Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

AN APPLICATIVE LANGUAGE FRAMEWORK FOR HARDWARE SYNTHESIS

Dorab Ratan Patel

October 1990
CSD-900031

An Applicative Language Framework for Hardware Synthesis

Dorab Ratan Patel

September 18, 1990

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Miloš Ercegovac, for his patience, encouragement, and wisdom. Miloš is probably the best advisor anyone could ever have. He knew when to challenge me into deeper explorations and also when to leave me to explore on my own. I have learned a lot from Miloš—not only about computer science, but also about life. I would like thank Gerald Estrin for continued support throughout my graduate career, especially when I needed it the most. Jerry was always there with thoughtful advice whenever I needed him. The members of my committee: Stott Parker, Rajeev Jain and Kirby Baker deserve thanks for their comments and participation. Stott provided particularly useful feedback on appendix B.

During my stay at UCLA I have had the opportunity to interact with many different students, especially in the SARA/IDEAS and FP/VLSI groups. Discussions with Martine Schlag helped clarify many of the ideas presented here. Martine also wrote the tools which were used to produce the topological layout diagrams in this dissertation. John Harding performed way beyond the call of duty helping me achieve self-sufficiency with the Lager tools.

My sojourn at UCLA was made much more comfortable by the staff of the department. In particular, I would like to thank Verra Morgan for her help easing the way through the bureaucratic labyrinth of graduate studies. Doris Sublette, of the departmental archives, was an invaluable source of information, obscure and obvious.

I have been fortunate to have friends who, by sharing the good times and bad, made the process of getting a PhD much more bearable. Beheroze Shroff and Ketu Katrak have been my friends almost since the first day I arrived in the U.S. In addition, the denizens of "Hershey Hall West" provided a congenial, supportive and neighborly environment. My "extended roommates"—Kulvant Lail, Charles Tashiro, Sarma Sastry, Neela Sastry and Christian Matthiessen—supplied companionship, food, and ... dessert. Charles also gets special mention for a thorough proof-reading of the dissertation.

Lastly, I would like to thank my parents, Mani and Ratan Patel, without whom this would not be possible. What I am today is due in large part to them.

This research was supported in part by a fellowship from Systems Engineering Laboratories, a State of California MICRO Fellowship, the Department of Energy, the Office of Naval Research, and the State of California MICRO contracts in conjunction with Systems Development Corporation, Hughes Aircraft Corporation, NCR, TRW and Rockwell International.

Abstract

This dissertation presents a method, based on applicative languages, for the specification, analysis, synthesis and implementation of hardware algorithms. Using the same language to represent the algorithm from specification to mask generation provides a consistent and coherent framework. Designers are provided with an environment in which they can efficiently explore alternative designs for their algorithms throughout the synthesis process. It is possible to specify the algorithm at arbitrary levels of abstraction and have the system rapidly evaluate certain parameters (e.g., speed, area) so that designers can make informed decisions during the synthesis process. Semantics-preserving transformations convert a specification to an implementation that is guaranteed to be correct. Evaluations of designs with respect to performance parameters are performed at any desired level of abstraction. A visual display of the algorithm throughout the design process provides designers with feedback on the spatial implications of their design decisions. Using an applicative language during design helps to prove the correctness of refinement and abstraction. Symbolic evaluation of the algorithm with representative symbolic inputs is used to generate a planar topological layout of the algorithm. Sequential circuits are incorporated into this applicative framework using space/time transformations. An interface to a back-end system is used to generate mask layouts from topological descriptions.

Contents

Chapter 1

Introduction

A discipline for complexity management is the key to the effective design of any large scale computer system, be it hardware or software [1, 2]. The design complexity in a large scale computer system comes from two major sources. The first is the large number of primitives involved. In the case of hardware, they could be transistors or chips. In the case of software, they could be lines of source code. However, a large number of primitives does not necessarily make for complexity. For example, a memory chip has a large number of transistors but is not necessarily complex.

A second source of complexity is the irregularity of connections or couplings between primitives. Traditional software design has been made simpler because the underlying model of computation has been a single sequential machine. In addition to handling concurrency, hardware design deals with the mapping of a design into a physical medium. This mapping has to be performed in the presence of constraints on power, space and time. The primarily two-dimensional nature of integrated circuits and the need to have direct physical connections between interacting modules makes the synthesis process even harder. These issues make the process of designing large scale hardware more difficult than that of software. However, lessons learned in software engineering can be applied to the hardware design process to address some of these problems.

This research concentrates on problems arising out of the process of designing hardware. A method, based on applicative languages, is proposed for the specification, evaluation and synthesis of hardware algorithms. Using the same language throughout the synthesis process leads to a coherent framework for design. Properties of applicative languages make them attractive for handling the complexity of large-scale design. The mathematical foundations of these languages enable the development of an algebra of programs that can be utilized both in the synthesis process and in proving properties about the designs.

The complexity of designing Very Large Scale Integration (VLSI) circuits can only be managed by the application of Computer-Aided Design (CAD) tools at all levels of the design process. In order to be effective, these tools must be flexible enough to be tailored to any specific design. Generally, VLSI CAD tools may be distinguished as being of either or both of two types: bottom-up composition tools or top-down synthesis tools. For bottom-up composition tools, the user either exactly specifies the placement of modules and the interconnections between them, or relinquishes control over the layout to the tool's algorithm. Examples of composition tools are graphic layout editors (e.g., Magic [3]) and placement and routing tools (e.g., PI [4, 5]). Top-down synthesis tools are capable of generating layouts from high-level specifications. Examples include various register-transfer silicon compilers that have been proposed and built [6, 7, 8].

Generally, these tools only provide an estimate of the area or delays of the circuit at the end of the synthesis process. That is, designers do not know the effects of their decisions on the performance until the design is complete. This makes the design process more batch-oriented rather than interactive. It also makes it difficult to explore alternative implementations and to evaluate objectively the alternatives with respect to user-specified criteria.

Small designs can be comprehended easily by a designer. As the size of designs grows larger it becomes more important to be able to formally prove properties about the design. Ideally, it should be possible to prove that the implementation meets the specification, but this is not always feasible. A formal basis for the representation of the design provides a means to reason about the representation.

In order to prove that the implementations match the specifications, the specifications themselves have to be generated in a suitable specification language. There are many kinds of specifications required in VLSI design. At highest level, the functional behavior of the algorithm is of interest. At lower levels it is the interfaces between modules and the interconnections between modules that are important. At the lowest level, it is the spatial positioning of modules and the geometry of their layout and interconnections that are important. At all these levels the timing of the design has also to be specified. At the higher levels the timing is abstract (e.g., sequencing), but at lower levels the specification of timing is more concrete (e.g., propagation and inertial delay).

Though most specifications tend to be described in textual languages, VLSI design, especially at the lower levels, is oriented towards layout on silicon and hence spatial in nature. The design process would be aided if a visual representation of the specifications and the implementations were provided at all levels of abstraction.

In the following sections, each of these issues is taken up in turn. Various problems are exposed and their solutions discussed.

1.1 Complexity Management

This section explores some methods that are useful in managing complexity in VLSI design.

1.1.1 Abstraction

Using a computer language for description makes any description more precise since these languages have specific semantics. In addition, using a language description for an intuitive process aids in the documentation of that process by providing a written description of what the designer envisions. Some languages (e.g., Ada[1]) also provide the ability to separate the interface of a function from the body of the function. This capability allows the concerns of describing the interface to be treated separately from the concerns of designing the body. All languages provide some form of abstraction. An *abstraction* is a simplified description, of a system that emphasizes some of the system's details or properties while suppressing others. A *good abstraction* is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary [9]. One useful example of abstraction that is commonly found in computer languages is *functional abstraction* where a particular process, used with different parameters, is abstracted with respect to those parameters, to form a function. Timing or type abstraction are examples of other kinds of useful abstractions.

The process of designing Integrated Circuits (ICs) can be described by the Gajski-Kuhn Y-chart [10] (figure 1.1) where each axis signifies one representation of the circuit. The representations towards the center of the chart are less abstract than the ones towards the outer edges. The synthesis process starts at the most abstract behavioral representation and ends at the most concrete geometric representation. The actual path followed is dependent on the particular synthesis method employed. Thus we see that in the VLSI domain there are three main classes of abstractions—structural, behavioral and geometric.

Structural Abstraction

Structural abstractions deal with the composition and interconnection of modules at one level of abstraction to construct a module at a presumably higher level of abstraction. Language designers have taken different approaches to the problem of describing such relationships.

One approach is "connection-oriented" in the sense that it explicitly specifies which terminal is to be connected to which other terminal. This is equivalent to using a net-list to describe

[1] Ada is a trademark of the US Department of Defense (Ada Joint Program Office).

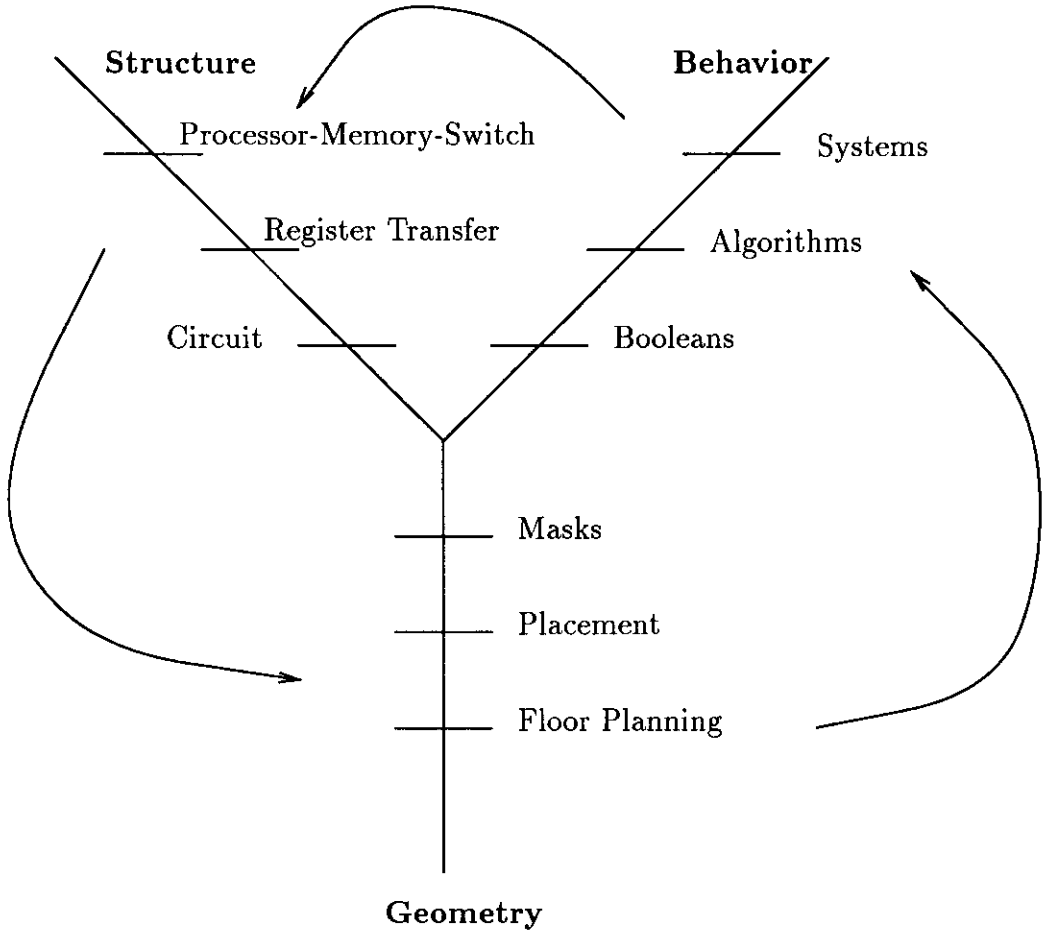


Figure 1.1: Gajski-Kuhn Y-chart

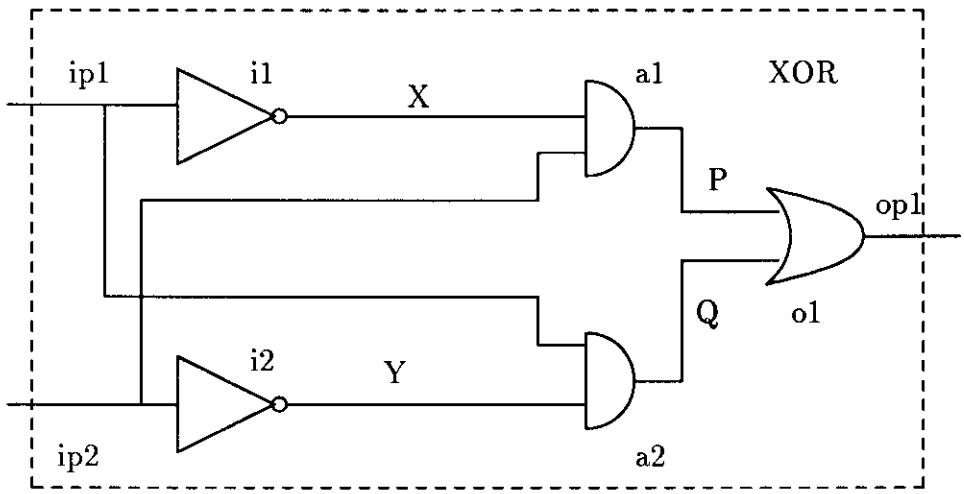


Figure 1.2: XOR gate implementation

the connections. Hardware design and description languages like SILAGE [11] and ELLA[2] [12] use such an approach. For example, an exclusive OR gate implementation shown in figure 1.2 could be described by the ELLA program in figure 1.3 (taken from [13]). The `MAKE` statement specifies the component modules of the current module and the `JOIN` statement specifies which connections are made. For example,

```
(i1, ip2) -> a1
```

means that the output of `i1` and the input terminal `ip2` are to be connected to the two inputs respectively of the module `a1`.

Concurrent Prolog [14, 15] uses a similar approach but it names each connection and thus does not explicitly specify the endpoints of each connection. For example, the ELLA example in figure 1.3 could be written in Concurrent Prolog as shown in figure 1.4. The basic model of Concurrent Prolog is that of processes communicating through ports and thus this is a good match to the requirement of describing interconnected modules. This approach allows arbitrary connections to be specified between terminals. Though this makes the approach very powerful, it is not structured and can lead to situations that are difficult to analyze. However, CIRCAL [16, 17] provides a formal basis for describing interconnections between modules and a mechanism for structural abstraction by abstracting away internal ports and connections.

Another approach specifies connectivity implicitly. Functions correspond to modules. Parameters correspond to input ports and result values to output ports. The call graph of the

[2]ELLA is a trademark of the UK Secretary of State for Defense.

```
FN XOR = (bool: ip1 ip2) -> bool:
BEGIN
   MAKE AND: a1 a2,
        INV: i1 i2,
        OR: o1.
   JOIN ip1 -> i1,
        ip2 -> i2,
        (i1, ip2) -> a1,
        (ip1, i2) -> a2,
        (a1, a2) -> o1.
   OUTPUT o1
END.
```

Figure 1.3: Explicit ELLA description of XOR

```
XOR(ip1, ip2, o1) :-
  INV(ip1, X),
  INV(ip2, Y),
  AND(X, ip2, P),
  AND(ip1, Y, Q),
  OR(P, Q, op1).
```

Figure 1.4: Concurrent Prolog description of XOR

```
FN XOR = (bool: ip1 ip2) -> bool:
  OR( AND(INV(ip1), ip2),
      AND(ip1, INV(ip2))  ).
```

Figure 1.5: Implicit ELLA description of XOR

functions corresponds to the structural hierarchy of the modules. The naming of parameters corresponds to the interconnections between the modules. Languages like μFP [18] and $\nu\mathcal{FP}$ (described in [19] and this dissertation) follow this approach. SILAGE and ELLA support both approaches. For example, an implicit version of the ELLA description of the XOR gate in figure 1.2 is shown in figure 1.5. Though this approach makes it easy to analyze the behavior of the resulting connections, it is less powerful than the connection-oriented method. For example, it is not possible to describe cross-coupled NAND gates in this manner.

Languages like VHDL[3] [20] and Standard ML (SML) [21, 22] allow the designer to separate the concerns regarding the specification and checking of interface descriptions and the design of the body of the functions. In such languages the module hierarchy can perform the role of the structural hierarchy. For example, in VHDL, the interface description of the XOR gate could be described as shown in figure 1.6 and the body of an implementation corresponding to figure 1.2 is shown in figure 1.7.

Replication of cells to form regular structures is provided for in all the languages mentioned above. VHDL provides the *generate* statement, ELLA provides the *replicator* construct and μFP and $\nu\mathcal{FP}$ use the *insert* form.

