightarrow y := y + x$ ||
$y > x \Rightarrow x := x + y$ od;
GCD = $x := a; y := b;$
do $x < y \Rightarrow y := y - x$ ||
$y \leq x \Rightarrow x := x - y$ od;

However, it is easier to see the exact relationship between the two programs in the visual form than in textual form.

6. Oberon Implementation

SHF has been implemented on a SPARCstation IPX running the SPARC-Oberon System created by Templ [15]. SPARC-Oberon is an implementation of both the Oberon System [21] and the programming language Oberon [20] for SPARC processors. The SPARCstation is equipped with a WACOM HD-648A pen tablet [18] which is used as the interface for SHF. The tablet sends pen position information to the system via a serial port connection. The display portion of the tablet is controlled by a Vigra VS10 video adapter card [17] which has been added to the SPARCstation. Integrating the pen tablet into the SPARC-Oberon environment involved creating two small shared object libraries of code written in C, one library for retrieving input from the tablet's digitizer and another for displaying output on the tablet's screen. These libraries were then opened and used by Oberon modules designed to allow easy access to the tablet. One module was written for each of the two libraries. The approach taken in the design of these modules was to write only enough code in C to allow access to the tablet. The significant drawing algorithms were written in Oberon.

The current Oberon implementation of SHF includes a simple graphics editor which incorporates a shape recognition algorithm [1]. The shape recognizer converts user inputs into the two primitives of the Boxgraph model, recognizer converts user inputs into the two primitives of the Boxgraph model, boxes and arrows. The editor allows the user to create, select, and delete groups of and select groups of boxes and arrows. In addition, the user can save SHF programs to an ASCII text file which can be read back into the system later. Provisions for dumping the screen to a TIFF [16] file have also been added.

Also included is a reduction system which is used to execute programs written using the system. In addition to the the simple box type, predecessor, successor, and iteration boxes have been implemented. Figure 16(a) shows a screen dump of a binary full adder program written using the system. Figure 16(b) shows the result of executing the program after supplying

appropriate inputs. Figure 17 shows the Fibonacci function. Iteration boxes are shown as boxes with a thick frame. Successor boxes and the predecessor boxes are shown containing an 's' and a 'p' respectively. Since all inputs and outputs of the iteration boxes are connecting points for horizontal concatenation, the small circles, used to differentiate these connecting points from other points of entry into an iteration box in Figure 14, are not used in Figure 17.

The system was created using an object-oriented design. The implementation of this design was made easier by the use of the Oberon-2 [11] extension to the Oberon programming language, particularly type-bound procedures. Using type extension, the following hierarchy of types was designed. Indentation is used to show the type/extension relationship; comments are shown, in Oberon style, enclosed in "(*" and "*)".

```
Object
 LocatedObject
  Option                  (* Used to implement the option menu *)
  Shape
   Line
    Arrow
   Rectangle
    HFBox                 (* Hyperflow Box *)
     HFPredBox            (* Predecessor Box *)
     HFSuccBox            (* Successor Box *)
     HFIterBox            (* Iteration Box *)
 SortedListNode           (* Abstract List Type *)
  ArrowListNode           (* Instantiation of Abstract List Type *)
  OptionListNode
  TreeNode                (* Abstract Tree Type *)
   HFBoxTreeNode          (* Instantiation of Abstract Tree Type for Hyperflow Boxes *)
   HFIterBoxTreeNode      (* Iteration Box Tree *)
```

The arrows are maintained by the system as a sorted list; the boxes are maintained in a tree in which the parent-child relationship corresponds to box containment.

Future plans for this system include the adding of a character recognition algorithm to allow the user to input text-based data. Using the character recognition system, more extensive box types could be designed allowing the user to enter Oberon programs which define the internal workings of the box. Additional editor functions such as moving and resizing shapes, grouping and ungrouping shapes, and adding patterns to the frames of boxes and the lines of arrows are also planned.

7. Conclusions

It has been argued that communicative organizations utilize two types of communication

modes, 'telephone' (point-to-point) type and 'to-whom-it-may-concern' (broadcasting) type [19]. The boxgraph model of computation has both, message passing by dataflow and broadcasting by inconsistency propagation. Hyperflow, as a practical visual language, has another communication mode called *posting* where the sender does not know who receives the posted message but the receiver knows the location of the posted message. The concept of variable (memory cell) in the traditional programming languages embodies the posting capability. Incorporation of posting into the boxgraph model will be a future work.

Another extension of the model will be in the area of continuous data types. As multimedia applications become a more significant genre of computer usage, processing of continuous data types, such as audio and video signals, becomes more necessary. Traditional message passing communication models are not suitable for that purpose. We believe that dataflow-based communication/computation models are better suited to deal with continuous data types. How can the boxgraph model be extended to include continuous data types?

The concept of elaboration is also applicable to bidirectional constraint satisfaction problems. For example, even though the boxgraph of Figure 18(a) cannot be solved simply by dataflow execution because it requires the equation solving capabilities, there exists an elaboration to satisfy the constraints, Figure 18(b). How to extend the boxgraph to a more general constraint satisfaction model is another future work we intend to pursue.

8. References

[1] Apte, A., Vo, V., and Kimura, T.D. "Recognizing Multistroke Geometric Shapes: An Experimental Evaluation," *Proceedings of UIST93*, Atlanta, November 1993.

[2] Carr, R. and Shafer, D. *The Power of PenPoint*. Addison-Wesley, 1991.

[3] Frege, G. "Begriffsschrift, a formal language, modeled upon that of arithmetic, for pure thought, 1879," in *From Frege to Goedel*, Harvard Press, 1967.

[4] Golin, E. J. "Parsing Visual Languages with Picture Layout Grammars," *Journal of Visual Languages and Computing*, 2:4, December 1991, pp. 371 - 393.

[5] Harel, D. "On Visual Formalisms," *CACM* 31:5, May 1988, pp. 514 - 530.

[6] Hils, D. D. "Visual Languages and Computing Survey: Data Flow Visual Programming Languages," *Journal of Visual Languages and Computing* 3:1 (1992), pp 69-101.[4]

[7] Kimura, T.D. "Determinancy of Hierarchical Dataflow Model," Technical Report WUCS-86-5, Department of Computer Science, Washington University, St. Louis, MO., March 1986.

[8] Kimura, T.D., Choi, J.W. and Mack, J.M. "Show and Tell: A Visual Programming Language," Invited paper in *Visual Computing Environments*, E.P. Glinert(ed.), IEEE Computer Society Press Tutorial, Washington, D.C., 1990, pp. 397-404.

[9] Kimura, T.D. "Hyperflow: A Visual Programming Language for Pen Computers," *Proceedings of 1992 IEEE Workshop on Visual Languages*, Seattle, September 1992, pp. 125-132.

[10] Kimura, T.D. "Hyperflow: A Uniform Visual Language for Different Levels of Programming," *Proceedings of 21st ACM Annual Computer Science Conference(CSC'93)*, Indianapolis, February 1993, pp. 209 - 214.

[11] Moessenboeck, H. and Wirth, N. "The programming language Oberon-2," *Structured Programming*, 12, 1991, pp. 179 - 195.

[12] Reiser, M. *The Oberon System: User Guide and Programmer's Manual*, Addison-Wesley, Wokingham, England, 1991

[13] Reiser, M. and Wirth, N. *Programming in Oberon: Steps Beyond Pascal and Modula,* Addison-Wesley, Wokingham, England, 1992.

[14] Roberts, D.D. *The Existential Graphs of Charles S. Peirce,* Mouton, The Hague, 1973.

[15] Templ, J. *SPARC-Oberon User Guide,* ETH Zurich, 1992.

[16] *TIFF: Revision 6.0 Specification,* Aldus Corporation, Seattle, WA, 1992.

[17] *VS10 and VS12 Color Frame Buffers for SPARCstation Computers: Installation Guide,* Vigra, Inc., San Diego, CA, 1991.

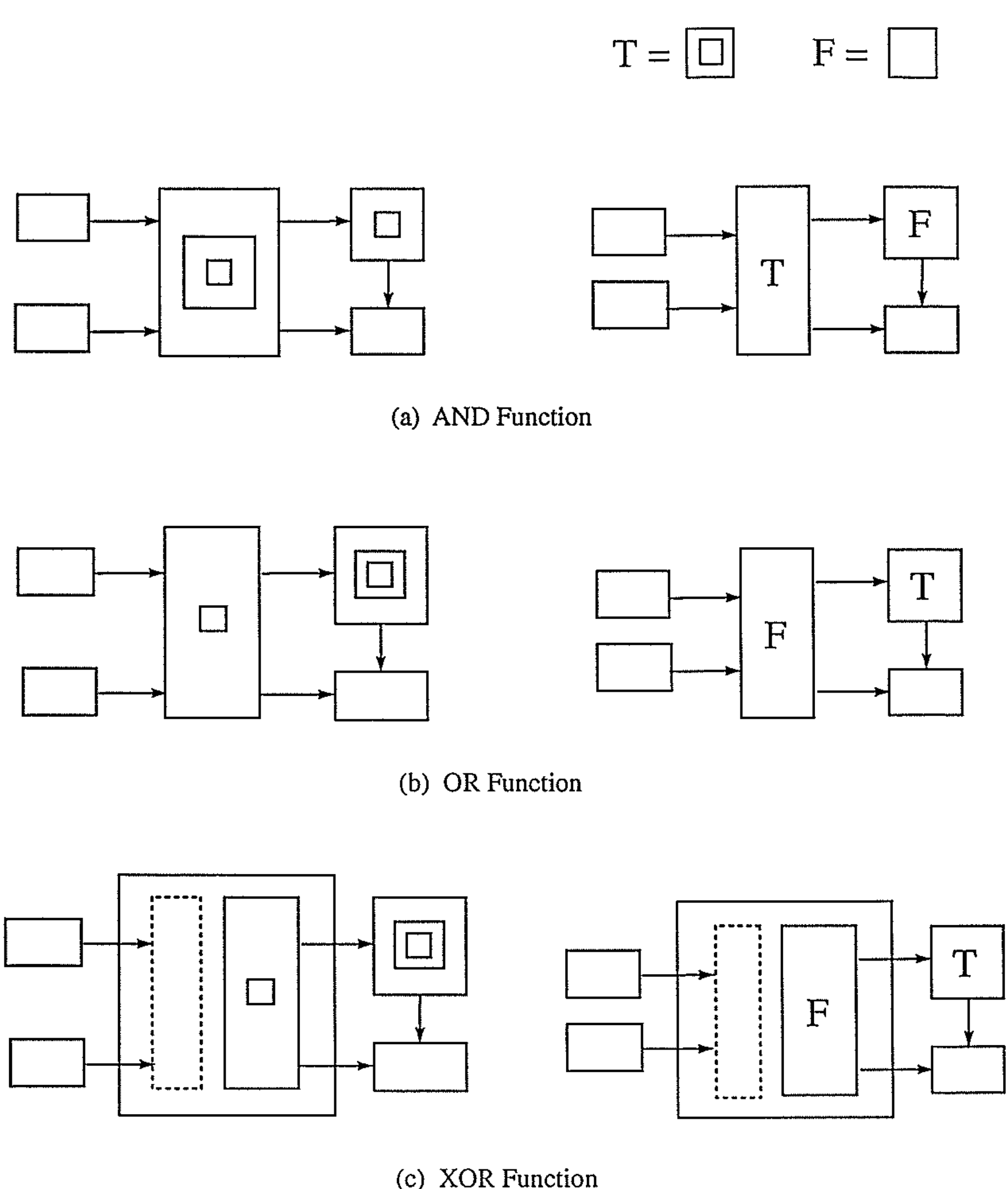
[18] *WACOM HD-648A LCD Integrated Tablet: Operation Handbook and Programmer's Manual,* WACOM Co., Ltd., Paramus, NJ, January 1991.