The constructs provided by $\nu\mathcal{FP}$ for structural abstraction are mentioned in chapter 2 which also gives an overview of the $\nu\mathcal{FP}$ system. The use of these constructs is shown in chapter 6 and a comparison with μFP's structural constructs is taken up in chapter 7.

Behavioral Abstraction

Behavioral abstraction groups behaviors together to form other behaviors at a higher level of abstraction. Behavior can be separated into the functional aspects (i.e., what function is performed by this module) and the timing aspects. In some cases, this separation can not be made cleanly since the function performed by the module might well depend on the timing of submodules.

[3]VHDL is a trademark of the US Department of Defense.

```
entity XOR_GATE
  -- ports
  (ip1,ip2: in BIT   -- inputs
   op1: out BIT) is  -- output

  -- parameter and default
  generic
   (DELAY: time := 32ns);

  -- assertions that all bodies must satisfy
  assertion
   DELAY > 10ns;
   op1'FANOUT <= max_fanout;
end XOR_GATE;
```

Figure 1.6: VHDL interface description of XOR gate

Functional abstraction is provided by all programming languages. However, only some of them allow easy composition of behaviors in a way that is guaranteed not to introduce unwanted interactions. This issue is taken up in more detail in section 1.1.2. FP is one language that provides for easy composition and the XOR gate example of figure 1.2 can be described as shown in figure 1.8. It shows how to combine the behaviors of AND gates, OR gates and inverters to result in the behavior of an XOR gate. Though pure CIRCAL only provides a mechanism for structural abstraction, an extension called TYPED CIRCAL [23] also allows data abstraction to be treated in a formal framework.

Another important benefit of behavioral abstraction is the ability to evaluate, simulate, verify or manipulate a cell at different levels of abstraction. For example, a transistor could be described as a rather detailed SPICE [24] model to evaluate its low level timing behavior. Then, transistors could be grouped together to construct a NAND gate. Once the characteristics of the NAND gate have been determined using the characteristics of its component transistors, it is no longer necessary to retain all the lower-level information about the transistors. It is possible to now use the NAND gate as a higher-level logical entity with a functional behavior in terms of boolean algebra (or, at least, discrete logic levels corresponding to booleans) and with a timing behavior in terms of a propagation delay. A simulator capable of using descriptions at various levels of abstraction simultaneously is described in [25].

```
architectural body GATE_DESC_OF_XOR of XOR_GATE is

  component AND_GATE (A,B: in BIT; OUT: out BIT);
  component INV_GATE (A: in BIT; OUT: out BIT);
  component OR_GATE (A,B: in BIT; OUT: out BIT);
  signal X,Y,P,Q: BIT;  -- internal signals

begin

  i1: INV_GATE (ip1, X);
  i2: INV_GATE (ip2, Y);
  a1: AND_GATE (X, ip2, P);
  a2: AND_GATE (ip1, Y, Q);
  o1: OR_GATE  (P, Q, op1);

end GATE_DESC_OF_XOR;
```

Figure 1.7: VHDL body for XOR gate

```
or °[ and °[ inv °1, 2 ],
      and °[ 1, inv @ 2] ]
```

Figure 1.8: FP description of an XOR gate

Timing in VLSI design needs to be specified in many ways. At the highest levels of abstraction, all that is required is the specification of sequencing information. That is, only the dependencies between computations need to be specified. As the algorithm is mapped to specific hardware, more concrete timing is specified, especially in a synchronous system. In particular, each computation will typically be mapped to a specific element of a time sequence in such a way that the original precedence constraints are satisfied. At still lower levels, each element of the time sequence is mapped to a particular point on a concrete time scale. This implies that the absolute delays associated with each computation have to be specified. These can be specified as inertial, transport or propagation delays.

There are many different models of time that are used by various languages. These models can either be asynchronous or synchronous. VHDL, for instance, uses a synchronous model of time that is similar to that used in CONLAN [26], with a macro and micro time scale. The micro time scale is measured in discrete units and is used to represent real time. The macro time scale denotes the unit delay of the cell and is not measurable. A macro time step can contain any number of micro time steps, and the number of micro time steps contained in each macro time step need not be the same. This allows the designer to simultaneously model the timing behavior at two levels of abstraction.

Interval Temporal Logic (ITL) [27, 28] allows the specification of digital circuits with a discrete sequential model of time. ITL is an extension of temporal logic, itself an extension of predicate logic, to handle dynamic behavior. Time is split up into discrete intervals and changes in values of signals are allowed only at interval boundaries. ITL makes no assumptions on the lengths of these intervals. In particular, they all do not need to be the same length. Thus, in addition to the normal process of data abstraction, it is also possible to specify timing at different levels of abstraction. ITL does not provide any way to specify time in actual units.

TYPED CIRCAL [23] introduces timing abstraction into pure CIRCAL. This allows the representation of a hierarchy of clocks where n clock ticks of a particular clock can be replaced by one clock tick of a less frequent and more abstract clock. Pezzé [23] shows how to use *time conversion* to convert a representation from one clock to another. In TYPED CIRCAL timing abstraction is dependent on data abstraction and both may need to be performed to verify that a particular implementation meets its specification. The time synthesis procedure described in section 6.4 follows a similar approach which results in hierarchically nested clocks.

Concurrent Prolog [15] describes processes communicating via streams connected to ports. At each level of abstraction, a circuit block is represented as a predicate and the wires connecting them are represented as the parameters of that predicate. Each parameter is treated as an infinite stream which represents the stream of values on that wire. Synchronization is performed by declaring some uses of a parameter as read-only. In this case, expressions using the read-only parameter will block until the value of the parameter has been instantiated. It is

not possible to directly specify timing in Concurrent Prolog. However, a clock can be modeled as an infinite sequence of ascending natural numbers. This stream has to be explicitly fed to every circuit that uses it. Each circuit can then synchronize itself with this stream by using it as a read-only parameter. In this sense, it can be said that Concurrent Prolog uses a sequential model of time. This is only true if such a "clock" stream is used. Otherwise, the language is relational and has no inherent notion of time.

The Graph Model of Behavior (GMB) [29] is an extension of Petri Nets [30] and is particularly useful to model concurrent asynchronous events. This model is event driven and does not have any inherent notion of synchronous behavior. All that it can specify is the precedence between events. However, synchronous behavior can be modeled, just as in Concurrent Prolog, by making the clock explicit and feeding it to each circuit that requires it.

Chapter 2 describes $\nu\mathcal{FP}$ constructs that can be used for behavioral abstraction. Examples of their use are provided in chapter 6 which also shows how to introduce sequential behavior into $\nu\mathcal{FP}$.

Interaction between Structure and Behavior

Each system has different ways of describing structure and behavior. Some systems such as VHDL use different constructs in the same language to describe the structure and behavior. For example, the `architectural body` construct is used to describe the structure, and the `behavioral body` construct is used to describe the behavior.

Design systems such as the Systems ARchitect's Apprentice (SARA) [29] provide a separate language to describe the structure of the design and a different language to describe the behavior. The SARA system also provides a mechanism to map the behavior to the structure. In fact, it is possible to map different behaviors to the same structure. This is similar to VHDL's mapping of different behavioral bodies to the entity interface description.

It has been suggested [2] that a structured design methodology for VLSI should have a one-to-one mapping between behavior and structure in order to manage the complexity. $\nu\mathcal{FP}$ and μFP take this approach. The notions of structural and behavioral specification are tightly integrated in these systems. During the design process, a particular description can be treated either as behavioral or structural depending on what the designer wants to emphasize at that time. For example, the description in figure 1.8 can be taken to mean that the behavior of an XOR gate can be specified in terms of the behavior of AND gates, OR gates, and inverters as shown in the figure; or it can mean that an XOR gate can be constructed structurally out of AND gates, OR gates and inverters as shown in the figure. In $\nu\mathcal{FP}$, the interpretation chosen is dependent on whether the module is tagged as a primitive at this level of abstraction. If it is tagged as a primitive, then its internals are treated as strictly behavioral by the system. If not,

```
behavioral body XOR_BEHAVIOR of XOR_GATE is
begin
  op1 <= ip1 xor ip2 after 35ns;
end XOR_BEHAVIOR;
```

Figure 1.9: Behavioral description of XOR gate in VHDL

```
architectural body XOR_STRUCTURE of XOR_GATE is
  component NAND_GATE (A,B: in BIT; C: out BIT);
  signal X,Y,Z: BIT;  -- internal signals
begin
  n1: NAND_GATE (ip1, ip2, X);
  n2: NAND_GATE (ip1, X, Y);
  n3: NAND_GATE (X, ip2, Z);
  n4: NAND_GATE (Y, Z, op1);
end XOR_STRUCTURE;
```

Figure 1.10: Alternate structural implementation of XOR gate in VHDL

then the internals are treated as a structural combination of modules. Chapter 4 demonstrates this with an example.

However, structure and behavior are independent and conceptually orthogonal and it is possible to change the structural hierarchy without changing the behavioral hierarchy. For example, figure 1.7 and figure 1.10 show two different structural implementations of the XOR gate whose behavior is specified in figure 1.9 and which meet the interface specifications of figure 1.6. In the case that the structural and behavioral hierarchies are distinct a mapping between the two has to be provided [31, 32]. It is not clear which approach is better in all situations. When the design complexity is high, experience shows that the payoffs of using the former approach compensates for the attendant loss in flexibility.

Geometry Abstraction

The third domain of abstraction in VLSI is that of geometry. At the lowest levels geometry consists of overlapping rectangles of different colors representing the different layers on a silicon

chip. At the next higher level, the rectangles can be combined in particular fashions to form transistors or other circuit elements. Circuit elements are combined together to form cells. Reusable cells (e.g., gates, adders, ALUs) can be placed in a library for use by designers. Such cells are treated as fixed size rectangles containing geometry with ports on the edges. These ports have a fixed location with respect to the cell and are in a particular silicon layer. In some systems, it is possible to stretch cells either in the vertical or horizontal direction along specific cross-sections. In those cases, the rectangles that are cut by the cross-sections are extended as necessary to effect the stretching of the cell. This is done in order to match the pitch of adjacent cells so that the interconnections between them can be achieved via abutment rather than routing. A designer can now use this as a piece of rectangular geometry without being concerned about the geometry contained inside. For example, there is no need to re-verify that design rules are satisfied inside the cell when the cell is used in conjunction with other cells in a design. Depending on the design, it may still be necessary to check for design rule violations at the edges and at the ports. A procedure for analyzing a hierarchical geometric description and filtering out only the parts that need to be checked is described by Whitney [33]. Cells may be combined with other cells and connecting rectangles to form other higher-level cells of geometry.

The consistency checking task is eased if a *separated hierarchy* [34] is used. A separated hierarchy is one in which the representation-dependent information is only present in leaf cells. All the other cells are composition cells. Composition rules must be established to determine how to implement a composition cell within the given representation. In such a system, only the leaf cells and the composition rules need be checked instead of all the geometry. Rowson, in [35], describes a rigorously defined and efficient composition rule.

An applicative framework can be used in different environments by just changing the set of primitives used and possibly the combining forms. Henderson [36] describes an applicative system for describing geometry and for combining pieces of geometry in different ways. The μFP system uses this system as a back-end for generating layouts suitable for VLSI implementation.

1.1.2 Composition

Another way to manage complexity is to provide a formal way of combining smaller pieces into larger ones. Most systems provide for ad hoc combining of modules. However, these methods do not guarantee any properties of the combination and hence it is difficult for the designer to treat the combination as a module without a thorough analysis of the components. For example, it is not possible to arbitrarily combine VHDL entities without analysis because they may refer to global constructs. If the global constructs used independently by the constituents conflict in some way, the designer will be faced with unwanted interactions between the constituents.

These interactions cannot be checked for by just examining the interfaces of the modules being combined, but the behavioral body of each will have to be analyzed completely before a claim of non-interference can be made. The use of packages alleviates but does not eliminate this problem.

In contrast, it is possible to combine a $\nu\mathcal{FP}$ description of a module with any other $\nu\mathcal{FP}$ description and to be guaranteed that they will not interfere with each other because systems like $\nu\mathcal{FP}$ are free of side-effects. In addition, proven laws about $\nu\mathcal{FP}$ combining forms may be used to reason about the behavior of the combined whole if the behavior of the constituent parts is known. Systems proposed by Cardelli [37] and Milne [38, 17] demonstrate mathematical formalisms for describing networks of interconnected modules. They also provide methods for formally reasoning about combinations of interconnected modules that form higher-level modules.

In general, modules can be connected in any ad hoc manner as long as the port constraints are satisfied. For example, it is syntactically valid to connect modules together as long as an input port on one module is connected to an output port of another. However, in many cases, designers use certain regular or structured combinations or idioms. For example, cell-iterative or parallel arrays are often used in regular design. Languages like $\nu\mathcal{FP}$, μFP and AHPL [39] provide convenient combining forms to be used in such situations. More than providing convenience, the structured nature of these connections allows $\nu\mathcal{FP}$ to prove theorems that can be used to deduce properties of the combination given the properties of the cell. Chapter 2 documents the combining forms available in $\nu\mathcal{FP}$. Some of the more useful forms are diagrammed in Chapter 6. Section 7.6 shows how the behavior of the composition of two sequential machines can be derived from the behavior of the individual machines.

1.1.3 Design Framework

Complexity can be managed effectively if all the various levels of the design process are carried out within a coherent design framework. A design framework can help in the integration of tools. There are two approaches to providing coherence. One way is to use the same language to describe the design at all levels of abstraction. This approach has the advantage that the designer has to learn only one language. Additionally, it provides coherence by using the same language throughout the design process. $\nu\mathcal{FP}$ uses this paradigm. Chapter 8 describes the components of the $\nu\mathcal{FP}$ system.

The other approach is to use different languages to represent the design at each level of abstraction. The advantage of this approach is that each language can be tailored for that level of abstraction. The semantic gap, between the concepts used by the designer at that level and the constructs supplied by the language, can be reduced by an appropriate choice of

constructs. In addition, the framework usually provides a common data representation that is used by each tool to communicate with other tools. The set of OCT tools [40] from the University of California at Berkeley is one example of such a framework.

1.2 Tools

Design tools often are overly specific in their domain of applicability. Often they are constructed that way because they are application-specific. It is much better to make a general design tool that can be tailored to the application area. The basic framework can then be shared among various application-specific tools. Another way to increase the domain of applicability of a tool it to make it extensible. Most tools are not and this make them frustrating to the user. The problem is that the tools are fine as long as the problems being addressed are what had been envisioned by the designer. Often enough, however, users are faced with problems that are just slightly different than what had been envisioned by the tool designer. In this situation, most tools prove to be inflexible and unusable. The solution is to provide tools which exist in a framework that can be easily extended by the user to cover the cases the existing tools do not.

One approach to extensibility is to use an embedded language. One of the earliest VLSI tools was LAP [41] which embedded drawing commands in SIMULA [42]. Thus, with very little effort, it was possible to create a VLSI layout system with the full power of SIMULA. The DPL/Daedalus [43] system takes a similar, but more sophisticated approach by embedding a description and constraint system in LISP [44]. Macpitts [6] and its commercial descendent MetaSyn [45], are also embedded in LISP but have typed variables and allow user extensions. The $\nu\mathcal{FP}$ system is user-extensible and tailorable. Users can extend the primitive function set to suit their requirements. In addition, the $\nu\mathcal{FP}$ framework is applicable to different domains by just changing the primitive function set and combining forms as required. Chapter 2 describes the primitives and functional forms of the $\nu\mathcal{FP}$ system.

The most common instances of application-specific languages and systems are those developed for Digital Signal Processing (DSP). The FIRST silicon compiler [46] was a system that was developed to generate bit-serial implementations of DSP algorithms. The language SILAGE [11] used in the LagerIV [47] and CATHEDRAL [48] systems, and LUSTRE [49] have been developed to describe and synthesize regular DSP algorithms.

1.3 Exploring Alternative Designs

The systems mentioned above do not provide any estimate of the area occupied or the delays experienced by the circuits synthesized. It is only at the end of the synthesis process that these performance criteria are available. This means that designers have to wait until the end of the synthesis before recognizing the effect of their design decisions on the performance of the circuit being designed. Therefore, it becomes difficult for the designer to rapidly evaluate alternative design strategies for the particular application. It is possible to provide designers with estimates of various performance parameters at all stages of the design process and at all levels of abstraction. Obviously, at the higher levels of abstraction, these estimates will be less accurate because the exact details of the particular implementation strategy will not be known. However, these estimates are useful in order to provide the designer with enough information to make go/no-go decisions with the particular approach. For example, after describing the algorithm at a high level, the system may estimate that it will take two orders of magnitude more area than is available. Clearly, the designer can benefit from such information and can take early remedial action and try out other approaches.