[19] Wiener, N. *The Human Usage of Human Beings*, Avon Books, New York, 1950.

[20] Wirth, N. "The programming language Oberon," *Software--Practice and Experience,* 18:7, July 1988.

[21] Wirth, N. "The Oberon System," *Software--Practice and Experience,* 19:9, September 1989.

[22] Wirth, N. and Guetknecht, J. *Project Oberon: The Design of an Operating System and Compiler,* Addison-Wesley, Wokingham, England, 1992.

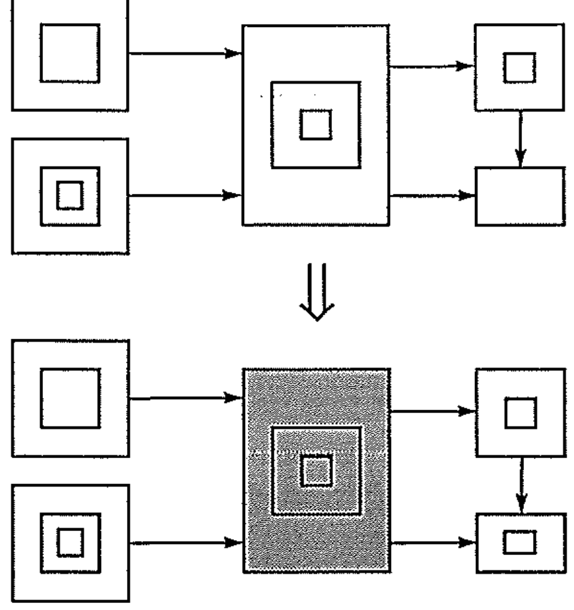


(a) AND Function

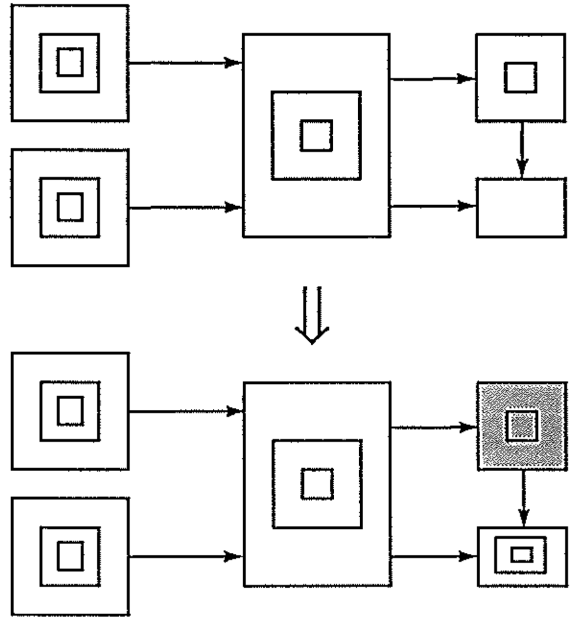
(b) OR Function

(c) XOR Function

Figure 1: Logical Functions in Boxgraph

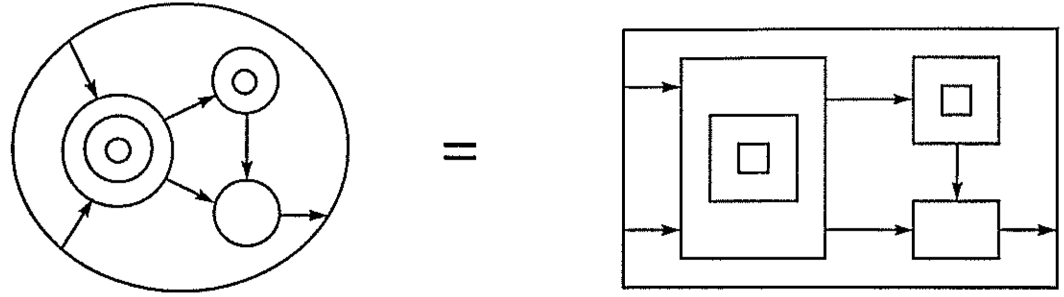


(a) Computation of F = AND(F,T)

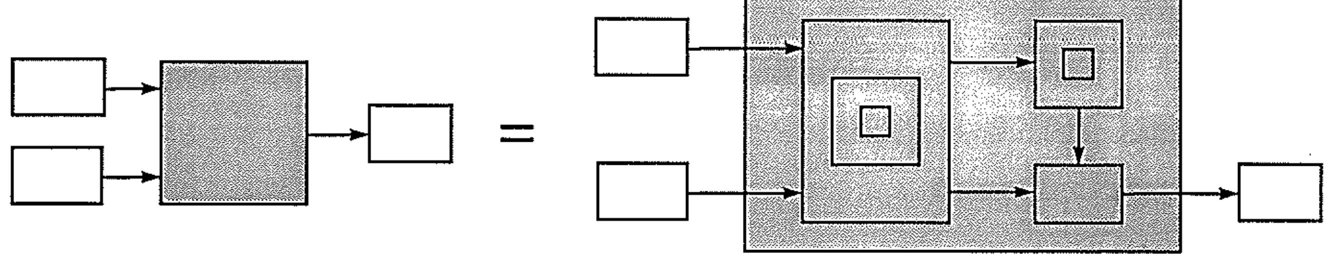


(b) Computation of T = AND(T,T)