This method of design allows the designer to perform rapid prototyping of alternative implementations at all levels of abstraction, leading to a design that is more suitable under the given design requirements. $\nu\mathcal{FP}$ provides such a system and evaluation mechanisms. Not only must there be a way to evaluate performance criteria at all levels of abstraction, but this evaluation must be fast and must not involve prohibitive overheads. If the evaluation takes too much time, users will tend not to use the features.

During the formulation of design alternatives, the designer has to be sure that each of the alternatives actually implements the same specification. One way of guaranteeing this is to take a transformational approach as in [50]. In this system, transformations are applied to the original specification to generate the various alternatives. These transformations are guaranteed to maintain the semantics of the specification. Chapter 4 demonstrates how this is approach is used in $\nu\mathcal{FP}$ to evaluate alternate designs.

1.4 Reasoning about Designs

Most of the design systems available today have representations of the design that are not amenable to easy verification. The problem consists of proving that two descriptions are equivalent in some sense. Usually the two descriptions are at different levels of abstraction. One is a description of a specification and one is the description of a particular implementation of that specification. Often the criterion for equivalence is that the input-output behavior

of the two descriptions is the same. That is, for the same inputs, both the descriptions will produce the same outputs. Sometimes the descriptions may be at the same level of abstraction but may represent different views of the same design. In this case, the equivalence criterion is usually to show that one aspect of the descriptions (say, the control flow) is the same in both descriptions. Chapter 5 shows how $\nu\mathcal{FP}$ transformations can be used to prove equivalences between different views during the design process.

In order to be able to prove such equivalences, the semantics of the descriptions must be provided. These semantics can then be reasoned about. There are two ways of conducting this reasoning. One way is to use the semantics of the descriptions to obtain the denotation of each. Formal methods can then be used to show that the two denotations are equivalent. This is the approach taken in [37, 17, 23]. The other approach is to use transformations to transform one description to another. If these transformations have previously been shown to maintain the equivalence property, whatever it may be, then the two descriptions can be shown to be equivalent. ITL, μFP, and $\nu\mathcal{FP}$ use this approach. The Yorktown Silicon Compiler [51] also makes use of transformations at specific stages in the silicon compilation process. The advantage of using this method is that the proof is carried out in the same domain as the descriptions and there is no need to involve another domain (that of denotations) in order to prove properties about the descriptions. The disadvantage is that the kind of properties that can be proved are restricted to the set of transformations provided. This can be alleviated by admitting new transformations into the system, but if these new transformations cannot be deduced from already existing ones, then it will usually be necessary to prove the new transformations by appealing to another domain. Chapter 6 gives various examples of transformations used for synthesis in $\nu\mathcal{FP}$. As an extended example of the use of such transformations, section 7.6 demonstrates the use of transformations to derive the proof of a theorem about combinations of sequential machines.

1.5 Specifications

In order to prove equivalence between a specification and implementation, it is necessary to construct a specification first. This specification is written in a specification language. An implementation at one level can be thought of as a specification for the next lower level. It is therefore necessary to have specifications at all levels of abstraction. Any specification should be minimal. That is, it should specify only those properties that are relevant at that level and no more. At the highest (algorithmic) level of specification, it is particularly important that the minimum amount of sequencing be specified. This implies that there are no extra sequencing constraints in the specification other than those necessary for the proper operation of the

system. An applicative language like $\nu\mathcal{FP}$ or μFP fits these constraints well because the only sequencing that is specified is that the input data must be available before the computation can start. The Church-Rosser property of applicative languages [52, 53] allows even more flexibility in sequencing. The Church-Rosser property states that the order of evaluation of its sub-expressions does not change the meaning of the whole expression. Therefore, the order of evaluation of sub-expressions is not constrained.

Structural specifications consist of specifying modules that encapsulate behavior. Behavior is only visible to the outside world via ports. Modules may be combined by connecting their ports together. In order to facilitate this composition, it would be useful if the ports could be typed or otherwise possess attributes. With that information, it is possible for a composition system to check that the ports being connected are compatible. The simplest example of such checking is the type-checking performed between a called function and the arguments passed to it. In order to be flexible, typing and checking should be polymorphic. Languages like SML [54] provide such static type-checking. $\nu\mathcal{FP}$ currently uses run-time type-checking and so type errors are only detected at run-time. However, this is not inherent in the language and the addition of a type system is suggested as future work.

At lower levels of abstraction the spatial positioning between components is important. It can be specified in different ways. One is by just specifying topology. That is, which ports on certain modules are connected to which ports on other modules. In addition to this connectivity information, some relative placement information can also be specified. This consists of specifying which modules are on which side of a certain module. Further positioning information can be specified by specifying geometry. That is, actual locations, either absolute or relative to some other point, are specified. The $\nu\mathcal{FP}$ system is capable of specifying and displaying relative placement information. Schlag, in [55], shows how this information is extracted from the specification. Chapter 6 shows the specification of structure and behavior in $\nu\mathcal{FP}$. Chapter 3 explores the use of applicative languages, and $\nu\mathcal{FP}$ in particular, in the specification of algorithms at the highest levels.

1.6 Visual Feedback

VLSI design is a graphical process—especially at the lower levels which deal with the geometry of the layers in silicon. Most of the lower level layout tools are visual, but higher level tools like silicon compilers are often textual. Intermediate tools like floor planners tend to be graphical or both graphical and textual. What is needed is to have visual feedback at all levels of abstraction, for the same reasons as it is good to have performance feedback at all levels of abstraction. For instance, suppose there is a description of the algorithm at a high level that

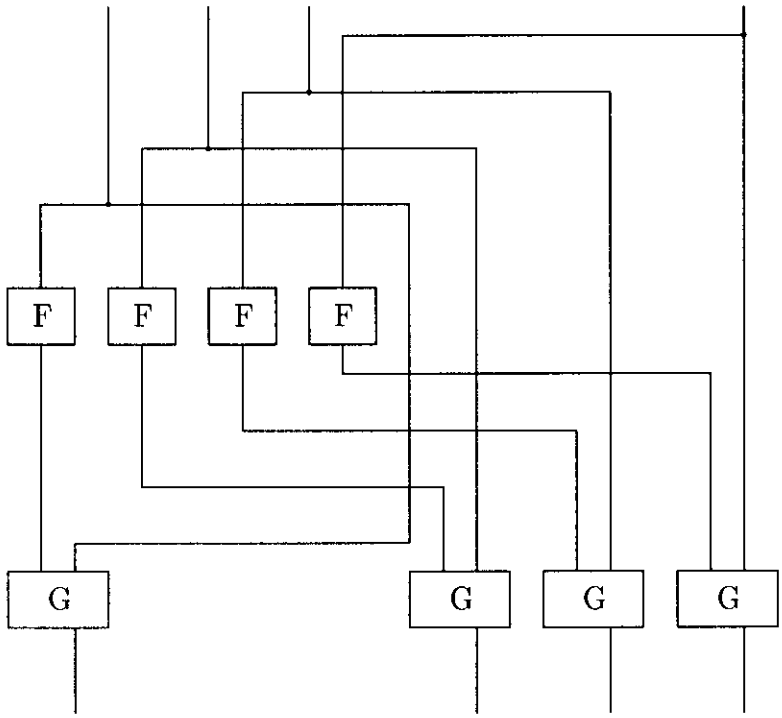


Figure 1.11: Blocks with routing outside

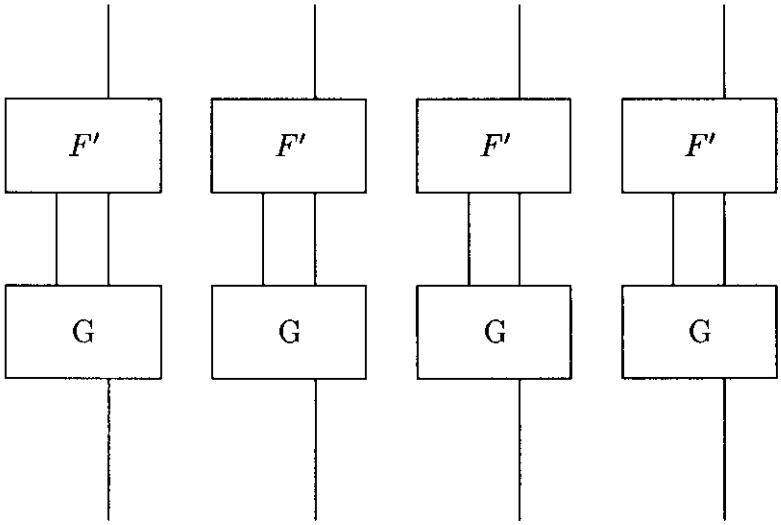


Figure 1.12: Blocks with route-throughs

implied a lot of routing around a block to get to another block. If designers could see the layout or routing implications of their high level decisions, they would be able to modify their approach to take this into consideration. For example, in the previous case, the designer could consider using cells that routed the extra wires through the cells of the first block rather than around it. Figure 1.11 shows how the layout would look like with the routing around the cell. Figure 1.12 shows that the routing area has been saved by having the wires pass through the cell F'. F' is a cell that is the same as F except that it additionally routes its input to another output. A simple transformation is used to prove the equivalence of the two representations.

A graphical language is very useful at the geometry and layout planning levels of the design since most of the information at those levels is pictorial in nature. Hence there is a better match between the objects used for description and the objects being designed—the semantic gap is small. These languages also provide a better human-machine interface at these levels. Textual languages are better at specifying algorithmic behavior and therefore more useful at higher abstraction levels. They also provide an easier interface between different tools since the interface can be plain text.

Both kinds of languages also have their deficiencies. Visual languages are limited in what they can easily specify and are often forced to revert to a textual mode to specify complex functions. Since visual systems are required to show the current picture, structuring or abstraction information is either not allowed to be specified or it is hidden from the user. Often different

parts of a picture are supposed to bear a spatial relationship to each other. However, unless this relationship is explicitly specified (usually textually) somewhere, as parts of the picture are moved, these relationships might change. Pictures specified via exact or computed locations are difficult to specify in a visual editor. Though iteration can be represented (textually) via ellipses; recursion, loops and conditionals are difficult to represent in a visual framework.

On the other hand, it is often difficult to visualize the graphical structure implied by a textual description. This necessitates a tedious run-display-edit cycle to get the required layout. Another problem is that exact positional values are always required, even when the user does not care about the exact placement. Positioning visual elements to provide an aesthetically pleasing picture are difficult to produce via textual languages.

The solution to this dichotomy, of course, is to provide a system that integrates both textual and graphical modes for manipulating and describing the design. Much work has been carried out in this area. The Tpack module generation system [56] takes the approach of using graphical editors to create small pieces of geometry called *tiles* and using a procedural language to lay out and connect these tiles. DPL [57] and Sam [58] show the user both the graphical and textual view of the design being constructed and allow the independent editing of each. If the graphical view is edited, the textual view changes appropriately and vice versa. Tweedle [59] deals with general graphics and takes a similar approach but extends it by allowing the textual language to be procedural and incremental. A less desirable though still useful approach is taken by Henderson's functional geometry system [36] in which the graphical display is generated from a textual description. Any editing has to be carried out on the textual description. The graphics cannot be directly edited. This is essentially the same approach taken in the μFP and $\nu\mathcal{FP}$ systems. All of the layouts in this dissertation are generated directly from $\nu\mathcal{FP}$ descriptions.

1.7 Overview

This research describes a method based on applicative languages [60] for the specification, evaluation and synthesis of hardware algorithms. Though this dissertation explores the role of applicative languages can play throughout the design spectrum—from specification down to implementation—it mainly concentrates on the synthesis of space and time domain hardware implementations of algorithms. This method is supported by a set of tools that is being developed at the University of California at Los Angeles. The goal of this effort is to provide designers with an environment in which they can rapidly explore various alternative designs for their algorithms. Thus, it is possible to specify the algorithm at any arbitrary level of abstraction and have the system rapidly evaluate performance parameters (e.g., speed, area) so that designers can make informed decisions during the synthesis process. The advantage

of using an applicative language is that it ties together the specification of the algorithm, the synthesis of the circuit and the evaluation of the implementation.

$\nu\mathcal{FP}$, the system described here, is based on using a strict[4], applicative-order[5] functional language to represent combinational and sequential behavior. Others have explored incorporating applicative languages into VLSI design and have shown them to be viable. Lahti [61] used an applicative language to describe various combinational hardware structures. Johnson [62] utilized a non-strict, normal-order[6] applicative language to describe and synthesize sequential digital circuits. Cardelli and Plotkin [63] take a formal approach to describing sequential circuits with an emphasis on verification. Meshkinpour [64] and Sheeran [65] extended Backus' FP language with operators to handle sequential circuits. ELLA [12], SILAGE [11], and LUSTRE [49] are applicative design languages which have been used to describe sequential hardware structures.

$\nu\mathcal{FP}$ uses a transformational approach to synthesis. A representation at one level of abstraction is converted into a representation at a lower level of abstraction using pre-defined substitutions[7]. These substitutions have been previously proved to preserve the meaning of the representation. Hence the final implementation is guaranteed to behave the same as the specification. Others have used a transformational approach in the design of VLSI systolic arrays [66, 67]. In these approaches delays are added and removed according to transformation rules in such a way that the output function remains invariant. Transformation approaches have to be supplied with an initial solution.

There are other methods [68] for the synthesis of circuits. Most use some form of a labeled graph with nodes as cells or functions, and edges as communication lines or precedence relations [69, 70, 71, 51]. This graph is then mapped into an implementation by scheduling and allocation. Allocation is mapping the data flow graph into hardware modules and data paths between these modules [72, 73]. Scheduling is the mapping of the control nodes to particular instances of time. Formal approaches of synthesizing behavioral descriptions into implementations in space-time are shown in [74, 75].

This dissertation highlights the impact of applicative languages in the design process from specification to implementation. Chapter 2 provides an introduction to the $\nu\mathcal{FP}$ system and some of its features. $\nu\mathcal{FP}$ is a strict, applicative-order functional language. The role of applicative languages in the specification of algorithms is explored in the next chapter. Chapter 4

[4]A function is said to be strict if and only if its value is undefined when any of its arguments is undefined.

[5]In applicative-order evaluation the arguments to a function are evaluated before the function application is performed.

[6]In normal-order evaluation, the leftmost outermost reducible expression is evaluated first. Hence functional applications are carried out before the arguments to the function are evaluated.

[7]Transformations that replace a function name by its definition will be called substitutions.

discusses how applicative languages (and the $\nu\mathcal{FP}$ system in particular) are helpful in evaluating various alternative designs that may be constructed from the same specification. In order to explore the effects of using an applicative language in abstraction and refinement during design, $\nu\mathcal{FP}$ was proposed as the interpretation domain language for the Graph Model of Behavior (GMB) in the SARA system. Chapter 5 describes how $\nu\mathcal{FP}$ allows provable refinements in the GMB and how it can be used to prove equivalences between different views of the same representation.

The transformation of an algorithm in the abstract to an implementation in space-time is described in chapter 6. The $\nu\mathcal{FP}$ synthesis process proceeds as follows. First the specification is prepared. Then various high-level designs are carried out and evaluated with respect to performance criteria. The design process continues using refinement until primitives are reached. The design is evaluated at each stage to make sure it will meet the goals. At this point, the layout of the circuit can begin. The first step is to lay out the algorithm in space (section 6.3). If this meets the design goals, the synthesis is over. If this approach takes too much space, time synthesis may be attempted. This is covered in section 6.4. When the synthesis is complete, the internal representation is converted into a form acceptable to lower level layout tools for final fabrication. This is the subject of section 6.6. Sheeran's μFP [65] is the system that is closest to the one proposed in this dissertation. Chapter 7 discusses the differences between the two systems and points out the tradeoffs involved. The dissertation concludes with a description of the contributions of this work and suggestions for future explorations. Appendix A provides a complete and precise definition of the $\nu\mathcal{FP}$ language. Appendix B clarifies some aspects of GMB semantics.

Chapter 2

The $\nu\mathcal{FP}$ System

This chapter gives an overview of the $\nu\mathcal{FP}$ language and the $\nu\mathcal{FP}$ system that includes language processors and evaluators.

2.1 Brief Introduction to $\nu\mathcal{FP}$

$\nu\mathcal{FP}$ extends the language FP [60] proposed by Backus with additional functional forms and primitives. In contrast to μFP [65], which extends FP's semantics to operate on streams, the semantics of $\nu\mathcal{FP}$ are the same as those of FP when it is used to specify algorithms. A program in $\nu\mathcal{FP}$ (as in FP) is an expression, corresponding to a function, that maps objects into objects. Objects are either atomic (numbers or strings) or sequences of objects. The distinguished atom $\perp$ denotes an undefined value. By definition, any sequence which contains $\perp$ as an element is itself undefined and thus equal to $\perp$. This is another way of saying that $\nu\mathcal{FP}$, like FP, has strict semantics. This means that if any argument to a function is undefined, the result of evaluating the function is also undefined. Formally,

$$\forall f, f : \perp \equiv \perp. \tag{2.1}$$

The strictness property allows functions to be defined as total functions over all inputs and implies that $\perp$ is in the domain of every function. Other applicative languages have non-strict (sometimes also called *lazy*) semantics. There are advantages and disadvantages of each approach. Having strict semantics is particularly helpful in a distributed environment because if an error develops in any particular part of the program, it is guaranteed that the error will propagate everywhere and eventually the whole program will terminate. On the other hand,

arithmetic functions

$$+ : \langle 1, 5 \rangle \rightarrow 6$$
$$* : \langle 3, 2 \rangle \rightarrow 6$$

logical functions

$$\texttt{andg} : \langle 1, 0 \rangle \rightarrow 0$$
$$\texttt{nandg} : \langle 1, 0 \rangle \rightarrow 1$$
$$\texttt{org} : \langle 0, 0 \rangle \rightarrow 0$$
$$\texttt{xorg} : \langle 1, 1 \rangle \rightarrow 0$$

predicates

$$\texttt{atom} : \langle 1, 2 \rangle \rightarrow \texttt{false}$$
$$=: \langle 3, 3 \rangle \rightarrow \texttt{true}$$

selector functions

$$3 : \langle 2, \langle 4, 5 \rangle, 6, \langle 8, \langle 9, 10 \rangle \rangle \rangle \rightarrow 6$$
$$\texttt{last} : \langle 1, 4, 6 \rangle \rightarrow 6$$

Table 2.1: Primitive Functions

strict semantics prohibit the use of incomplete (and possibly infinite) data structures which can be useful in some computations.

The primitive functions of $\nu\mathcal{FP}$ consist of arithmetic, logical, predicate and selector functions, examples of which are shown in table 2.1. The cases involving an argument of $\perp$ (equation 2.1) are omitted from the definitions for brevity. Examples of primitive structure modifying functions are shown in table 2.2. Functional forms are used to combine primitive functions into more complex functions. Some example functional forms are shown in table 2.3. A complete description of the $\nu\mathcal{FP}$ primitives and forms is found in appendix A.

A major syntactic difference between $\nu\mathcal{FP}$ and Backus' FP is that parameters to functions may be named and then referred to in the function body with restrictions similar to those specified in Backus' extended definitions [76] for FP. As with Backus' extended definitions, it is understood that the parameter names do not refer to input *objects*, but rather to the *functions* that create the particular input objects.

$$
\begin{array}{ll}
\texttt{trans} : \langle\langle 1,2,3\rangle,\langle 4,5,6\rangle\rangle & \rightarrow \langle\langle 1,4\rangle,\langle 2,5\rangle,\langle 3,6\rangle\rangle \\
\texttt{apndl} : \langle 1,\langle 2,3,4\rangle\rangle & \rightarrow \langle 1,2,3,4\rangle \\
\texttt{apndr} : \langle\langle 1,2,3\rangle,4\rangle & \rightarrow \langle 1,2,3,4\rangle \\
\texttt{distl} : \langle x,\langle a,b,c\rangle\rangle & \rightarrow \langle\langle x,a\rangle,\langle x,b\rangle,\langle x,c\rangle\rangle \\
\texttt{distr} : \langle\langle a,b,c\rangle,x\rangle & \rightarrow \langle\langle a,x\rangle,\langle b,x\rangle,\langle c,x\rangle\rangle \\
\texttt{iota} : n & \rightarrow \langle 1,2,\ldots,n\rangle \\
\texttt{pick} : \langle k,\langle x_1,\ldots,x_n\rangle\rangle & \rightarrow x_k \text{ if } k \leq n \\
\texttt{null} : x & \rightarrow (x = \langle\rangle \Longrightarrow \texttt{true};\texttt{false})
\end{array}
$$

Table 2.2: Structural Primitives

compose

$$(f \circ g) : x \rightarrow f : (g : x)$$

construct

$$[f,g,h] : x \rightarrow \langle f : x, g : x, h : x\rangle$$

applytoall

$$\&f : \langle p,q,r\rangle \rightarrow \langle f : p, f : q, f : r\rangle$$

constant

$$\%k : x \rightarrow k \text{ if } x \text{ is not } \perp$$

rightinsert

$$!f : \langle x_1,\ldots,x_n\rangle \rightarrow f : \langle x_1, !f : \langle x_2,\ldots,x_n\rangle\rangle$$

treeinsert

$$\clubsuit f : \langle x_1,\ldots,x_n\rangle \rightarrow f : \langle \clubsuit f : \langle x_1,\ldots,x_{\lceil n/2\rceil}\rangle, \clubsuit f : \langle x_{\lceil n/2\rceil+1},\ldots,x_n\rangle\rangle$$

sequential

$$\textbf{seq}\, f : \langle x_1,\ldots,x_n\rangle \rightarrow \texttt{apndr} \circ [\textbf{seq}\, f : \langle x_1,\ldots,x_{n-2}, 1 \circ y\rangle, 2 \circ y] \text{ where } y = f : \langle x_{n-1}, x_n\rangle$$

map

$$\{f_1, f_2,\ldots,f_n\} : \langle x_1,\ldots,x_n\rangle \rightarrow \langle f_1 : x_1, f_2 : x_2,\ldots,f_n : x_n\rangle$$

Table 2.3: Functional Forms

A $\nu\mathcal{FP}$ definition like

```
defun f ( x , y )
      ...
enddef
```

is exactly equivalent to the FP extended definition

```
xdef f ∘ [ x , y ] = ...
```

There are two restrictions on which functions can be defined with parameters. The first is that, as in Backus' extended definition, the body of the function must be *distributive* with respect to the parameters. Formally, a $\nu\mathcal{FP}$ expression E is distributive with respect to the variables $v_1, v_2, \ldots, v_n$ if and only if

$$E(v_1, v_2, \ldots, v_n) \circ h \equiv E(v_1 \circ h, v_2 \circ h, \ldots, v_n \circ h)$$

For example,

```
defun f ( x , y )
      [ q ∘ x , r ∘ y ]
enddef
```

is distributive, but

```
defun g ( x , y ) [ p , q ∘ x , r ∘ y ] enddef
```

and

```
defun h ( x ) & x enddef
```

are not. The second restriction, which is not present in Backus' extended definition, is that the parameter names must be simple and distinct. That is, a function like

```
defun k ( x , x )
      + ∘ [ x , % 3 ]
enddef
```

is not allowed. In other words, unlike Backus' extended definitions, there is no pattern matching allowed in the parameters. Under these restrictions, it is very easy to convert a definition with parameters into one without parameters. The conversion process consists of replacing each occurrence of a parameter in the body by the corresponding selector function. For example, function f above could be converted to

```
FullAdder =
    [org ∘ [org ∘ [andg ∘ [1,2],andg ∘ [2,3]],andg ∘ [1,3]],
     xorg ∘ [1,xorg ∘ [2,3]]]
```

Figure 2.1: A Full Adder in Backus' FP

```
defun FullAdder(a,b,Cin)
      [((a andg b) org (b andg Cin)) org (a andg Cin),
      a xorg (b xorg Cin)]
enddef
```

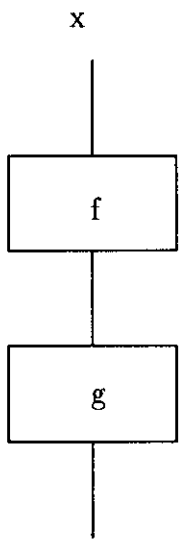
Figure 2.2: A Full Adder in $\nu\mathcal{FP}$

```
defun f
      [ q ∘ 1 , r ∘ 2 ]
enddef.
```

The restrictions were chosen to simplify the implementation of the conversion process. There is no reason (other than time) that the full Backus extended definition could not be implemented. If the function being defined does not meet the restrictions, or if the function takes a variable number of arguments, no parameters are specified in its definition. The absence of parameters indicates that the "normal" FP definition should be used.

In addition to specifying parameters, the arithmetic, logical, and predicate functions may be used either in a prefix or an infix manner. This improves the readability of hardware specifications. For example, the definition of a *FullAdder* in Backus' FP (figure 2.1) could alternatively be written in $\nu\mathcal{FP}$ as shown in figure 2.2.

In $\nu\mathcal{FP}$, unless sequentialized via the use of the composition operator, all functions execute in parallel and hence $\nu\mathcal{FP}$ descriptions are suited to describing parallel hardware algorithms. These specifications are executable and hence can be tested by applying the function to an argument and checking to see whether the expected result is obtained. Specifications can also be executed symbolically, at which time it is possible to extract the topological structure of the algorithm. This is done by using a symbolic interpreter instead of the normal value interpreter and using symbolic inputs rather than value inputs. There is a direct relationship between the structure of an algorithm written in $\nu\mathcal{FP}$ and the planar topology of its layout. For example,



g: (f:x)

Figure 2.3: Composition of two functions

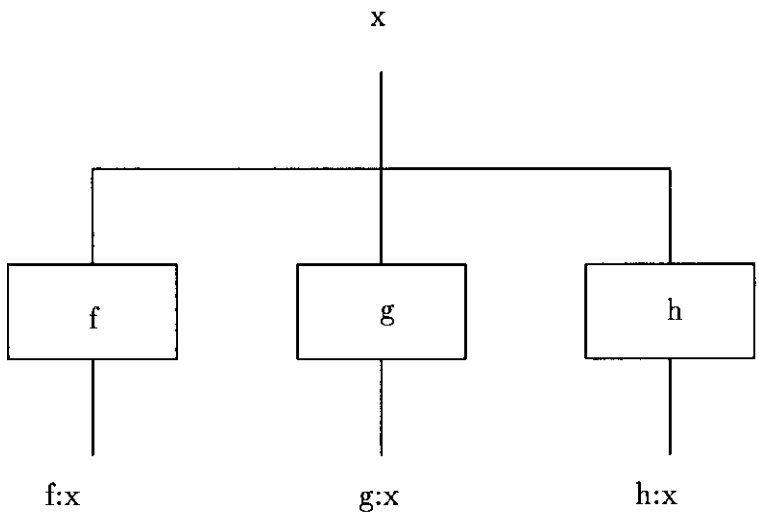


Figure 2.4: Construction of three functions

figure 2.3 shows what the composition of two functions would look like topologically. Figure 2.4 shows what the topology of a construction of three functions looks like. It is assumed that the inputs are at the top and the outputs at the bottom—data flows down the page.

2.2 Value and Symbolic Evaluation

There are two modes in which a $\nu\mathcal{FP}$ program can be evaluated. The first is the "normal" mode in which the program is applied to an argument and returns a value as its result. This is called *value evaluation.* The second mode is called *symbolic evaluation.* In this mode, the program is called with symbolic arguments instead of actual arguments and the result is a symbolic value rather than an actual value. Another result of symbolically evaluating a $\nu\mathcal{FP}$ expression is a computation flow graph. The edges of this graph are labeled with symbolic values, and the nodes are labeled with the name of the function being applied.

As mentioned in chapter 4, any user-defined function can be *tagged* as primitive. As far as the computation flow graph is concerned, the inner structure of the computation inside such a node will not be kept. However, the system will symbolically evaluate the inner functions in order to generate the correct symbolic output for that function.

Such a simple-minded generation of the computation flow graph will lead to many edges that are unused. For example, a complete value may be generated, but only the first or last element of the resulting sequence may be used in further computation. In order to remove such unused edges, the initial computation flow graph is pruned by tracing back from the leaves of the graph and recursively removing nodes and edges that are unused.

This is further explained in section 6.6 and complete details may be found in [55].

2.3 Attributes

As will be explained further in chapter 4, during symbolic evaluation, the system will keep track of which tagged functions are executed at which point in the computation flow graph. The overhead associated with collecting this information is very low. These statistics can then be used to evaluate different algorithms and compare them with respect to whatever performance characteristics the designer desires. Currently, the system only keeps track of information relating to the algorithm itself. The location of the tagged functions in the computation graph gives an idea of where the parallelism lies in the algorithm. It is also useful to pick out algorithmic bottlenecks. The depth of the computation flow graph gives an estimate of the time required by the algorithm. The number and type of functions executed can be used to

get some idea of the space that might be occupied by an implementation of the algorithm. The important point to notice is that these attributes are evaluated at any arbitrary level of abstraction.

One of the benefits of using an applicative framework is the ease with which attribute evaluation can be implemented in it. Consider the simple evaluation of a function.

$$y = f : x$$

Here a function f is applied to an argument x and returns a result y. Instead of just operating on a simple value, we might consider that the argument x has attributes $a^1, a^2, \ldots a^n$. This would result in

$$\langle y, y_{a^1}, y_{a^2}, \ldots, y_{a^n} \rangle = \langle f : x, f_{a^1} : x_{a^1}, f_{a^2} : x_{a^2}, \ldots, f_{a^n} : x_{a^n} \rangle$$

where, for each f, f_{a^i} denotes the function that, given the attribute x_{a^i}, computes the result attribute y_{a^i}. Note that this can be introduced into a functional evaluation mechanism by replacing f by

$$\{f, f_{a^1}, f_{a^2}, \ldots, f_{a^n}\}$$

where $\{f_1, \ldots, f_n\}$ is the *map* functional form. A generic, user-definable attribute system based on attribute grammars was partially developed during this research, but never completed. It is suggested as future work.

Chapter 3

Applicative Languages in the Specification of Algorithms

Specification languages are used to specify what a particular algorithm does without specifying how that is to be achieved. Applicative languages have many properties that make them attractive as languages for specifying hardware or software algorithms. This chapter begins with a discussion of features that are desirable in a specification language and shows how properties of applicative languages fulfill these requirements. Some problems with using applicative languages for specification are discussed in conclusion.

3.1 Properties of Specification Languages

Specification languages are used at the highest level of abstraction to specify the essence of the algorithm without unnecessarily constraining the implementation. This section describes features that are desirable in a specification language.

Formality: It is most important that the specification language be a formal language so that there is no ambiguity in the meaning of a specification written in the language. In addition, if the language has a formal semantics, it is possible to devise programs that could check a specification for consistency.

Constructibility: It always takes a lot of effort to convert a designer's ideas into a formal specification. The easier this task is made by the specification language, the better the chance that the specification will match the intent of the designer.

Comprehensibility: Just as the language should make it easy to go from intent to specification, it should also make it easy for a reader to go from the specification to the concept. In this context, the size and lucidity of the language are major factors.

Minimality: It is important that the language be able to express all the relevant properties of the concept. It is also important that no extra (overhead) properties should be needed to specify the concept. This is not always possible, but the closer a language can get to this goal, the better. The smaller the number of concepts that are required to specify an algorithm, the better. It should be mentioned that it is important to reduce the number of lexical tokens in a specification and not just to have fewer symbols. Just reducing the number of symbols may make a specification look smaller, but the cognitive load for the reader to understand the specification will be higher.

Wide Range of Applicability: The greater the range of domains the language can be applied to, the better.

Extensibility: It will not be possible to come up with a language that covers a wide variety of domains because of differing primitives and needs. One way of alleviating the situation is for the language to provide a basic framework and to allow domain-specific extensions.

Executability: If a specification is executable and/or analyzable by machine, it makes it easier for the specifiers to check the specification and make sure that it is indeed specifying what they thought it was.

3.2 Properties of Applicative Languages

Applicative languages possess many properties that make them suitable for use as specification languages. The most important is the fact that these languages do not have any side-effects. Since there are no side-effects, it has been shown that these languages (in particular, FP [53]) possess the Church-Rosser property [77]. This means that the result of a computation, or the meaning of a computation is unaltered by the order of evaluation of its sub-expressions. Since the order of evaluation is not constrained, this implies that only the necessary (i.e., minimum) sequencing information for the algorithm is used in an applicative language. In turn, this implies that the maximum parallelism inherent in the algorithm can be expressed. The sequencing that is necessary is only that required by functional dependencies and no extra control dependencies need to be introduced as would be necessary in an imperative language. Many imperative languages are unsuitable for specifying loops with independent iterations because they overly specify the sequencing between iterations.

A related property of applicative languages is that they are referentially transparent. That is, the result of a computation is independent of the context of the computation. Because of this, the meaning of an expression is discernible just by looking at the lexical scope of the expression. There is no way the evaluation of some other expression in some other location can affect the result of evaluating this expression. This makes the specification comprehensible. In contrast, imperative languages often have constructs like pointers and non-local variables which makes them difficult to analyze lexically.