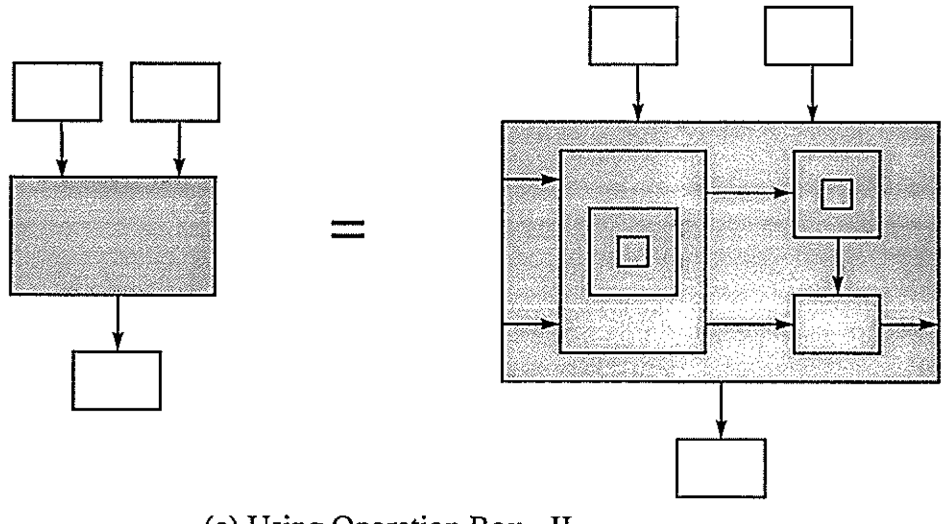
Figure 2: Computations of AND



(a) Operation Box



(b) Using Operation Box - I



(c) Using Operation Box - II

Figure 3: AND Operation Box

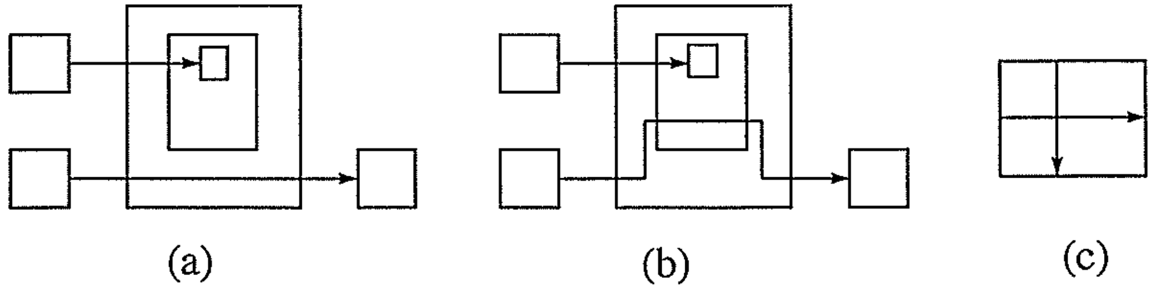


Figure 4: Positions of Arrows

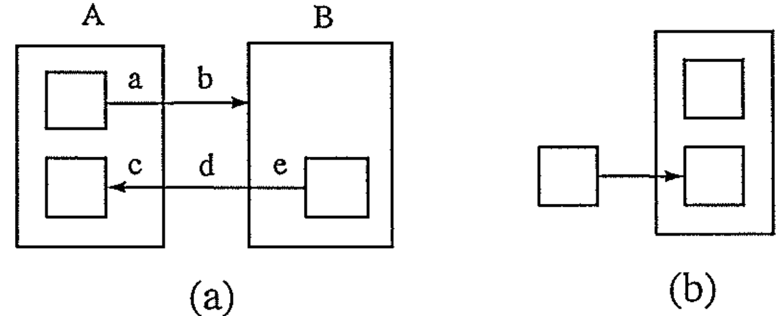
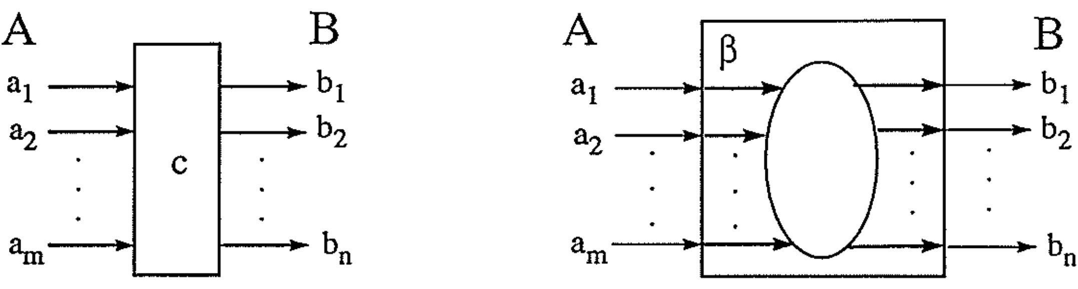
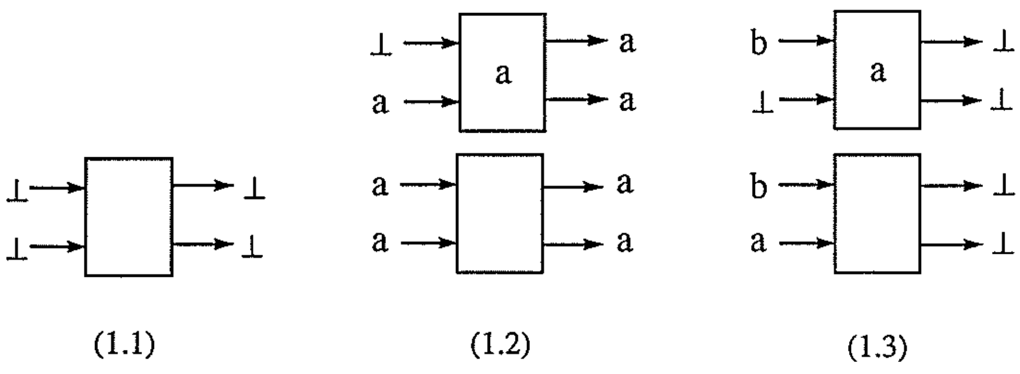


Figure 5: Non-Boxgraphs

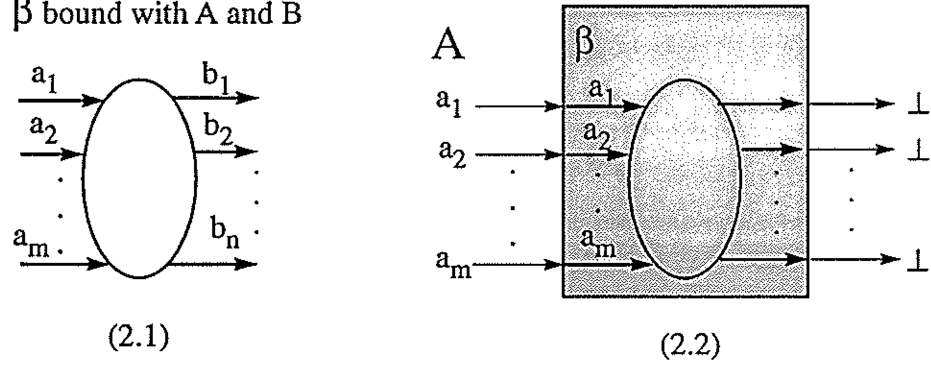


(a) Box Elaboration



(1.1) (1.2) (1.3)

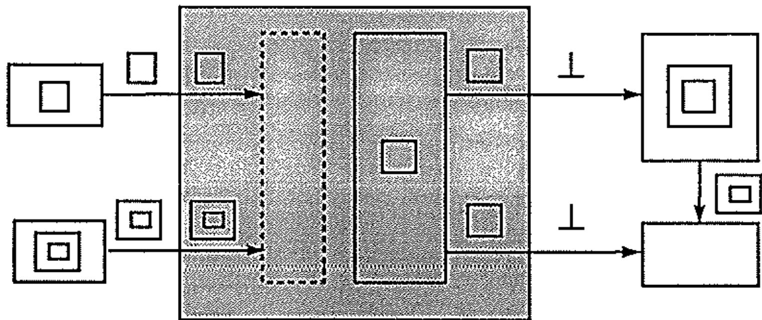
(b) Memory Box Constraints



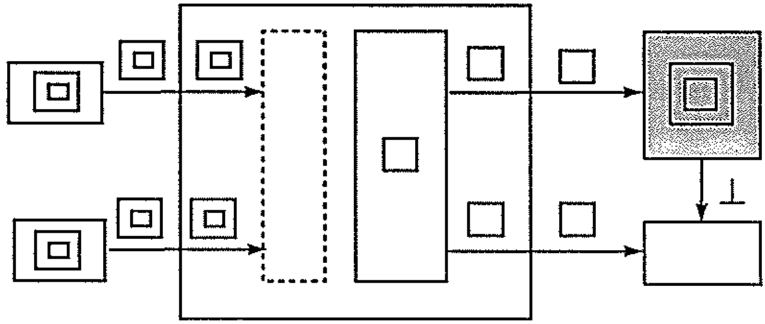
(2.1) (2.2)

(c) Operation Box Constraints

Figure 6: Local Constraints



(a) Elaboration of T = XOR(F,T)



(b) Elaboration of F = XOR(T,T)

Figure 7: Elaborations of XOR Function

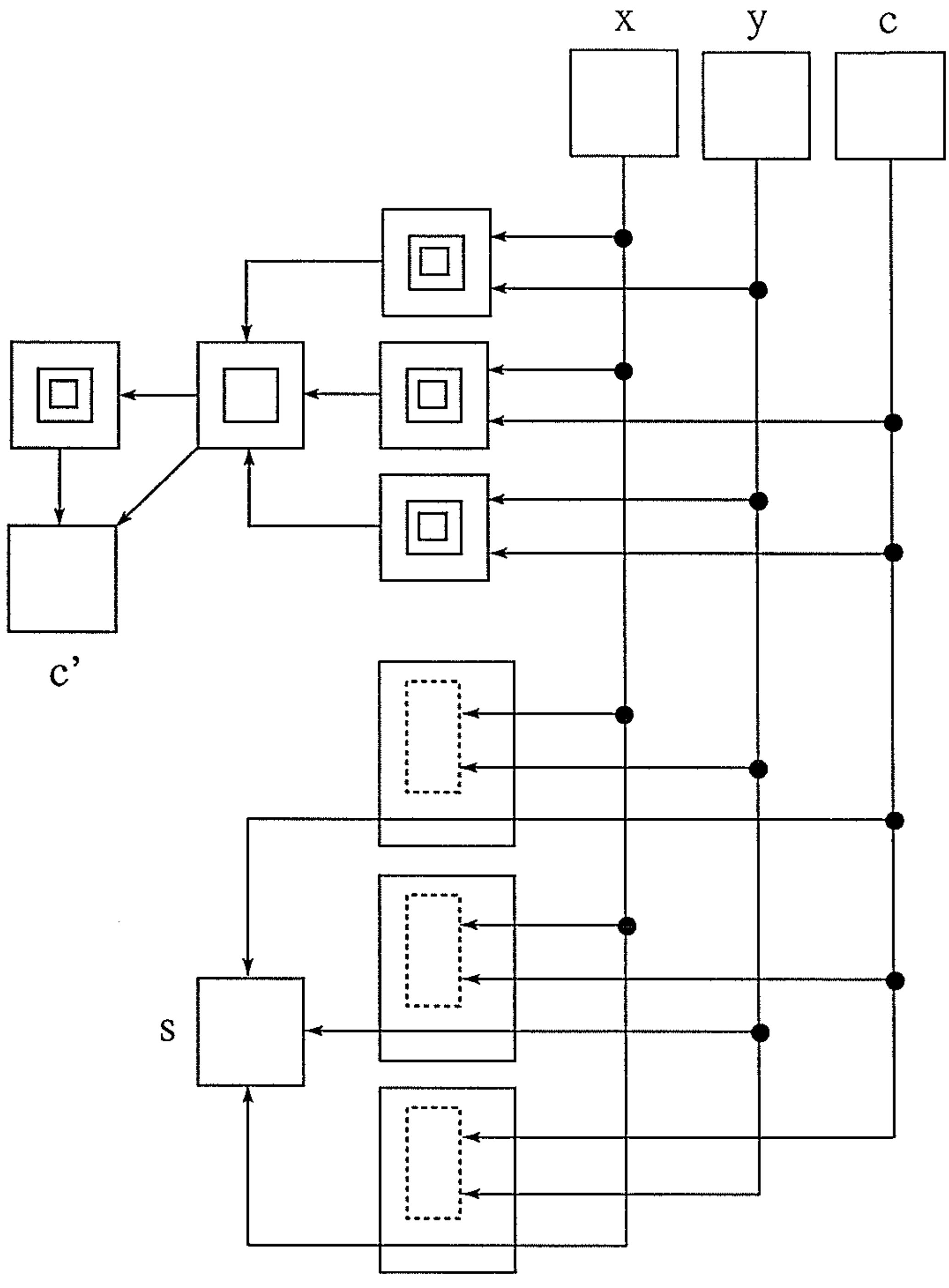
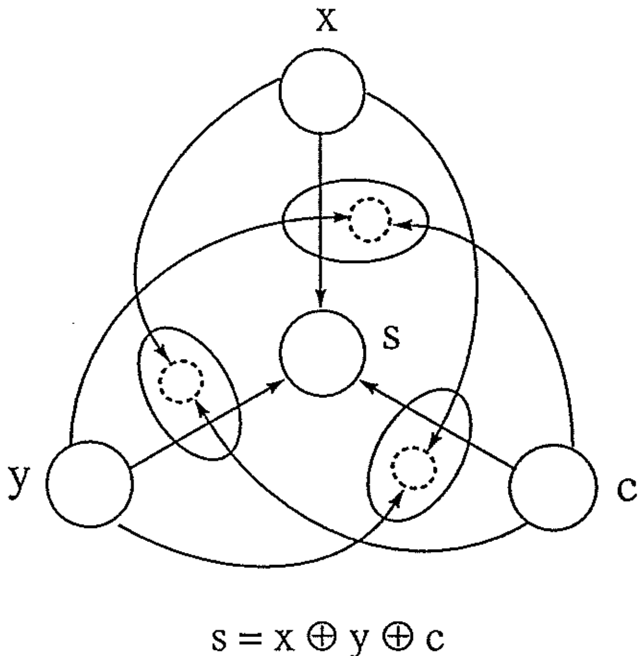
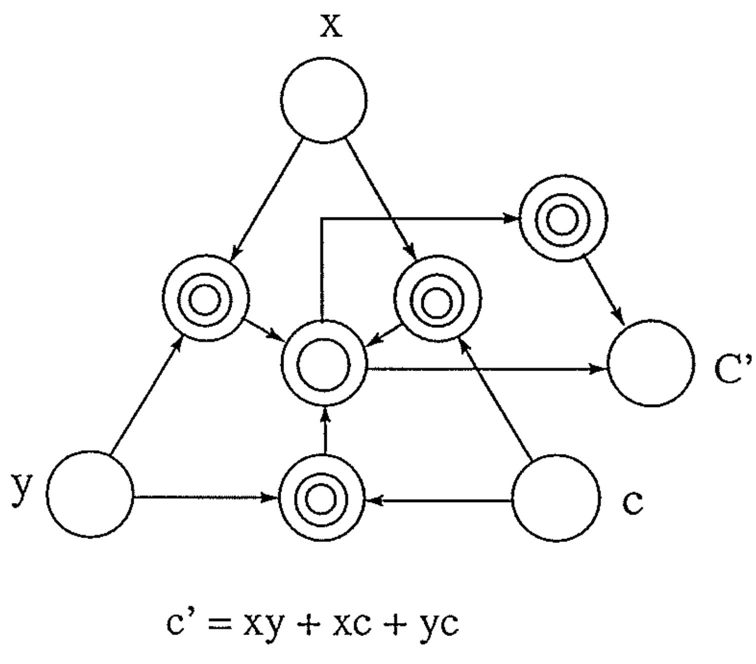


Figure 8(a): Boxgraph Representation of 1-bit Full Adder

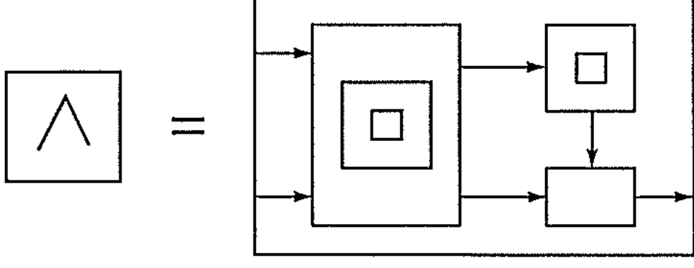


$s = x \oplus y \oplus c$

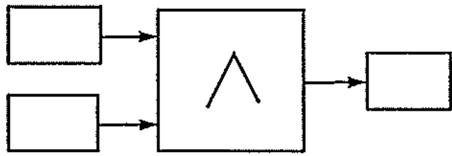


$c' = xy + xc + yc$

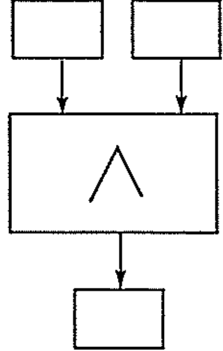
Figure 8(b): Symmetric Representation of 1-bit Full Adder



(a) Name Declaration

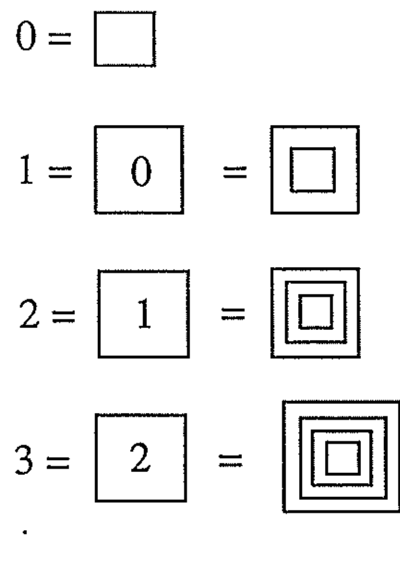


(b) Using the Named Operation Box (Figure 3(b))

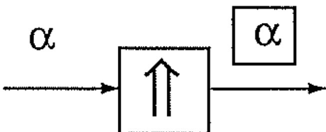


(c) Using the Named Operation Box (Figure 3(c))