Of course, the specification is executable, thus making it easier for designers to confirm that the specification faithfully implements their idea.

Applicative languages, and FP in particular, usually are small in size and have simple and concise semantics that are easy to understand. In addition to aiding the lucidity of the specification, the formal semantics and allows the easy construction of theorems to prove properties about specifications written in these languages. The algebraic foundations of the language allow one to write transforms that can be used to prove properties about programs without leaving the domain of programs. These transforms can also be used for semi-automatic synthesis of implementations from their specifications.

The non-procedural nature of these languages makes it easy to describe a specification that is orthogonal to the implementation.

Applicative languages, and FP in particular, provide a framework of composition forms that can be used to compose primitive functions to build other functions. The base set of primitive functions can easily be tailored to suit the application without changing the framework. This makes it easy to adapt the system to different problem domains. User-defined functions are treated semantically the same as primitive functions and so it is easy for the user to extend the set of primitives to better match the problem at hand.

3.3 Problems with Applicative Languages

There are some features of applicative languages that may hinder their use as specification languages. The major problem is the lack of history-sensitivity. This is not usually a problem with algorithms because most do not inherently involve any state information at the highest specification level. However, when it is necessary to describe larger systems, or systems in which state plays an important role, this shortcoming becomes burdensome. Since there is no way to specify time explicitly, this formalism cannot describe real-time or other systems requiring timing constraints.

An alternative is to embed a purely applicative system within a system with state. The computation is performed by the applicative part and the state is used only to provide history

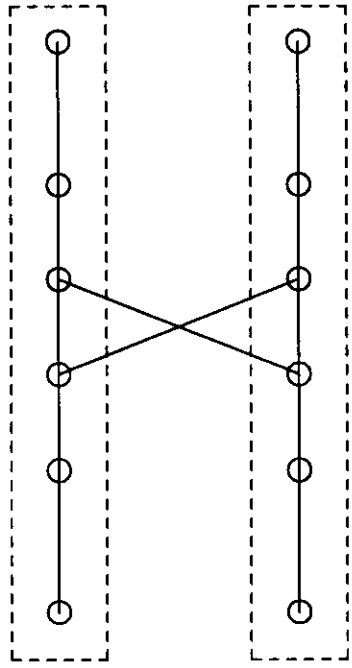


Figure 3.1: One-dimensional partition

sensitivity. The state-transitions of such a system are driven by the applicative sub-system. Backus [60] refers to such a system as an Applicative State Transition System. A successor to FP, FL [78], uses a similar scheme to provide input/output from within an applicative framework.

In other specification languages, programs can be composed sequentially, in one dimension. $\nu\mathcal{FP}$, being a parallel language, allows composition to be performed both sequentially and concurrently, in two dimensions. This makes it possible to encapsulate communication between program threads into another program and thus abstract it out. Another benefit of this approach is that non-planar communication, routing and connections can be hidden within a function. This encapsulation can be used to make a non-planar computation-flow graph planar. Schlag [55] uses this property to generate planar topology from $\nu\mathcal{FP}$ expressions. The problem is that $\nu\mathcal{FP}$ *forces* the designer to use this two-dimensional composition even when it might not be the most appropriate thing to do. Figure 3.1 shows the normal one-dimensional partitioning of the two communicating processes. Figure 3.2 shows the corresponding partition using $\nu\mathcal{FP}$ which encapsulates the communication between the processes inside one of the partitions. A related problem is the inability of $\nu\mathcal{FP}$ to describe processes that have true bi-directional data flow (e.g., systolic arrays with data flow in more than one direction).

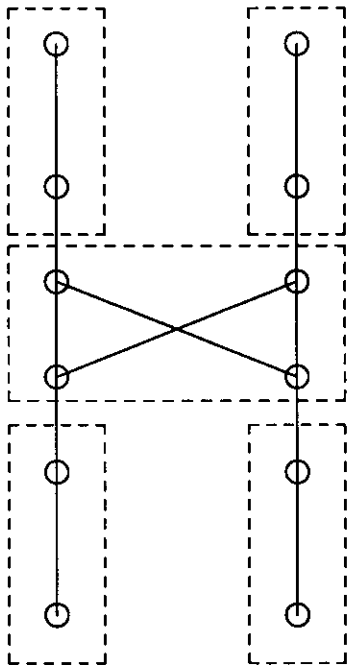


Figure 3.2: Two-dimensional partition

3.4 Summary

Applicative languages have many features that make them attractive as high-level specification languages for algorithms. However, the lack of history sensitivity hampers their use as specification languages for real-time or state-oriented systems.

Chapter 4

Applicative Languages in the Performance Evaluation of Algorithms

Most design systems provide designers with performance measures of the synthesized system—but only upon completion of the synthesis. Hence it is difficult for designers to get feedback about the performance (speed and area) early enough in the design cycle. $\nu\mathcal{FP}$ provides designers with rapid feedback on the performance of their algorithm at all levels of abstraction. This feedback is provided with a very low overhead.

4.1 The Evaluation of $\nu\mathcal{FP}$ Algorithms

It is possible to *tag* selected user-defined functions so that when a $\nu\mathcal{FP}$ specification is executed an estimate of the performance of the algorithm can be provided. Tagging a function tells the system that this is a function of interest at the current level of abstraction. In $\nu\mathcal{FP}$ a function is tagged by turning on tracing for that function.

`)trace on` *functionName*

A tagged function is treated as a primitive by the system because the user has indicated that the internal behavior of the function is not of interest at this time. As the execution proceeds, the interpreter keeps track of the *level* at which a tagged function is executed.

The *level* of a tagged function is defined as one plus the maximum of the *levels* associated with the atoms in its input object. The *level* of each atom is initially zero. Each time a tagged function is encountered, its level is determined and is assigned to the atoms in its output object.

Let the level of an object or a function be denoted by a superscript. For example, let the level of a function f be l_f and be denoted as f^{l_f}. Let the result of applying the function f to the argument $\langle x_1, x_2, \ldots x_n \rangle$ be y. Then

$$f^{l_f} : \langle x_1^{l_1}, x_2^{l_2}, \ldots x_n^{l_n} \rangle \Longrightarrow y^{l_y}$$

and

$$l_f \equiv l_y \equiv \max(l_1, l_2, \ldots l_n) + 1$$

If the output of f is a sequence, each element of the sequence is tagged with the same level l_y.

A similar formalism is used to represent attributes other than the level except that the equation for computing the resultant attribute is different for each of the other attributes. For example, in the case of area, the input areas are summed to form the area of the resultant cell. In general, if each $\nu\mathcal{FP}$ value, x, has attributes $a_1, a_2, \ldots a_k$, the combination of the value and its associated attributes can be represented as the following sequence.

$$\langle x, x^{a_1}, x^{a_2}, \ldots, x^{a_k} \rangle$$

If there is a function f, such that

$$y = f : \langle x_1, x_2, \ldots, x_n \rangle$$

a new function, f^a, which uses attributed values can be described as

$$y^a = f^a : \langle \langle x_1, x_1^{a_1}, \ldots, x_1^{a_k} \rangle \ldots, \langle x_n, x_n^{a_1}, \ldots, x_n^{a_k} \rangle \rangle$$

where

$$f^a \equiv \{f, f^{a_1}, f^{a_2}, \ldots, f^{a_k}\} \circ \texttt{trans}$$

and each f^{a_i} is a function that knows how to calculate the attribute a_i of the result y^a given the input attributes $x_1^{a_i}, \ldots, x_n^{a_i}$.

The above formalism suggests that it is straightforward to modify the evaluation mechanism for $\nu\mathcal{FP}$ functions to incorporate the attributes into the framework. Moreover, this modification is done in a functional manner and does not entail any modifications to the rest of the system.

However, a slightly modified algorithm has to be used when a tagged function occurs within another tagged function. In this case the level of the inner function is determined with respect to the outer function resulting in a hierarchy of levels that can be represented as a Dewey-decimal number. This is accomplished by assigning the level zero to each atom of the outer function's input object, and computing the level of tagged functions as before, until the

computation of the outer function has been completed. The level of the outer function and the atoms in its output object is determined as before and hence is independent of whether or not any tagged function occurs within the outer function.

The implementation of this scheme is straightforward. The current level is stored in a global variable. Each tagged function increments this variable just before exiting. Another global indicates whether a tagged function is currently active. If a new tagged function begins execution while this variable is `true`, the old value of the level is pushed on to a stack and the level is reset to `0`. Most functional forms do not change the value of the level. However, the *apply-to-all*, *map*, *tree-insert* and *construct* forms save and restore the value of the level variable in between invocations of their functional arguments.

For example, during the execution of

$$[\ f,\ g\ ]$$

the current level is first saved. The function f is executed. During the execution of this function, the level may be incremented depending on whether any tagged functions are evaluated. The resulting level is saved in a temporary location and the original level is restored. The function g is then evaluated. The final level is the maximum of the level after the execution of f and g. The *apply-to-all*, *map*, and *tree-insert* use similar strategies. An earlier implementation of this scheme is reported in [79] and an alternative implementation is reported in [80].

At the end of execution, the system will have gathered statistics about each tagged function. The level of each instance of each tagged function is reported. Assume an ideal situation, in which infinite resources (processing agents, communication paths and memory) are available and that each tagged function has a unit delay. In this situation, the level of a function instance represents the point in time this function instance would be executed in an optimal schedule. Thus, under the above idealistic assumptions, the maximum level in the circuit gives the latency of the circuit in terms of unit delay. This can be used to predict the speed at which the circuit would perform. The system also keeps track of the number of function instances at each level. This information can be used to obtain an idea of where the parallelism in the algorithm is.

The total number of tagged function instances can be accumulated to provide an estimate of the area that would be occupied by the circuit. It should be noted that this area estimate only contains the sum of the areas of the cells. It does not incorporate any additional routing area that may be required. In addition, at a high level of abstraction, the estimated area is determined from an implementation that is isomorphic to the behavior because the eventual structure is not known at this point. Hence the estimate will only provide a rough measure until the structure is further defined. A better estimate of the area is obtained by methods mentioned in chapter 6.

4.2 Time-Space Tradeoffs

This capability of having the system estimate performance parameters is useful in tradeoff analyses. For example, consider the following function:

$$z = \begin{cases} 1 & \text{if } a = (b-1) \bmod 8 \\ 2 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$

schemeA and *schemeB*, below, are two algorithms for implementing the function. It is assumed that `a` and `b` are three-bit unsigned integers and that `z` is a two-bit unsigned integer. From the specification it can be seen that z_0 is `1` when $a = (b-1) \bmod 8$ and z_1 is `1` when $a = b$.

schemeA checks the equality of two numbers by first sending each number to a decoder. Only one output of each decoder will be a `1`, corresponding the the value of the input. Thus, to check for equality, the corresponding outputs of both decoders must be a `1`. The "$(b-1) \bmod 8$" is implemented by rotating the output of the decoder before feeding it to the AND gate tree. Figure 4.1 shows a layout of *schemeA*.

schemeB takes a straightforward approach to implementing the specification. Equality is determined by the use of comparators. The "$(b-1) \bmod 8$" is implemented by using a two's complement adder, adding `7` to the value of `b` and ignoring the carry out from the adder. Figure 4.2 shows a layout of *schemeB*.

Transformations can be used to prove that both *schemeA* and *schemeB* implement the same specified function. If the boolean functions (`andg`, `org`, `notg`, and `xorg`) are tagged, the results shown in tables 4.1 and 4.2 are obtained.