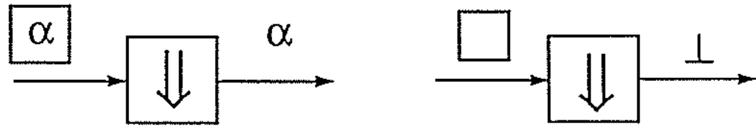
Figure 9: Naming Boxgraph



(a) Number Definition



(b) Successor Function



(c) Predecessor Function

Figure 10: Arithmetic Primitives

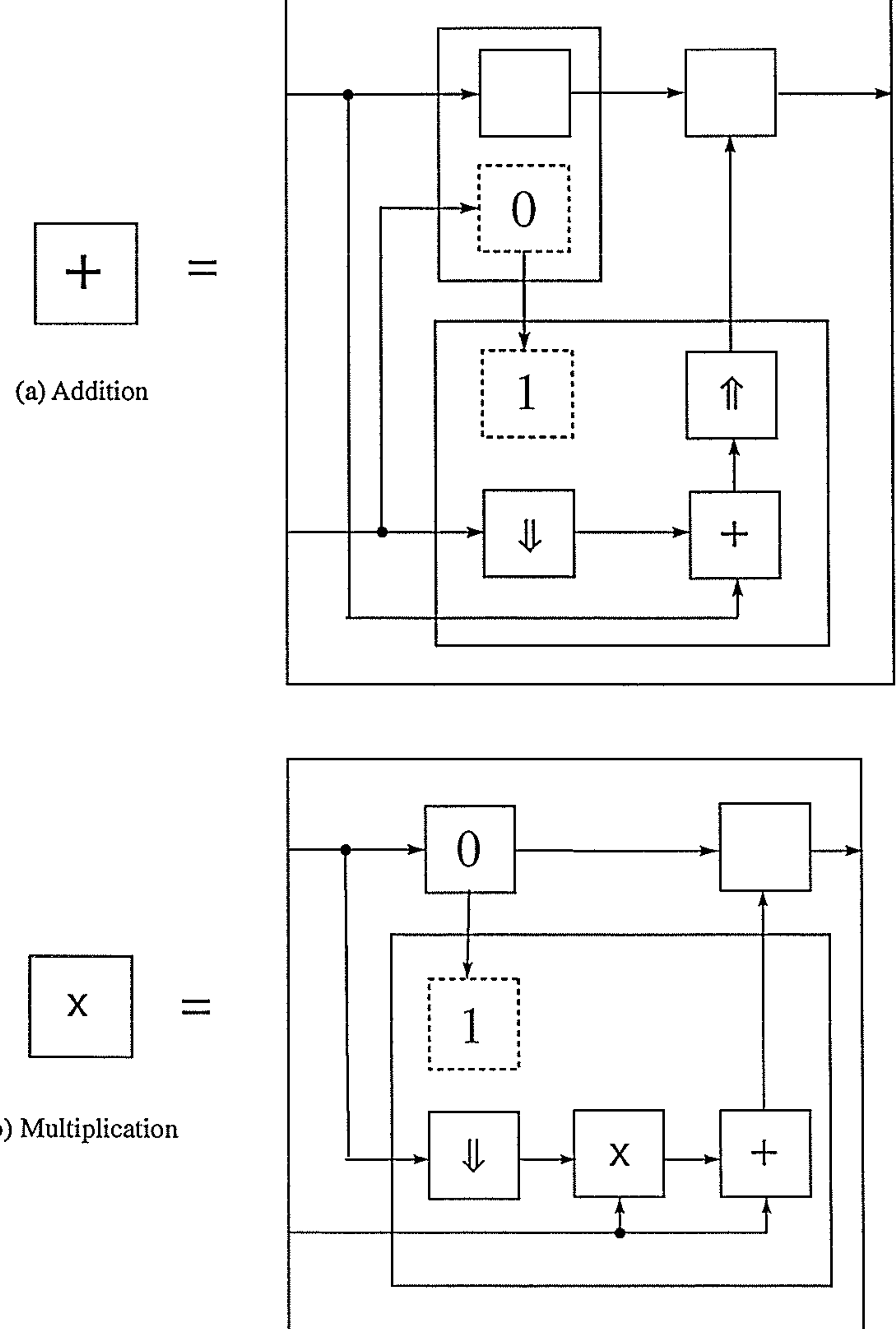
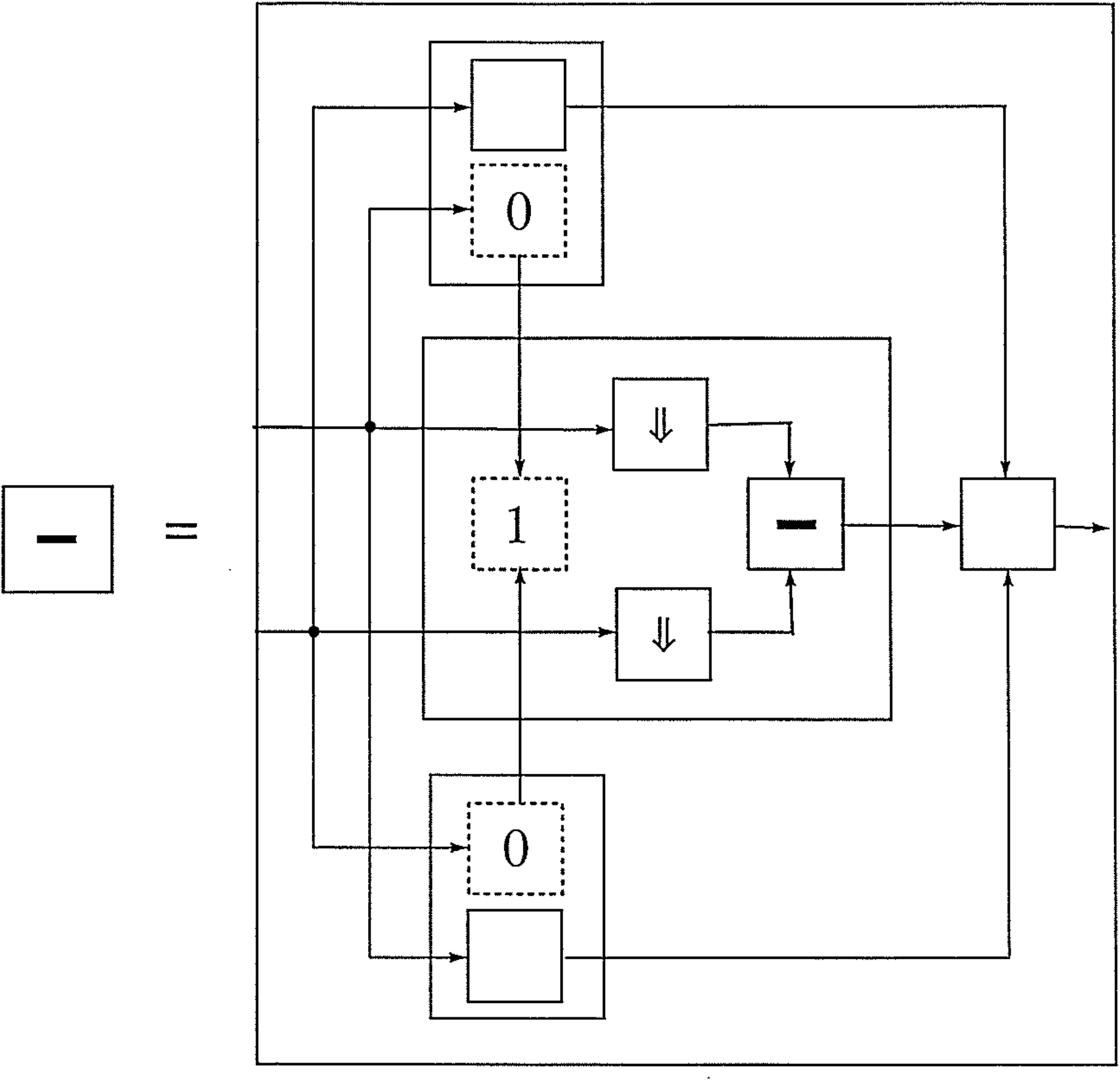
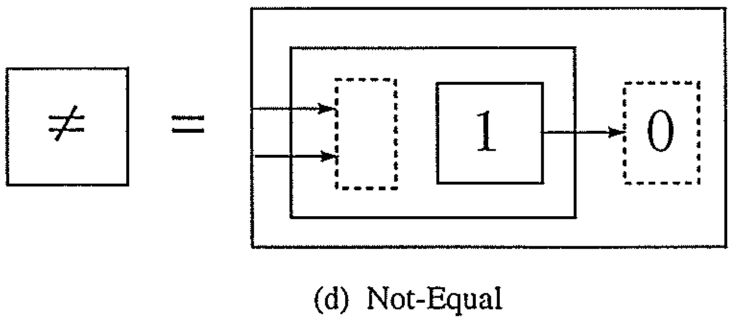


Figure 11: Arithmetic Operations

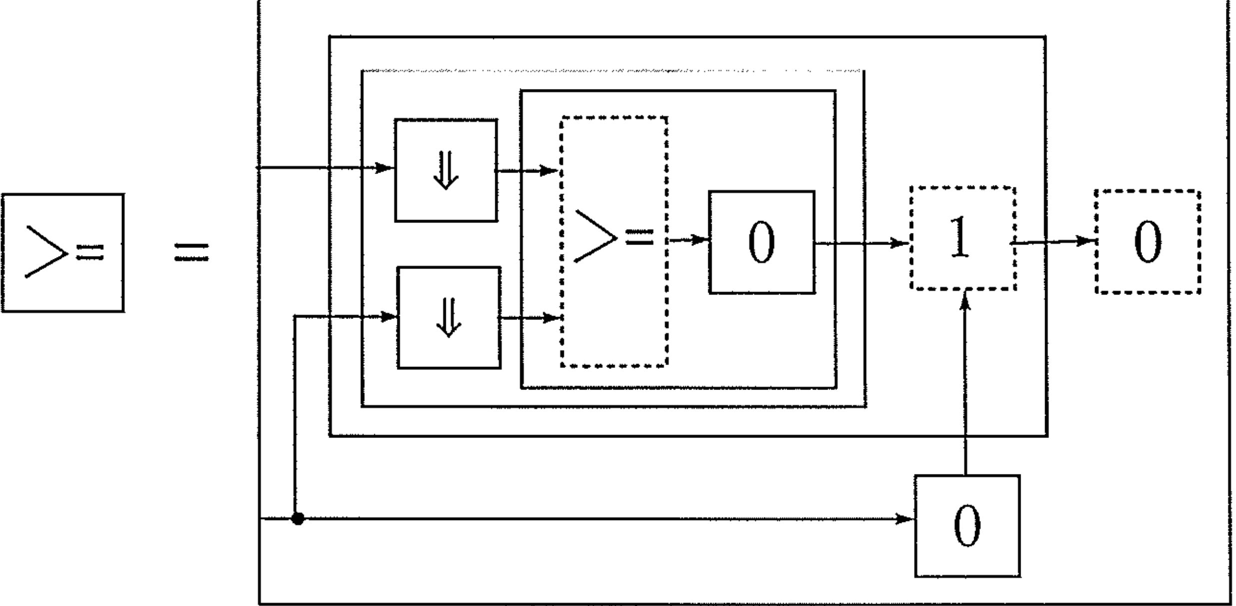


(c) Symmetric Difference

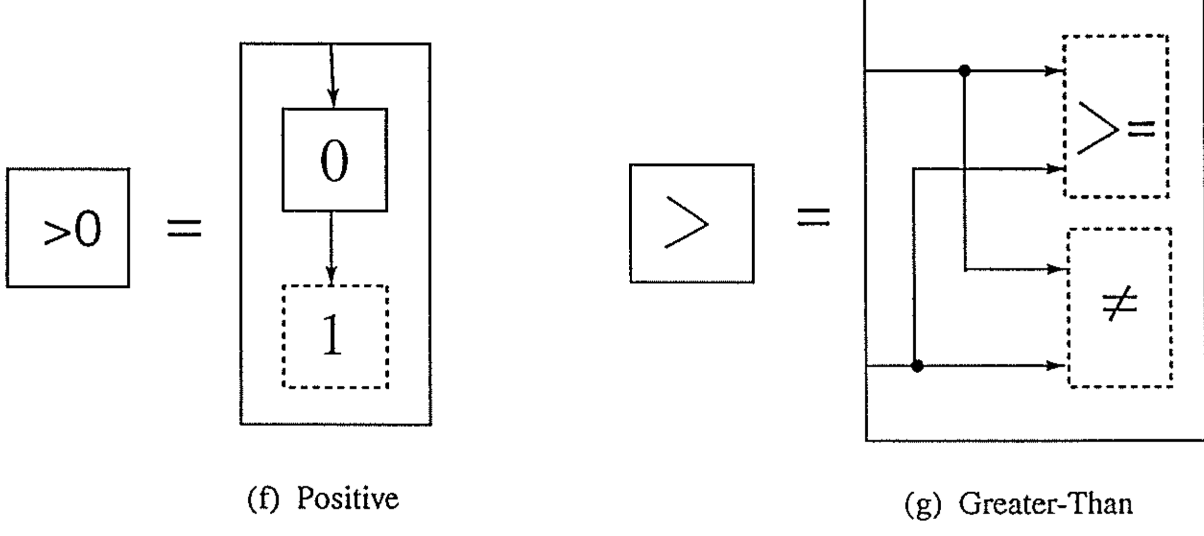
Figure 11: Arithmetic Operations



(d) Not-Equal



(e) Greater-Than-or-Equal



(f) Positive

(g) Greater-Than

Figure 11: Arithmetic Operations

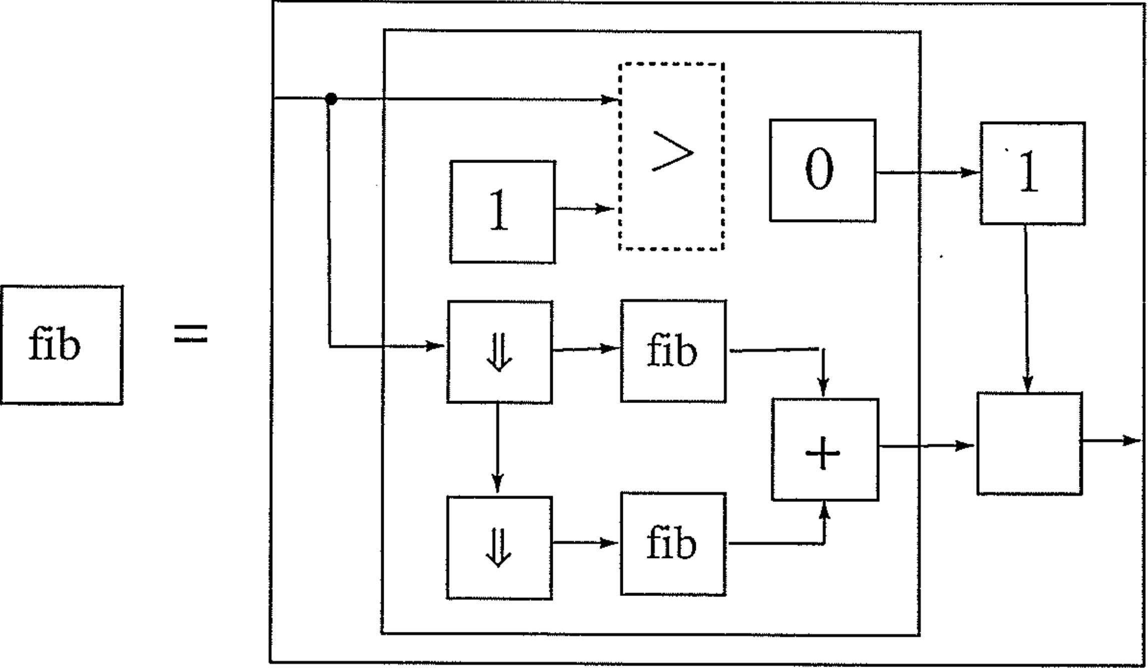


Figure 12: (a) Fibonacci Function

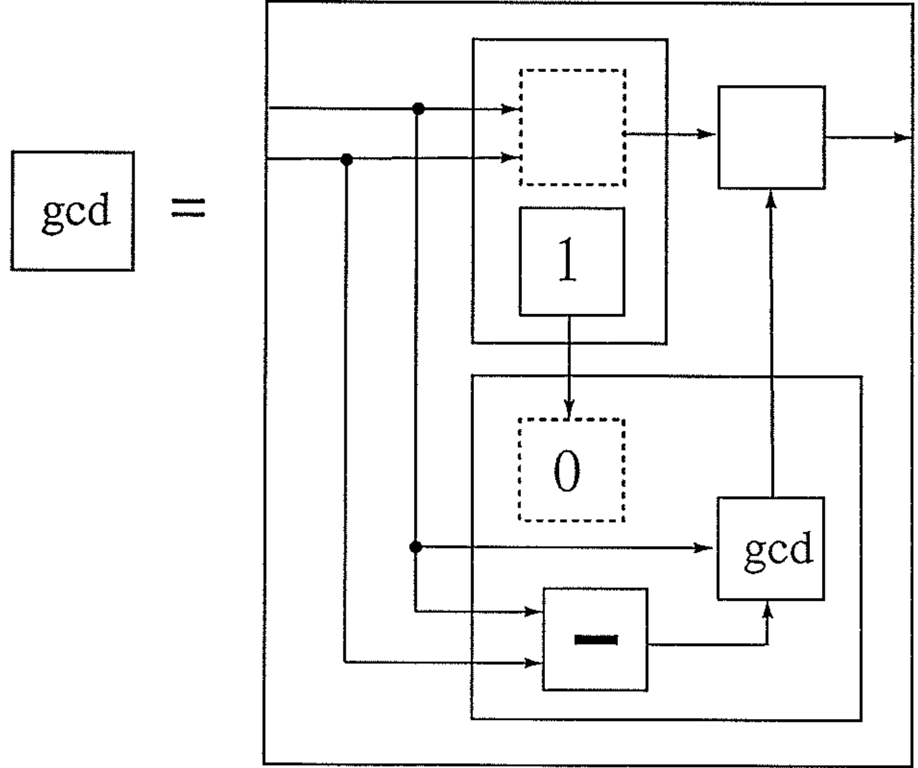
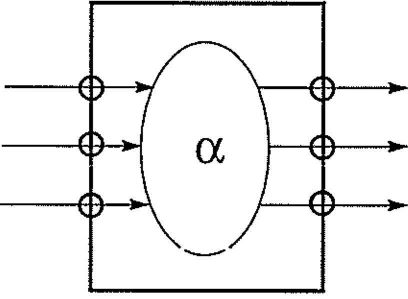
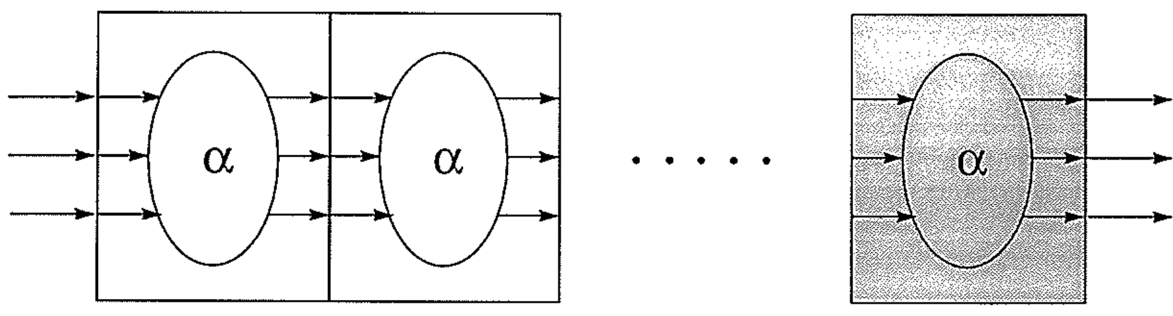


Figure 12: (b) GCD Function

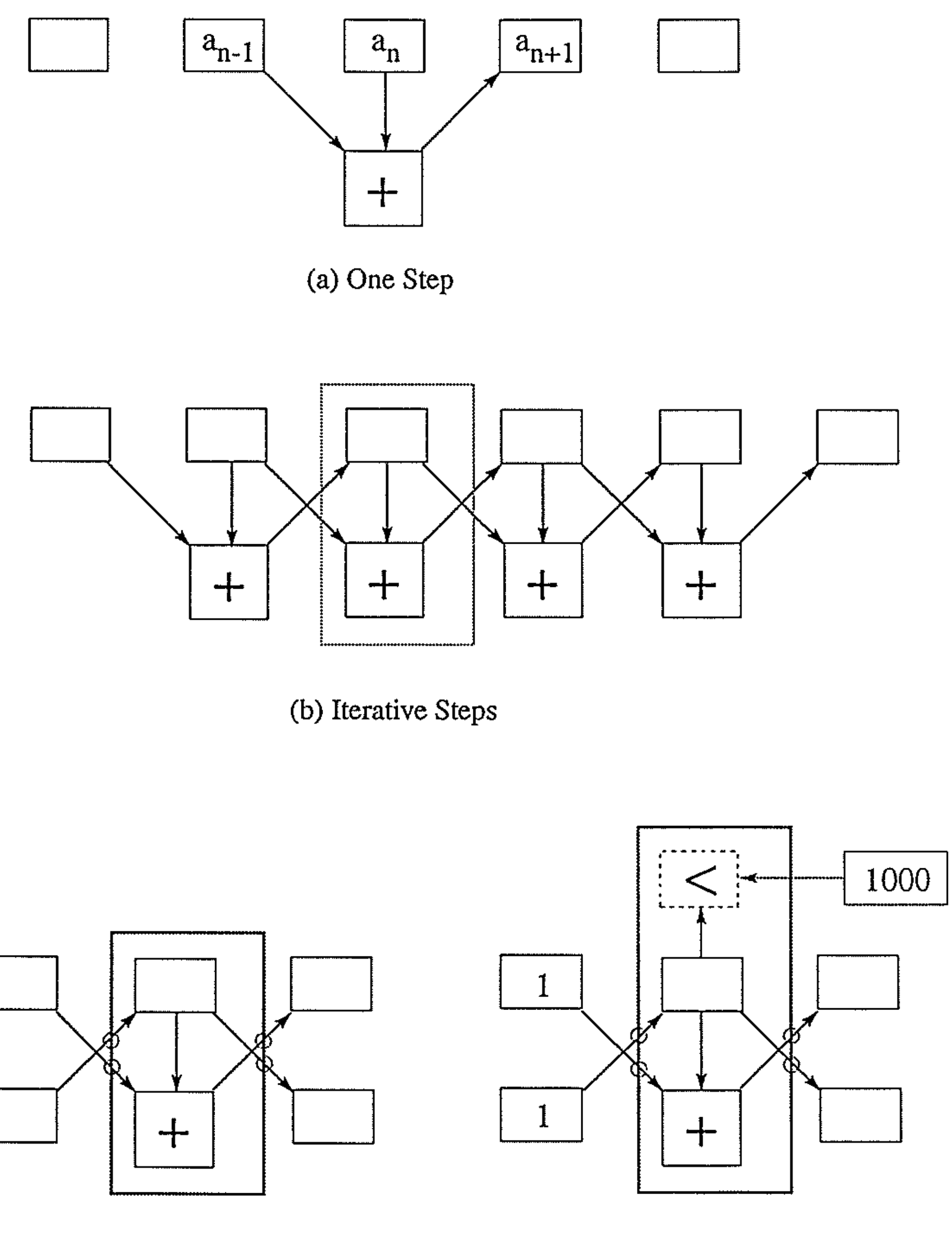


(a) Iteration Box (Syntax)