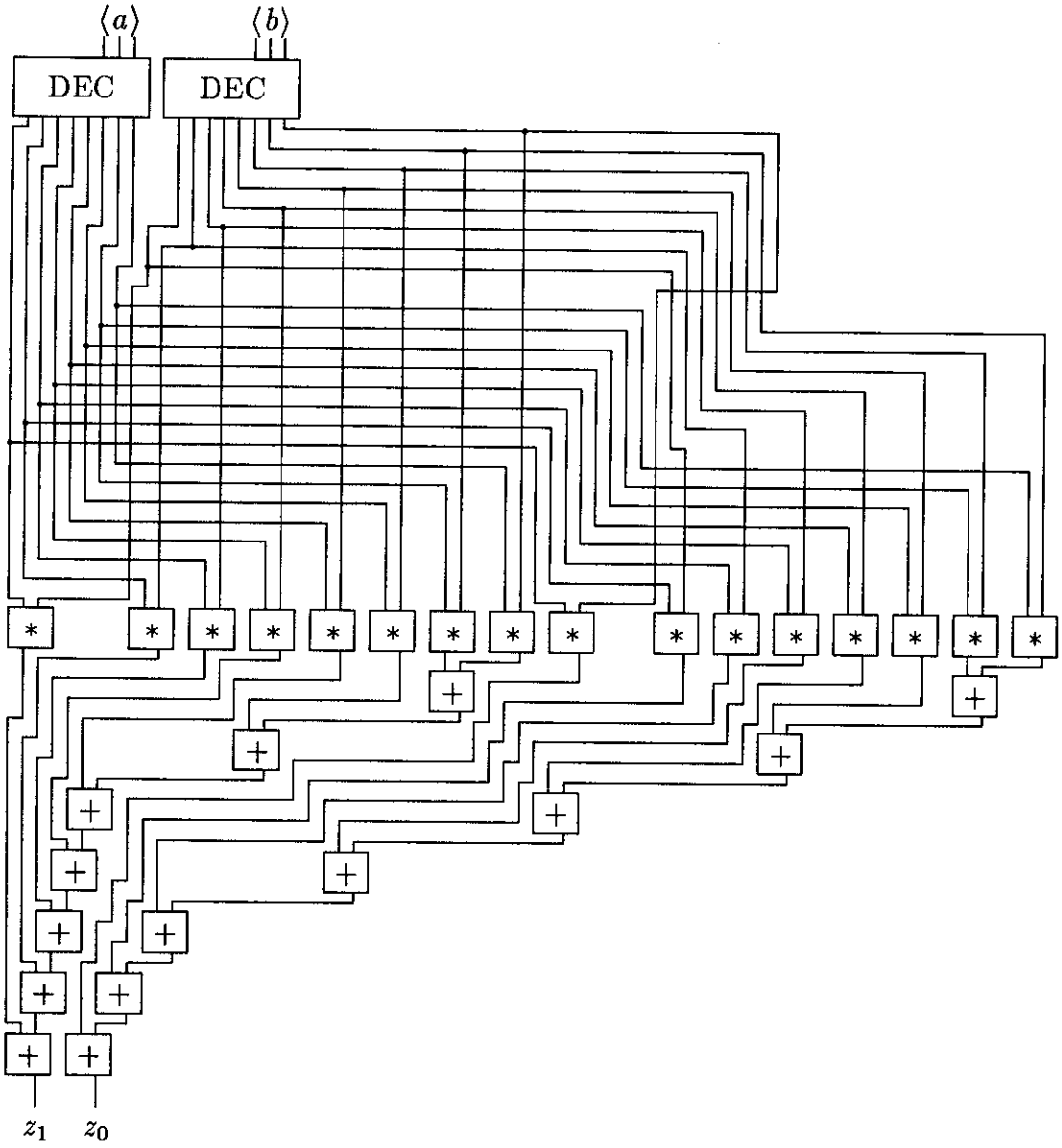


Figure 4.1: Layout of *schemeA*

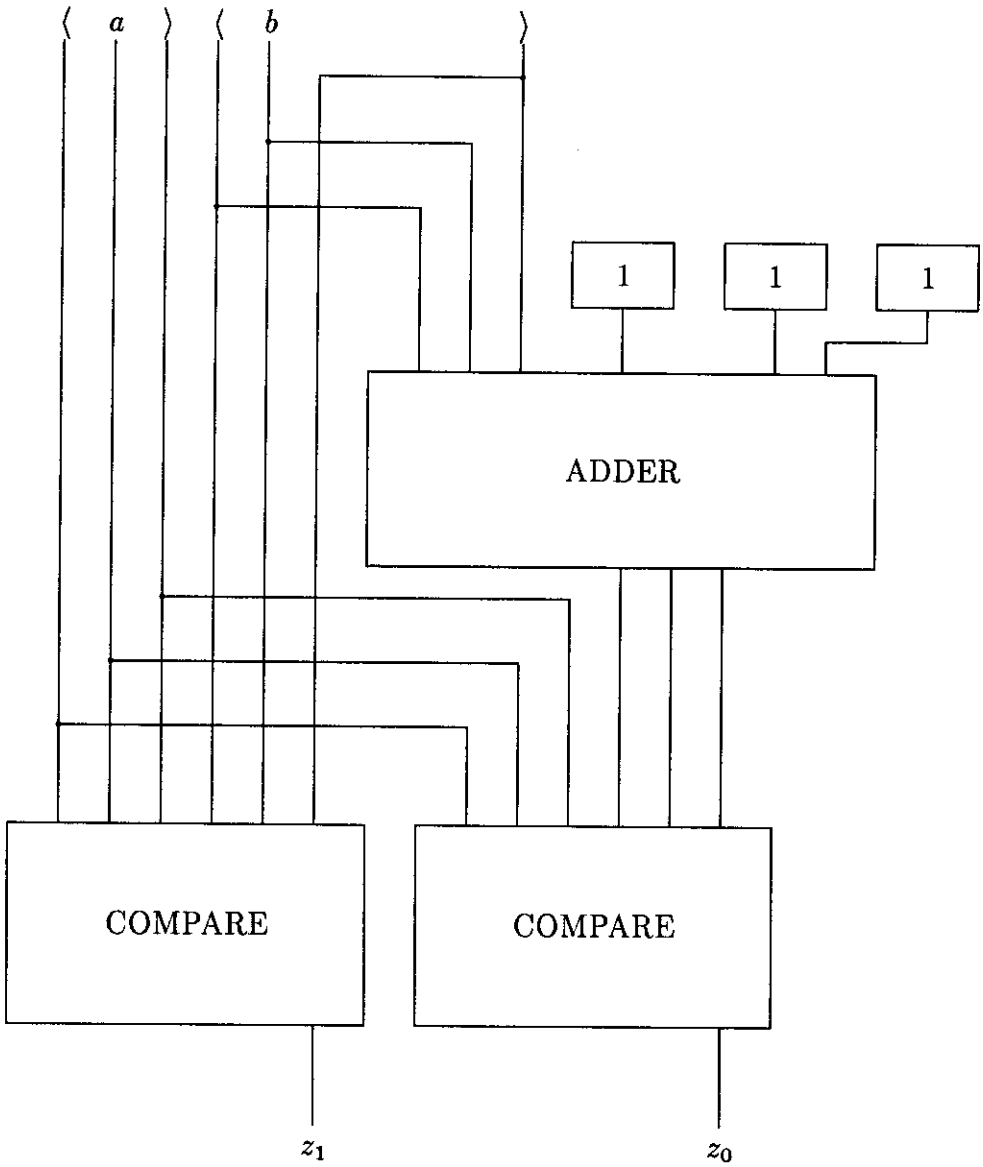


Figure 4.2: Layout of *schemeB*

level	andg	org	notg
1	2		6
2	10		
3	14		
4	14		
5		8	
6		4	
7		2	
Totals	40	14	6

Table 4.1: Statistics for *schemeA*

```
# scheme A inputs :  <<a> <b>> outputs :  <z1 z0>
defun schemeA
      &(♣org ° & andg ° trans) ° distl ° [1,[id,rotr] ° 2] ° & decoder
enddef

# scheme B inputs :  <<a> <b>> outputs :  <z1 z0>
defun schemeB(a,b)
& compare ° distl ° [a, [b, tl ° adder ° [b, [%1, %1, %1]]]]
enddef

defun compare notg ° ♣org ° & xorg ° trans enddef

defun adder
      seq ( FullAdder ° apndr) ° apndr ° [trans, %0 ]
enddef
```

The results in tables 4.1 and 4.2 show that *schemeA* uses a total of 60 gates, while *schemeB* uses a total of 21 gates. However, it is to be noted that 11 of the 21 gates are xor gates which would normally occupy a larger area. Given an estimate of the area occupied for each of the gates, it is possible to have an estimate of the area occupied by each implementation. Since *schemeA* has 7 levels while *schemeB* has 8, *schemeA* would be faster than *schemeB* under the assumption that all the tagged functions had the same delay.

level	andg	xorg	org	notg
1	2	6		
2	1	2	1	
3		1	2	
4		1		1
5		1		
6			1	
7			1	
8				1
Totals	3	11	5	2

Table 4.2: Statistics for *schemeB*

4.3 Summary

This chapter shows how the $\nu\mathcal{FP}$ system provides simple means to incorporate performance estimates for descriptions at any arbitrary level of abstraction. It should be noted that these measurements are on the algorithm itself rather than an implementation of the algorithm because idealistic assumptions are used in calculating these measures. Thus, it is possible to get information about the location and amount of parallelism in the algorithm. Using this and other easily obtained information, it is possible to obtain an approximation for the data bandwidth involved at various locations in the algorithm. This will have impact on the interconnection requirements of the algorithm and the area utilization as mentioned above. At this crude level, the area that is calculated does not include any routing area that may be required. Power requirement estimates can be calculated by summing the power requirements for the individual cells. The two most important features of this method of gathering measures about the algorithm are that these measures can be generated at any level of abstraction and that there is a very low overhead associated with the collection of this data. In a preliminary implementation [79] the performance statistics gathering source code was 15% of the total amount of source code for the system. The performance penalty varied depending on the level of abstraction chosen. If none of the functions are tagged then the only extra cost at run-time is the checking of a variable which is negligible. If all primitives are tagged, the run-time overhead is around 20%.

In addition to the time and space estimates provided by the *level* mechanism, the system can be extended to allow the specification and calculation of user-specified parameters for each

tagged function and for the algorithm as a whole. Examples of user-specified attributes could be power dissipation and communication requirements (e.g., fanout). As was shown earlier, attribute evaluation fits nicely within an applicative framework. This system was only partially implemented. A complete implementation is suggested as future work.

Chapter 5

Applicative Languages in the Design of Algorithms

This chapter demonstrates how an applicative language is useful in the process of designing an algorithm or system. During the detailed designing of a system, it is common to build a behavioral model of the system or algorithm being designed to help in its evaluation and construction. Parts of this model are continually refined and abstracted until the designer is satisfied that the model faithfully reflects the intended behavior. Though most current modeling systems allow the designer to construct models at different levels of abstraction, no help is provided in *proving* behavioral equivalence under abstraction or refinement. One way to approach this problem would be to set up a system that would be able to take two models at different abstraction levels and then prove them equivalent. An alternative approach, the one taken here, is to provide refinement rules that guarantee that the refined model is behaviorally identical to the unrefined model. Either of these approaches is difficult, in general, because of the presence of side-effects in the descriptions. The use of an applicative language like $\nu\mathcal{FP}$ in the modeling system provides the basis for proving behavioral equivalence under refinement by avoiding side-effects.

Design systems often provide users with the ability to have different views of the same design object. This allows designers to use the view that is most appropriate at the given point in the design process. However, support for moving smoothly between these views is often lacking. This chapter shows how $\nu\mathcal{FP}$ can be used to change the representation view in a provable manner so that the designer is guaranteed that all the views represent the same behavior. This is accomplished via the use of transformations between views.

Any number of different modeling systems can be used to create the models. This discussion is restricted to models capable of expressing concurrent behavior. One such model is the Graph

Model of Behavior (GMB) [29]. It is an asynchronous model of behavior whose control flow model is similar to Petri nets [30]. The GMB was chosen because it is a sufficiently general model of computation, because it has many imperative features that could benefit from the introduction of an applicative language, and because an implementation of it was available to the author.

5.1 The Graph Model of Behavior

The Graph Model of Behavior [81, 82] is a model of computation that is suited for describing concurrent asynchronous behavior. The GMB consists of three separate but related domains. The *control domain* deals with sub-computations and the dependence between them. The *data domain* describes the flow and storage of data. The *interpretation domain* specifies the data transformations and control flow decisions made by a sub-computation. The token machine is an abstract machine which interprets a GMB and thus gives it meaning.

5.1.1 The Control Domain

The *control domain* is similar to a Petri net and deals with the specification of processes and the precedence between processes. The control domain is described by a directed hyper-graph[1] consisting of *control nodes* which represent processing activity and *control arcs* which define a partial order on control node initiation events. *Tokens* may be placed on control arcs. They represent loci of control during the initialization and execution of the model. More than one control arc may enter or leave a control node. In this case, a *control logic* is defined on the input and output arcs of a node. The input control logic and the associated semantics specify the conditions under which the control node will be initiated and the input tokens that will be removed when the control node fires. At the termination of the control node tokens are placed on output arcs in a combination that satisfies the output control logic of that control node. Let the boolean variable a represent the condition when one or more tokens are present on the control arc a. More formally,

$$\mathtt{a} = \begin{cases} \mathtt{true} & \text{if there is at least one token on control arc a} \\ \mathtt{false} & \text{otherwise} \end{cases}$$

The control logic is any boolean combination of the boolean variables using the boolean and and or operators. The boolean not operator is not allowed in the GMB and hence it is not

[1] A hyper-graph is one in which edges between vertices may have more than one head and more than one tail.

possible to model "inhibitor" arcs. An input logic expression can be described by the following BNF [83] :

$$\begin{aligned} expr = \quad & expr + expr \\ | \quad & expr * expr \\ | \quad & expr > expr \\ | \quad & expr \end{aligned}$$

As far as enabling is concerned, the priority operator, ">", can be treated the same as the `or` operator, "+". If the input logic expression evaluates to `true`, then the corresponding control node can be activated (fired). In particular,

$$\texttt{a} * \texttt{b}$$

as an input logic expression means that if there is at least one token on the control arc a, *and* there is at least one token on control arc *b*, the corresponding control node will be enabled. If the notation, used in an input logic expression, is

$$\texttt{a} + \texttt{b}$$

then it means that the node will be enabled if there is at least one token on control arc a *or* at least one token on control arc *b*, *or there is at least one token each on* both *control arcs* a *and b.*

When a node fires, tokens will be removed from input control arcs according to the input logic expression. However, the rules for doing so are slightly different than those for the enabling of control nodes. For token removal,

$$\texttt{a} = \begin{cases} \texttt{true} & \text{if one token is removed from control arc } a \\ \texttt{false} & \text{otherwise} \end{cases}$$

The input logic expression

$$expr_1 * expr_2$$

means that tokens should be removed from control arcs in such a way that $expr_1$ and $expr_2$ are `true`. An input logic expression of

$$expr_1 + expr_2$$

implies that tokens should be removed from control arcs so as to make either $expr_1$ or $expr_2$ `true`. The choice between making $expr_1$ `true` and making $expr_2$ `true` should be made non-deterministically. If the control arcs involved in the input logic expression are simple[2], then a

[2]A *simple* control arc is one which has exactly one head and one tail.

stronger statement can be made that either $expr_1$ or $expr_2$ will be `true`, *but not both*. However, in the case of complex[3] control arcs, this is not possible because removal of a token from an arc may affect both $expr_1$ and $expr_2$. If the input logic expression of an enabled control node is

$$expr_1 > expr_2$$

then tokens should be removed from control arcs in such a fashion that $expr_1$ is `true`. Only if that is not possible, should tokens be removed such that $expr_2$ is `true`.

If the control node is a *single-server* node, then the node cannot be re-activated while it is currently active. If it is an *infinite-server* node then it can be re-activated concurrently, if its input logic expression is satisfied while it is currently active. The single-server and infinite-server models were first proposed by Vernon in [84].

5.1.2 The Data Domain

The *data domain* deals with the flow and storage of data in the model. *Datasets* represent place holders for data values. Simple datasets only hold one value at a time. Writing a simple dataset destroys its previous value. Reading a simple dataset leaves the current contents unchanged. *Dataset queues* are datasets that can hold multiple values at the same time. Reading a dataset queue removes data from the dataset queue and writing to it appends data to it. If a dataset queue is empty and a data processor reads from it, the read will succeed (i.e., it is a non-blocking read), but the value returned is undefined (see page 43 of [84]). In the current implementation, dataset queues can either be first-in-first-out (FIFO) or last-in-first-out (LIFO). *Data processors* are data transformers. *Data arcs* are directed arcs between data processors and datasets. They show access paths between processors and datasets. Data processors can access datasets only via data arcs. Each data processor is associated with a set of control nodes. When any node in that set is active, the corresponding data processor is activated. It then reads data values from a subset of its input data arcs, performs any data transformations necessary and then writes out data values to a subset of its output arcs.

In [84] Vernon introduces the use of a *controlled read* and *priority read* data arcs. There is a one-to-one mapping between a control arc and a data set that is connected to one of tails of the data arc. In this case the data arc has one head and multiple tails. The idea is that there is an explicit and enforceable relation between actions in the data domain and in the control domain. If a processor attempts to read data from the head of such a data arc, the data from the appropriate data set is automatically provided. The data set chosen will be the one corresponding to the control arc which caused the invocation of the control node. For example,

[3] A *complex* control arc is a control arc that is not simple.

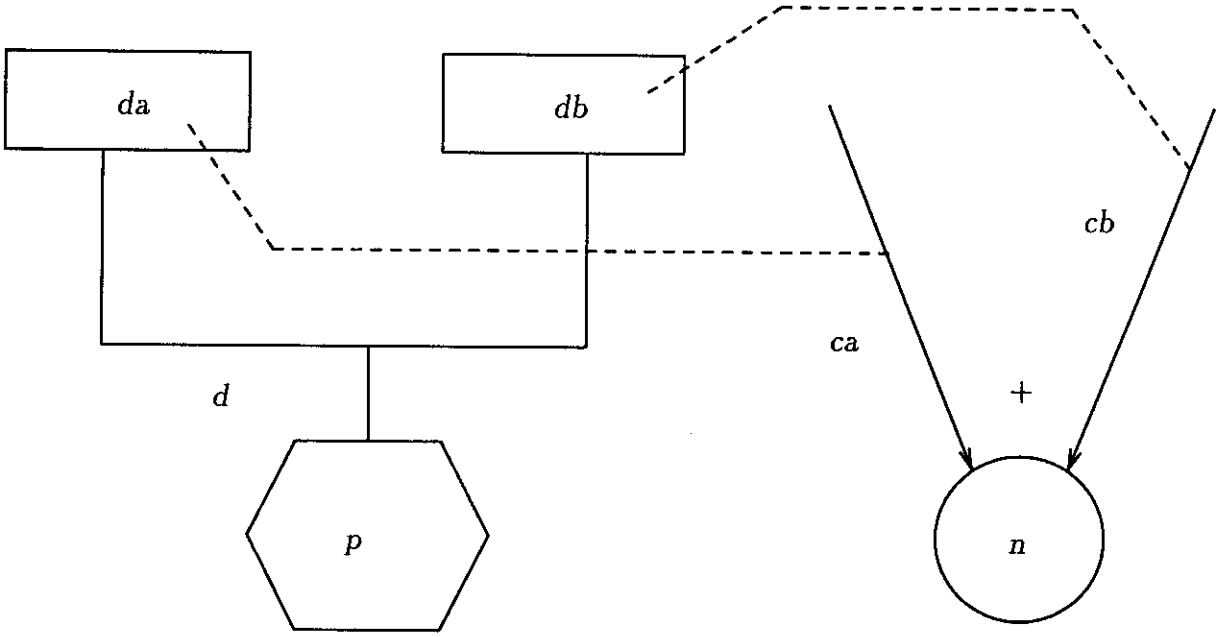


Figure 5.1: Controlled-Read Data Arcs

in figure 5.1, if there are tokens on both control arcs a and b, and if node n is a single-server node, either the token from a or the token from b will be removed when n is fired. If the token from a is removed, and processor p reads from data arc d, then it will get the value of data set da. If the token from b was removed, d would get the value of db. If the input logic for n was

$$\mathtt{a} > \mathtt{b}$$

instead of

$$\mathtt{a} + \mathtt{b}$$

the situation would still remain the same except that if there were tokens on both arcs, the token from arc a would be removed first and the corresponding data from da would be read. Details on precise semantics of the GMB may be found in [84] and in appendix B.

5.1.3 The Interpretation Domain

The *interpretation domain* specifies the particular data transformation performed by each data processor. In addition, it specifies how long each data processor (and hence the associated control node) is active, and which control arcs receive tokens at node termination. The node

delay and output token distribution are dynamically recomputed during each activation of a control node. The values written out on the output data arcs are computed in the interpretation domain. The output value can be written out to the data arc at any time in between the initiation and termination of the node. It is possible to selectively read input values from input data arcs as required. In the case of dataset queues, it is important to read from only those data arcs that are necessary because once the value is read, it is lost from the dataset queue. For this reason, a function called `$trigger` is available. It returns a list of the control arcs which had the tokens that triggered the node. This information can be used to decide which data arcs to read upon node initiation.

5.1.4 The Token Machine

The *meaning* of a GMB is specified by the *token machine*. It is an abstract interpreter that controls the flow of tokens during the execution of a GMB. It handles the storage and movement of data through the data graph and manages the communication between the interpretation domain and the other domains. If there are two data processors that have write access to the same dataset and they have overlapping execution times, the token machine will warn of potential conflict, because the semantics of the GMB specify that a data processor may write onto its output data arcs at any time between processor initiation and termination. One useful property of the token machine is that it does not require global information to determine which nodes can be activated. For this reason, it is possible to implement a distributed token machine. The number and connectivity of control nodes, data processors, and data sets are fixed and do not vary with time. Only the location and quantity of tokens on control arcs and the values contained within data sets can be changed.