(b) Unfolding of Iteration Box (Semantics)

Figure 13: Horizontal Abstraction



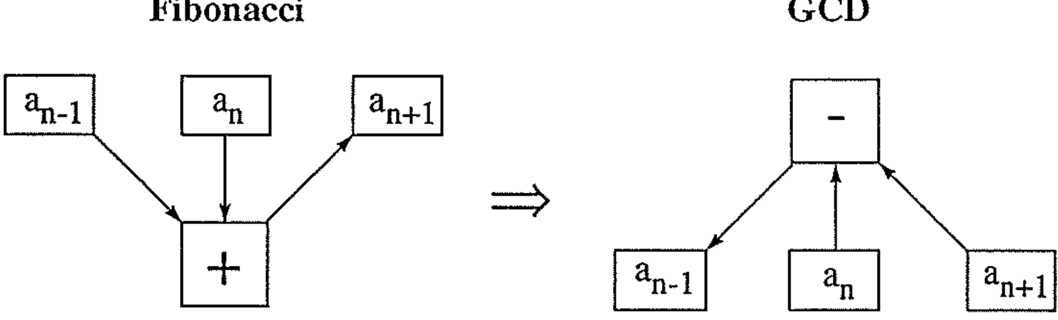
(a) One Step

(b) Iterative Steps

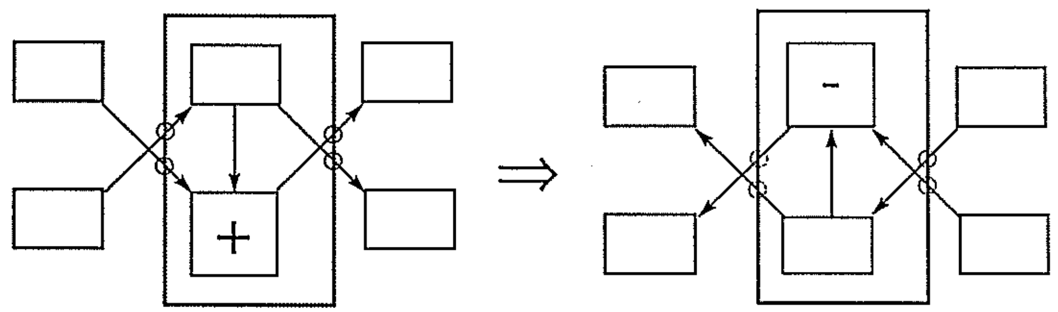
(c) Horizontal Abstraction of (b)