5.1.5 Control Flow Analysis

Control-flow analysis [85] is a process that analyses the control flow graph for certain properties independent of the data and interpretation domains. Because of this, control-flow analysis may point out anomalies that may not actually exist if the data graph and interpretation were taken into account. The properties are determined by building a computation flow graph (CFG) for the given control flow graph. Typical properties that are checked for are liveness, boundedness, proper termination, deadlocks, and infinite states. *Proper termination* implies that every state in the CFG can reach a proper terminal state. *Deadlocks* (non-proper terminal states), *traps* (cycles in the CFG that have no exit), and infinite states in the CFG destroy the proper termination property of a CFG. Discussions of these properties and their implications are found in [86, 84, 85].

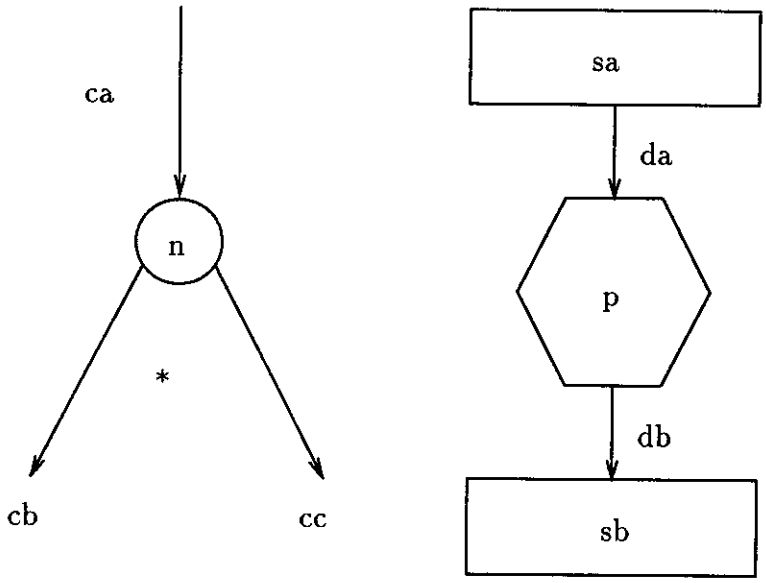
Since the construction of the computation flow graph is exponential in nature, a *reduction procedure* is used to shrink the size of the control flow graph before attempting to generate the computation flow graph. To aid this reduction, the analysis assumes that each control node has infinite-server semantics. Vernon, in [84], has shown that doing this does not introduce any new anomalies nor does it remove any anomalies that were already there. Thus, this simplification is valid if the user is interested in locating the anomalies, but may give incorrect results if control flow analysis is used to determine the actual computational paths in a control graph. All the extensions mentioned previously are compatible with control flow analysis. This means that their introduction does not destroy the ability of the analysis to detect the various anomalies.

5.2 $\nu\mathcal{FP}$ as an Interpretation Domain Language for the GMB

The SARA (System ARchitect's Apprentice) system [29, 82] is a collection of tools for the design, analysis, simulation, testing and synthesis of concurrent asynchronous systems. Currently, it uses the GMB as the primary means to describe the behavior of the system models. Any sufficiently rich programming language can be used to specify the interpretation domain in the GMB. In the past, PL/I and **T** [87] have been used, and Ada, ISP [88], and MPDL [89] have been proposed for use. The following sections show how $\nu\mathcal{FP}$ may be successfully integrated into the SARA system as an interpretation domain language for the GMB.

In addition to being able to calculate output values, any interpretation domain language used with the GMB has to be able to support four other operations. It has to be able to read values from input data arcs, write values to output data arcs, inform the system of the delay of the node, and specify which output arcs receive tokens on node termination. Normally, every $\nu\mathcal{FP}$ function takes one or more input arguments and outputs one result (which may be a list). If $\nu\mathcal{FP}$ is used as an interpretation domain language, the inputs and outputs take on specific meanings which are described in the next paragraph. The name of the $\nu\mathcal{FP}$ function that specifies the interpretation of a data processor is to be the same as the name of the data processor itself. The formal arguments specified for that function are a list of the input data arcs to that processor. All input data arcs are read at node initiation and their values are passed to the $\nu\mathcal{FP}$ function as its actual arguments.

The function's result is a list of three elements. The first element of the list is an association list of *(data arc,value)* pairs. This tells the token machine to write out the specified value to the corresponding data arc at node termination. The second element is a list of control arcs. This




```
defun p ( da )
      if da > %5
      then [[[%db, da-%1]], [%cb], %10]
      else [[[%db, %0]].  [%cc], %1] fi
enddef
```

Figure 5.2: A GMB and its interpretation function

informs the token machine that tokens are to be placed on the specified control arcs on node termination. The third element is the node delay and it tells the token machine how long this node is supposed to be active. For example, figure 5.2 shows a GMB and the corresponding interpretation function written in $\nu\mathcal{FP}$.

There are two restrictions on the type of interpretations that can be specified with this scheme. The first is that *all* the input data arcs are read at node initiation time. This is in contrast to the current scheme, where just those data arcs specified in an `$input` command are read. For simple datasets this does not make a difference, but this may adversely affect GMBs using dataset queues. In most cases where dataset queues are involved, the queue is read on each node initiation. Hence, for most practical cases, this restriction is unimportant. In the case where the control node has either a *priority* operator or an *or* operator in its input logic expression, it must use the appropriate *controlled read* or *priority read* data arcs. So, again,

there is no need to explicitly name the data arc to be read. The requirement for reading every data arc on node initiation, though it slightly reduces the flexibility, more than compensates by providing consistency. The consistency allows stronger statements to be proved about which data values will be used. In addition, automatically reading the corresponding data arc for priority and controlled-read arcs removes a common source of errors.

The second restriction is in the scheduling of data arc writes. Under the current scheme, values may be scheduled to be written to data arcs after any time that is less than or equal to the processor delay. In the proposed scheme, *all* output to data arcs is scheduled to happen upon processor termination. Again, this restriction is not applicable for most practical cases because most output does occur at processor termination. In those cases where intermediate output is required, a single data processor with intermediate output can easily be split up into two sequential data processors with the intermediate output coming out from the first one. Only those data processors which schedule non-overlapped writes can be converted in this manner. Formally, assume an output value of o_1 is scheduled at time t_{11} for output at time t_{12} on a data arc, and another output o_2 is scheduled at time t_{21} for output at time t_{22} on the same arc. Only if the following condition holds

$$t_{11} \leq t_{21} \Longrightarrow t_{12} \leq t_{22}$$

can the data processor be converted into two sequential data processors.

5.3 Abstraction and Refinement in the GMB

Past research [90, 91] on the Graph Model of Behavior (GMB) has suggested that *refinement* and *abstraction* can be carried out on GMB primitives such as control nodes or data processors. This chapter concentrates exclusively on control nodes. *Refinement*, in this context, means replacing a control node with a GMB consisting of more than one control node. This GMB has the same behavior as the original control node, as far as the rest of the GMB is concerned, but at a lower level of abstraction. In this case, "same behavior" means that if the same inputs (control tokens and data set values) are provided to the control node and the refined GMB, both will give the same output (control tokens and data set values). Initial state, if any, has to be modeled by static variables in the interpretation domain of the control node. *Abstraction*, is the opposite operation—that of taking a sub-GMB and abstracting it to a control node. Again, the new control node should behave the same way as the abstracted sub-GMB. Abstraction and refinement are useful processes in the design of general systems. Here they are used in the context of the SARA design method [82]. Refinement and abstraction of a control node may involve the refinement of other GMB primitives in order to maintain semantics, although this

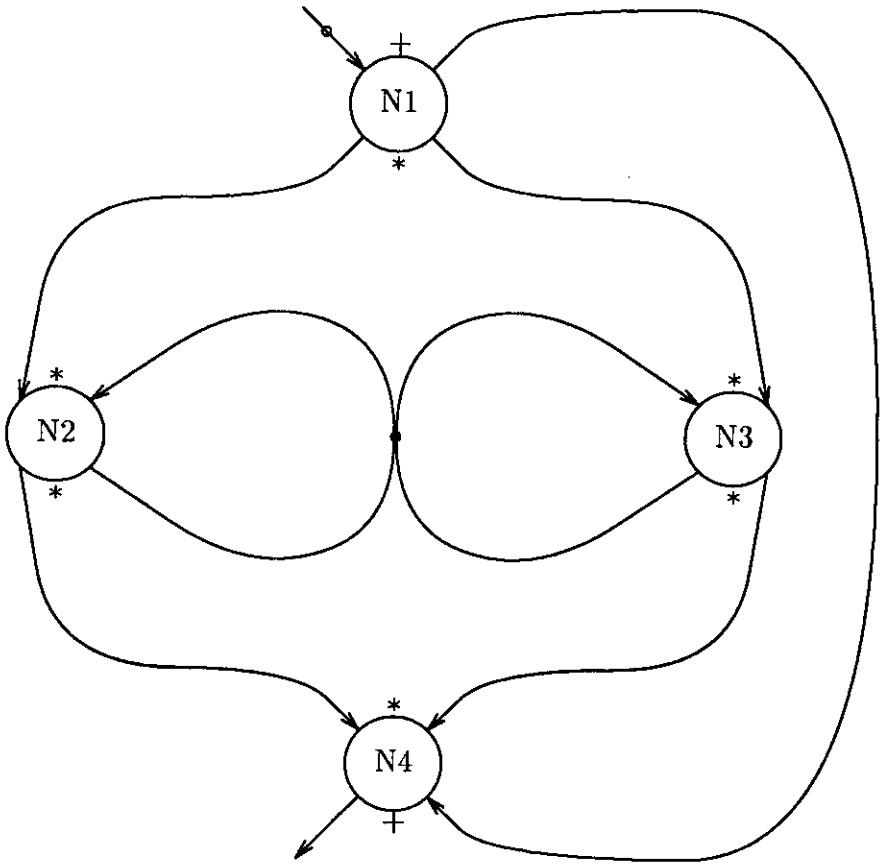


Figure 5.3: Sample GMB

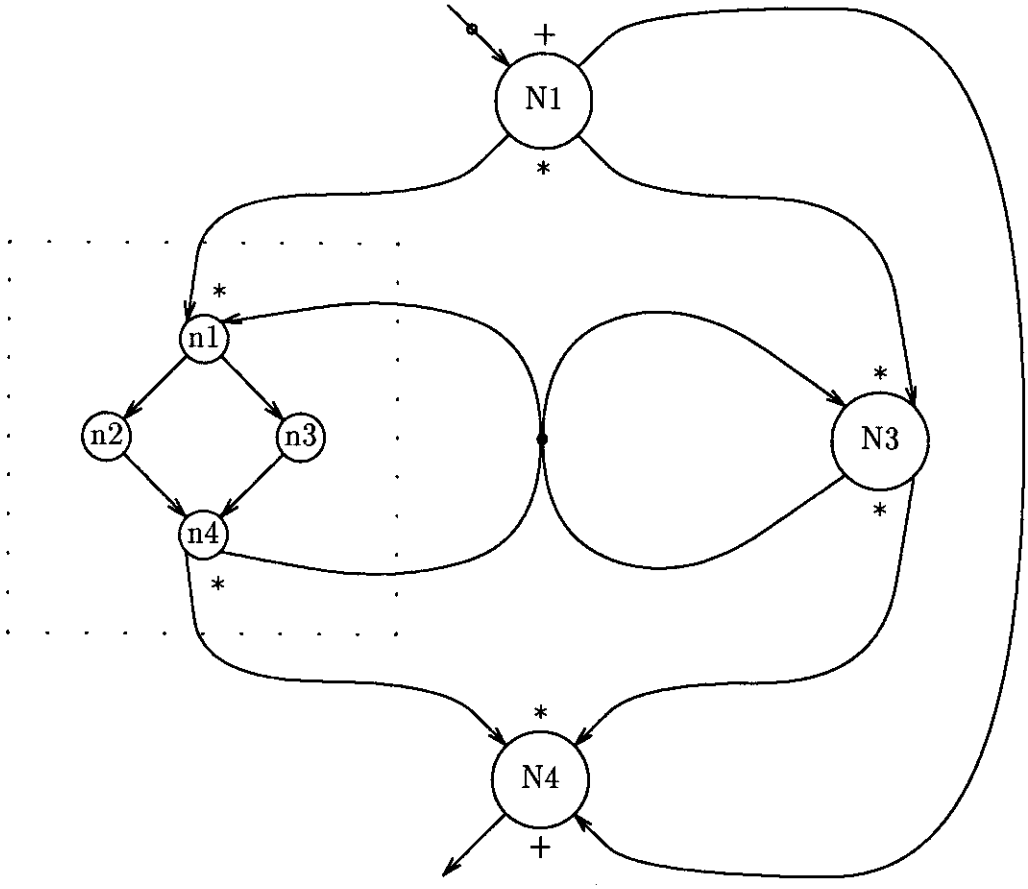


Figure 5.4: Refined GMB

is not addressed here for the sake of simplicity. The control domain is emphasized because it is the only domain used in control flow analysis. In addition, changing the data and interpretation domains to maintain semantics, though not automatic, is straightforward. Figure 5.3 shows a sample GMB and figure 5.4 shows the control graph of a refined GMB where the control node *N2* of the original GMB is refined into a control subgraph. The corresponding data graphs and interpretations are not shown because only control abstraction is considered here. The GMB is a powerful model of computation which allows the designer a lot of freedom in specifying the model. This unconstrained freedom makes it difficult to prove properties about the model that include information in the data or interpretation domains. Owing to this unstructuredness, previous work has been unable to show how to prove conclusively that the control node and the sub-GMB are equivalent with respect to external behavior. At best, test examples were used to verify that the control node and the sub-GMB reacted in similar fashion to each of the test cases.

In an attempt to deal with this problem, Ruggiero [91] and Campos [90] restrict the class of sub-GMBs to single-entry-single-exit (SESX) sub-GMBs. Intuitively, a SESX GMB is a GMB that has one distinguished input primitive and one distinguished output primitive. The type of the input primitive is the same as that of the output primitive (for example, they could both be control nodes). With this restriction, it is possible to perform a *syntactic* refinement or abstraction. "Syntactic", in this context, means, for example, that the SESX sub-GMB in figure 5.4 replaces the node *N2* of figure 5.3 in place without regard to any meanings (or interpretations) associated with the node. Conversely, only SESX sub-GMBs can be abstracted into single GMB primitives (control nodes, in our case). Ruggiero [91] shows that if the original GMB was properly-terminating, and if the replacement sub-GMB is properly-terminating, then the resultant GMB, after refinement, will also be properly-terminating. This is an important result, because it allows us to infer properties of the refined GMB independent of the interpretation domain. However, this result still does not say anything about whether the two GMBs will, in fact, result in the same external behavior. This is because the behavior of the nodes in the refined sub-GMB is not taken into account.

Even these restrictions are not strong enough in general because sequential specification languages have been used in the interpretation domain of the GMB for the most part. The GMB is a parallel model of computation, and if a sequential language is used in the interpretation domain, there is a mismatch created during abstraction or refinement. In one case, it is necessary to translate a program written in a sequential language to one in a parallel language. This, as shown in [92], is a hard problem to tackle in the presence of side-effects. The problem is that it is difficult to uncover the dependencies between statements or data values when the algorithm is expressed in a conventional imperative language. The converse problem of taking a program written in a parallel language and converting it to a program in a sequential language

is easier. Even so, it is not possible to perform a straight-forward syntactic transformation. A full semantic analysis has to be performed in order to transform the program. Normally, this is the case because even though the language allows the description of parallelism, the dependencies between statements are not syntactically obvious and have to be deciphered using semantic analysis. For instance, if there is a multiple parallel assignment statement that exchanges the values of two variables

```
a := b || b := a ;
```

then the translation has to figure out that there needs to be a temporary variable that is needed to store an intermediate value, when this parallel program is translated into a sequential one.

Using a parallel language in the interpretation domain alleviates some of these problems but creates other ones. This approach is taken by Krell [93] and Overman [89], though refinement or abstraction was not their emphasis. Krell concentrated on modeling Ada tasking primitives with the GMB. In most of the cases he was able to model all the semantics of the tasking primitives using only the control domain. In a few cases, he had to resort to using the interpretation domain as well. Krell also showed how to translate some specific GMB subgraphs into equivalent Ada programs. He leaves open the question of whether there are any GMB constructs that are impossible to model in Ada. Using an interpretation domain language capable of expressing parallelism (in this case, Ada) allows Krell to move representations of the algorithm that have been modeled in the interpretation domain into the control domain. Overman [89] describes how to translate a GMB into concurrent state deltas[4] which describes a parallel model of computation. In contrast to Ada, concurrent state deltas make the dependencies between the state deltas explicit and hence it is possible to easily transform concurrent programs in that representation into sequential ones.