(d) Fibonacci < 1000

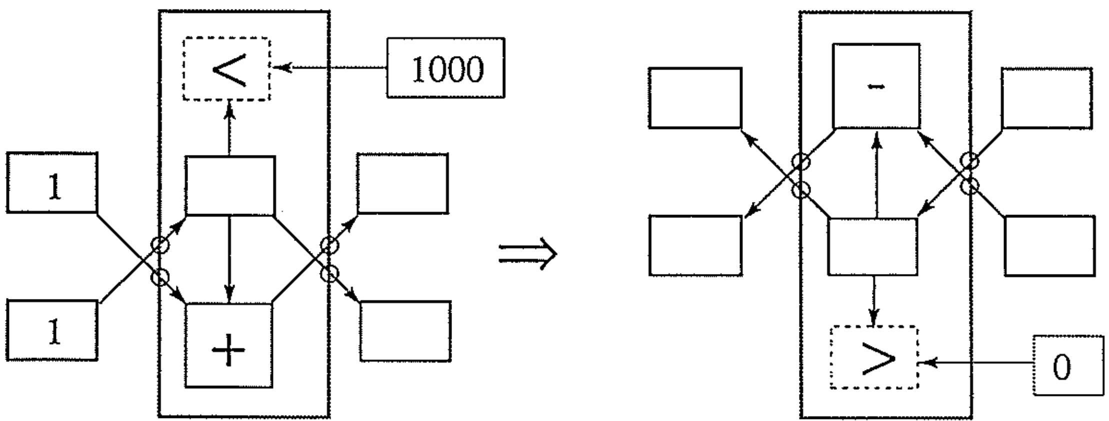
Figure 14: Fibonacci Program in SHF



(a) Inversion of One Step



(b) Inversion of Iteration Box



(c) Inversion of Bounded Iteration

Figure 15: Inversion of Fibonacci for GCD

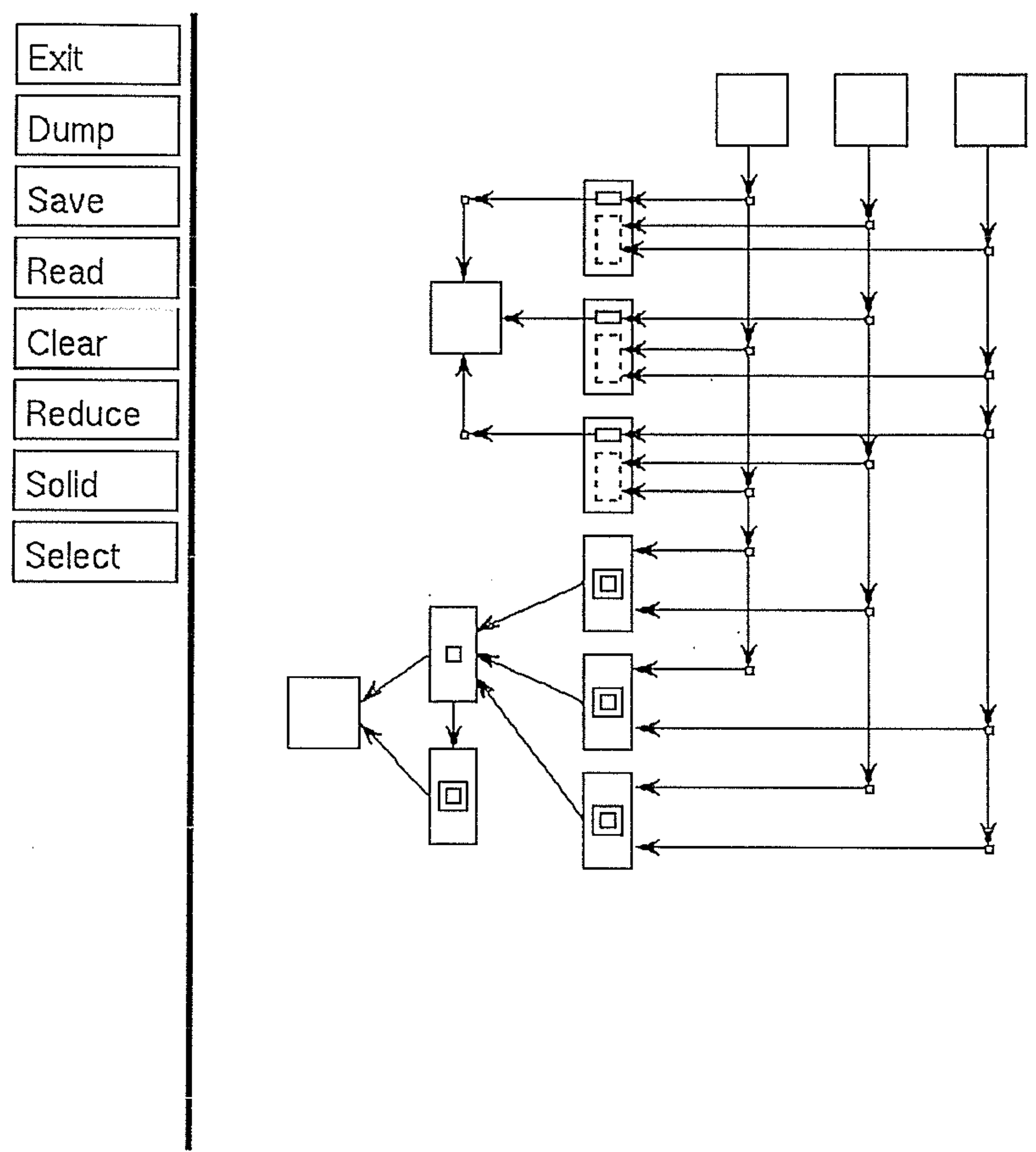


Figure 16(a): Screen Dump of Binary Full Adder Before Execution

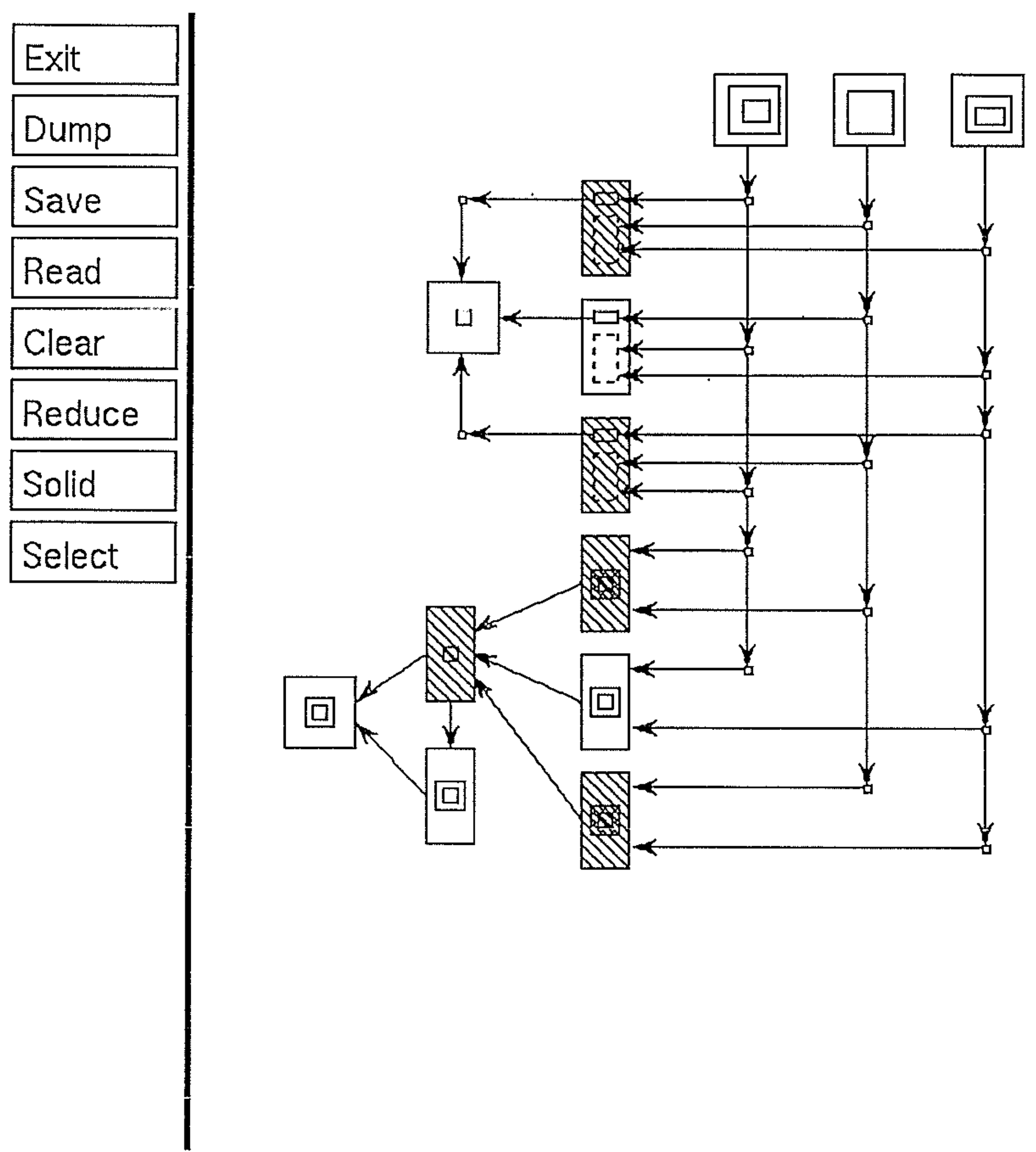


Figure 16(b): Screen Dump of Binary Full Adder After Execution

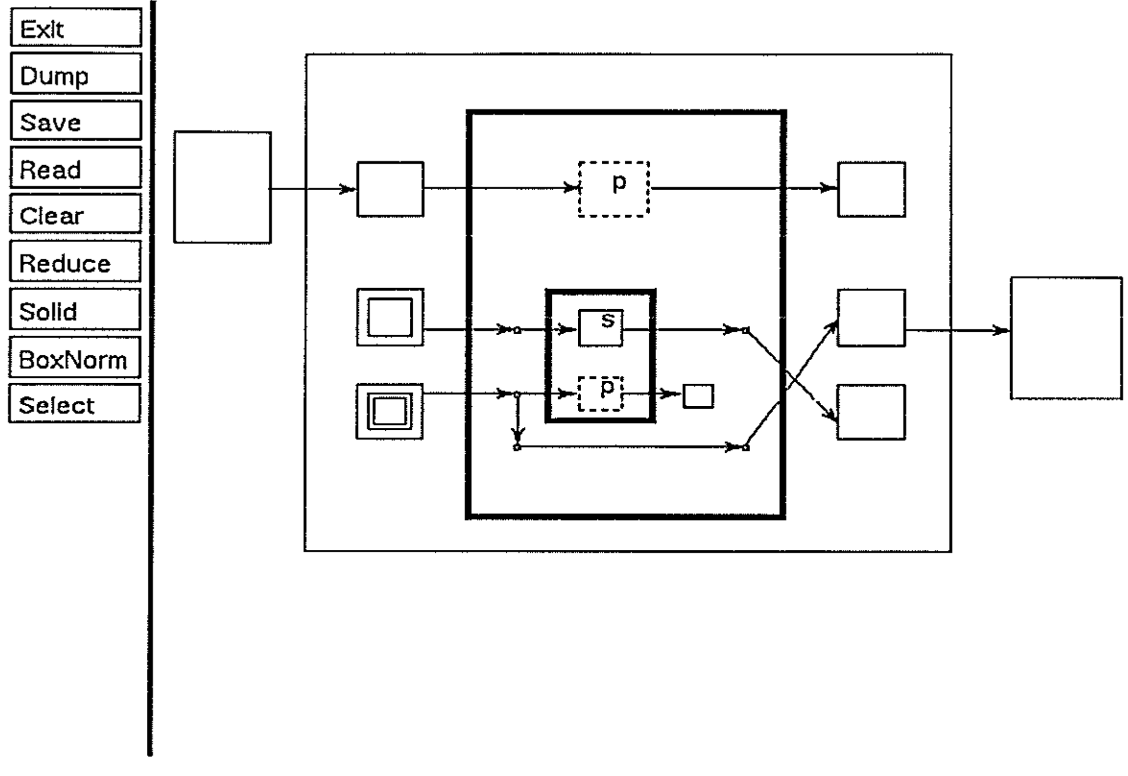


Figure 17(a): Screen Dump of Fibonacci Function Before Execution

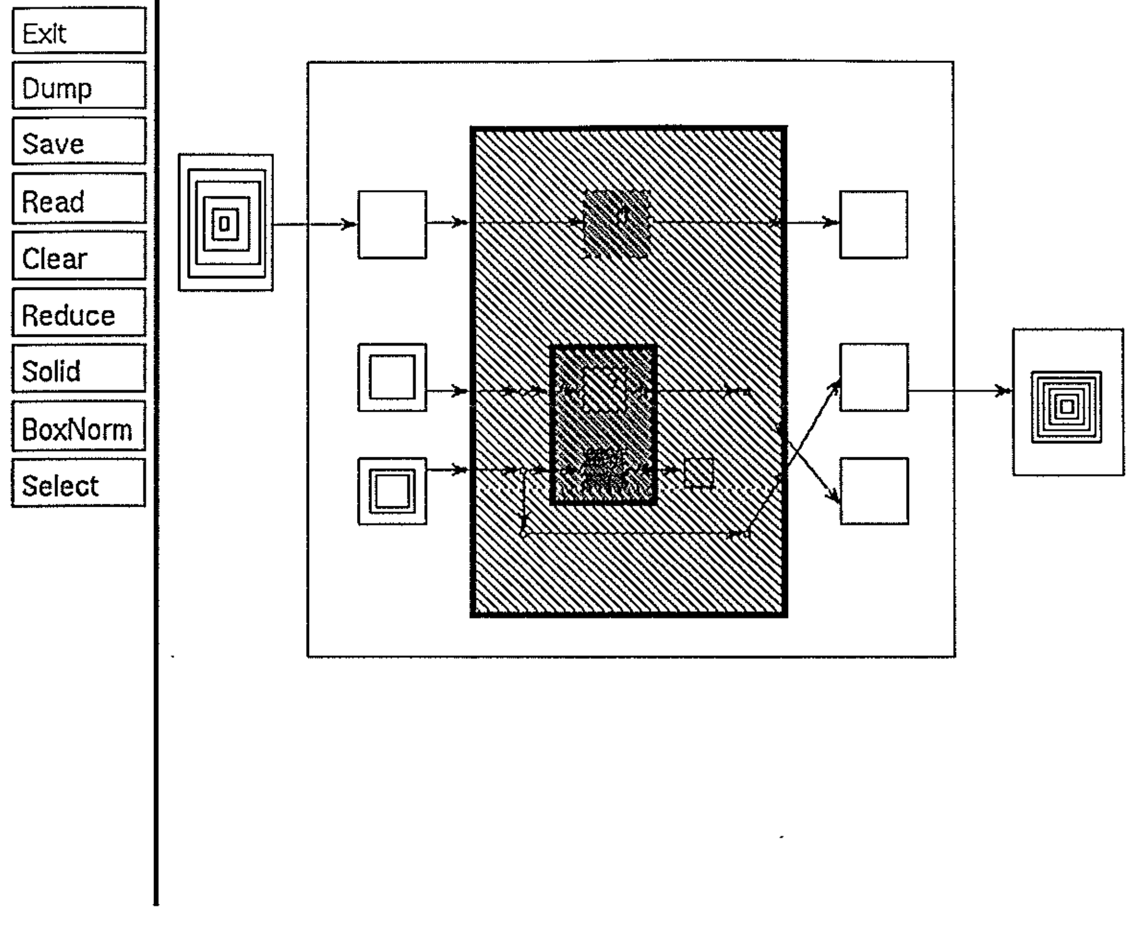


Figure 17(b): Screen Dump of Fibonacci Function Binary After Execution

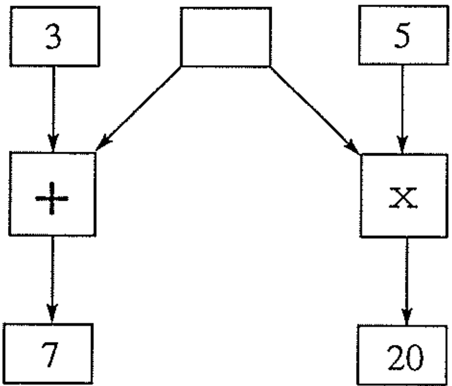


Figure 18 (a): Bidirectional Constraint

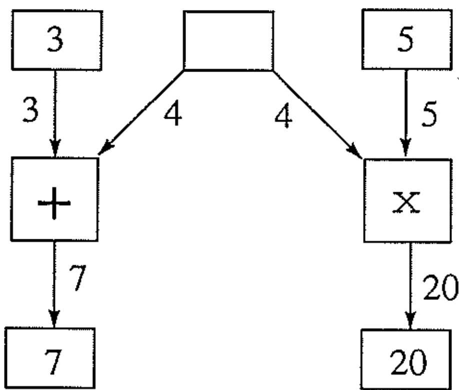


Figure 18 (b): Elaboration for (a)