In addition to the software engineering benefits of using a language without side-effects, $\nu\mathcal{FP}$ allows us to prove that the refinements and abstractions that we are allowed to perform do indeed maintain the behavior of the two GMBs. Obviously, any arbitrary control node cannot be refined; nor can any arbitrary sub-GMB be abstracted. The class of GMBs on which these procedures can be successfully applied is described at the end of section 5.2.

[4]Concurrent state deltas are a formalism for describing concurrent behavior. They consist of *state deltas* that may be active concurrently. State deltas represent actions, and are specified by pre-conditions, that must hold before a particular state delta will be activated, and post-conditions, that hold after the state delta has completed. This is a simplified view of concurrent state deltas.

5.4 Refinement via Transformations

The basis of the approach used to allow refinement in the GMB is to use an applicative language (in this case, $\nu\mathcal{FP}$) as an Interpretation Domain Language. Since $\nu\mathcal{FP}$ is referentially transparent, there exists an algebra of $\nu\mathcal{FP}$ programs which allows us to have a set of transforms that are guaranteed to maintain the meaning of the program. The main idea is to develop a translation between $\nu\mathcal{FP}$ constructs and GMBs. These should be direct enough that their correctness should be obvious. Using these translations, the interpretation of a control node can be transformed to an equivalent GMB. Because the translations are meaning-preserving, we can be sure that the resultant GMB behaves properly. Since $\nu\mathcal{FP}$ has inherently parallel semantics, it is a good match for some of the parallelism expressed by the GMB. Performing a naive one-to-one translation from $\nu\mathcal{FP}$ will result in a correct GMB, but one which may have less parallelism in it than would be desirable. Transformations can again be used to convert that GMB to one that exhibits more parallelism but does not change the meaning of the GMB.

In the process of incorporating $\nu\mathcal{FP}$ into the GMB, it was discovered that certain aspects of the semantics of the GMB had been inadequately specified in the past. Appendix B remedies that. All further discussions in this chapter assume the semantics specified in Appendix B.

5.5 Equivalences between $\nu\mathcal{FP}$ Constructs and GMBs

This section shows equivalences between some $\nu\mathcal{FP}$ constructs and GMBs. Only the equivalences for the basic combining forms of $\nu\mathcal{FP}$ are shown since all other combining forms can be modeled in terms of the basic ones. It also discusses under which conditions these equivalences are valid and which $\nu\mathcal{FP}$ constructs have no GMB equivalences. For example, general recursion is not possible in the GMB because it is a semi-static model of computation in the sense that it is not possible to create new GMBs on the fly.

The easiest is the *compose* functional form. This takes two functions and composes them together sequentially. Since composition in $\nu\mathcal{FP}$ is denoted in the same way as mathematical composition, the ordering is backwards and the output of the second function specified is passed as the input to the first function specified. For example, if we had the program segment

$$f \circ g$$

the corresponding GMB would be that shown in figure 5.5.

The *construct* is a parallel functional form that composes two functions together in a concurrent manner. The program segment corresponding to this is

$$[f, g]$$

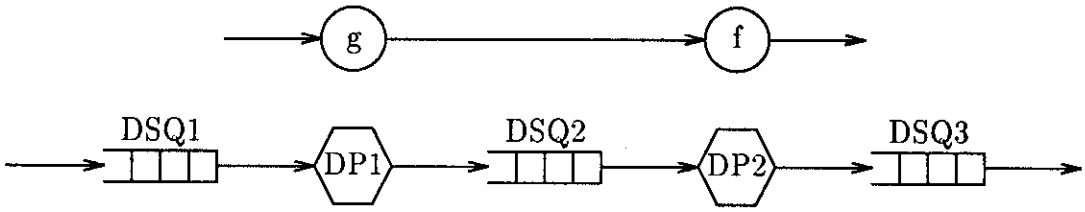


Figure 5.5: Compose

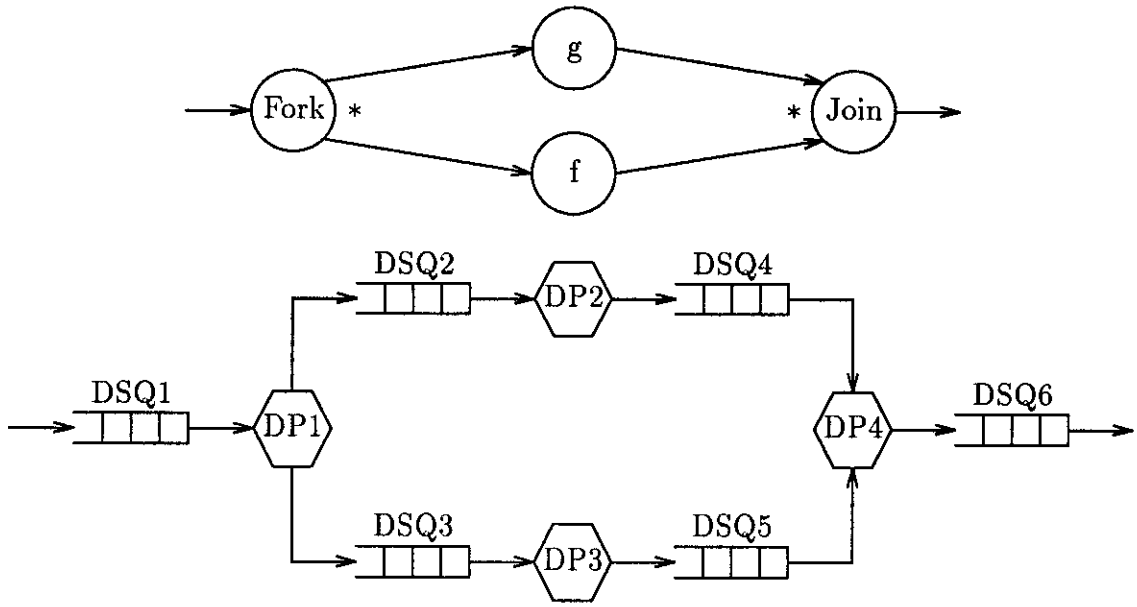


Figure 5.6: Construct

and its GMB is shown in figure 5.6. Two additional control nodes called *Fork* and *Join* have been introduced to handle the synchronization. The interpretation of the *Fork* node duplicates its input on both of its output arcs. The interpretation of the *Join* control node reads off one value from each of its input data-arcs and constructs a sequence out of them. This sequence is then written onto its output arc. The order in which the items are constructed into a sequence is important and dependent on the order specified in the original $\nu\mathcal{FP}$ program segment.

The *conditional* functional form concerns us next. The $\nu\mathcal{FP}$ program segment corresponding to the GMB in figure 5.7 is

if p **then** q **else** r **fi**

Again, we need an extra *Fork* control node to duplicate the input value. Its interpretation is

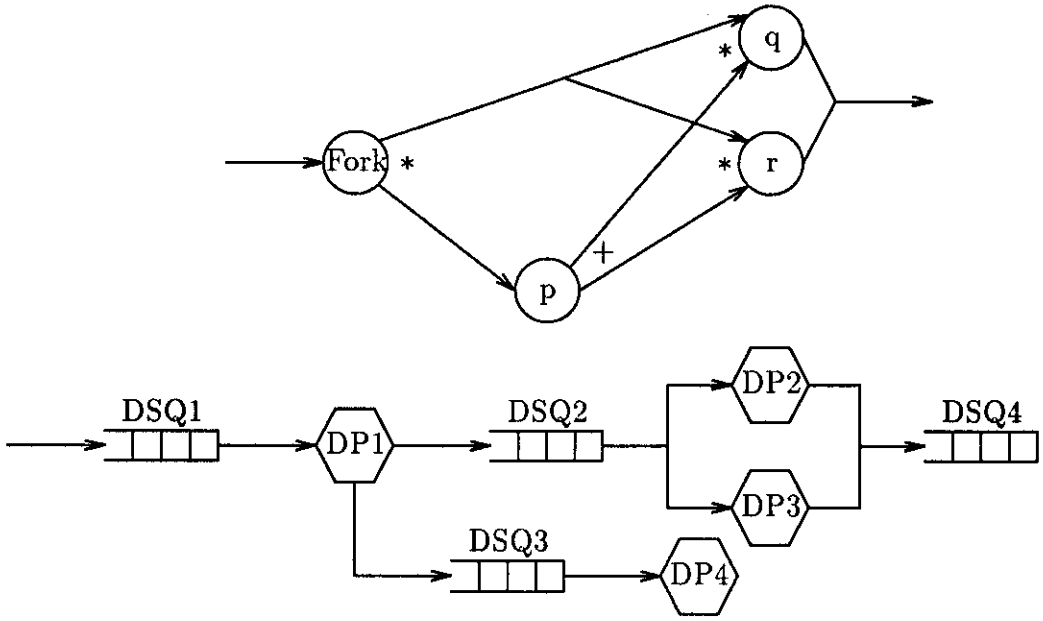


Figure 5.7: Conditional

the same as mentioned previously. However, since we know that only one of q or r is going to be executed, we need only make one copy for both of them. For the same reason, we do not need a *Join* control node to provide any synchronization. The control node corresponding to p will read a value from its input data-arc. If p applied to that value results in the boolean value `true`, a token is placed on the control arc leading to the control node q. If the result is `false`, then a control token is placed on the arc corresponding to the node r.

Unfortunately, it is not possible to represent general recursion in the GMB. The corresponding $\nu\mathcal{FP}$ program segment to represent general recursion would be

$$\textbf{defun } f \textbf{ if } p \textbf{ then } q \textbf{ else } r \circ f \circ h \textbf{ fi enddef}$$

The problem lies in the position of the function r. This could be done if the GMB had facilities to create GMBs at execution time, but since the GMB is a static model, this is not possible. Thus, it is impossible to convert a general $\nu\mathcal{FP}$ recursive function to an equivalent GMB.

For the special case of a tail recursion, it is possible to construct a GMB that performs the equivalent iteration. A tail recursion can be expressed in $\nu\mathcal{FP}$ by the following program segment

$$\textbf{defun } f \textbf{ if } p \textbf{ then } q \textbf{ else } f \circ h \textbf{ fi enddef}$$

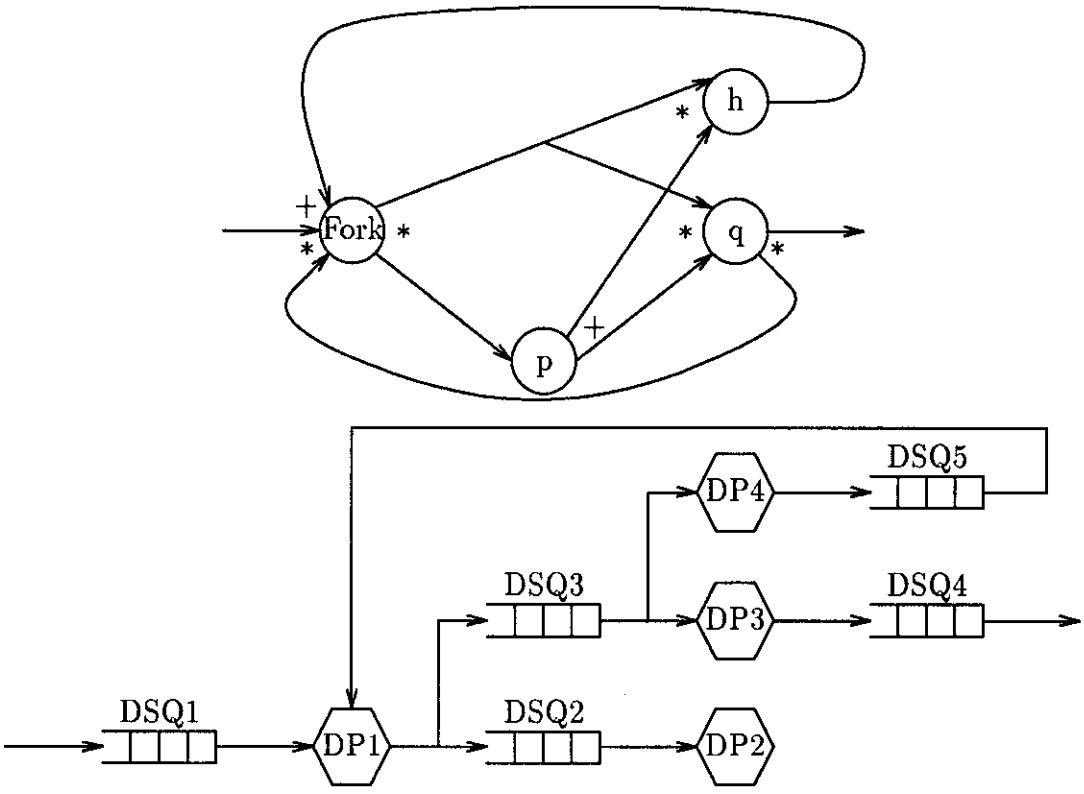


Figure 5.8: Tail Recursion

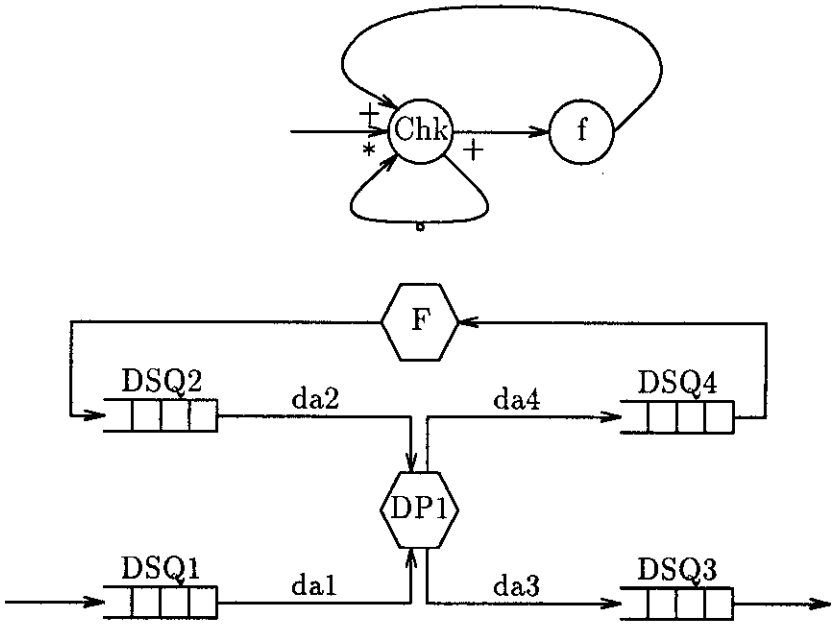


Figure 5.9: Right Insert

The corresponding GMB is shown in figure 5.8. This GMB is similar to that of the conditional, with a few additions. The first difference is that the *Fork* node can be triggered by either the initial token coming in, or from the iterated token. However, there is no change in the interpretation of the node since all the complexity is handled in the control domain. The other difference is the additional control arc from the node q to the *Fork* node. This is necessary to prevent this GMB from executing two different iterations of the loop at the same time. This forces a serialization of successive iterations even if there are no data dependencies. If the GMB had colored tokens, this restriction could be removed.

The *right insert* functional form in $\nu\mathcal{FP}$ performs an iteration over a sequence of elements. This is similar to the tail recursion just handled, except that the stopping criterion is the exhaustion of the elements rather than some predicate on the input. The $\nu\mathcal{FP}$ program segment is

$$! \ f$$

and the GMB corresponding to this is shown in figure 5.9. The data processor *DP1* is the key to this arrangement. It receives the complete input sequence and first picks off the last pair of elements. If there is only one element in the sequence, this is just sent to *DSQ3*. Otherwise,

the pair is sent off to *DSQ4* for consumption by the data processor F. Until all the elements of the sequence are consumed, the result from *DSQ2* is composed with the currently last element of the sequence and resent to *DSQ4*. After the sequence is exhausted, the output is sent to *DSQ3*. The exact interpretation domain code for the processor *DP1* is given below.

```
if = ° [length, %1]
then
     [[%da3, id]]
else
    [[%da4, concat ° [tl, tl ° tl]]]
fi
```

This particular transformation does not result in a parallel GMB, but can be useful if the designer needs to monitor each iteration. The transformation shown is for the *right insert* functional form. The *left insert* is similar except that the sequence elements are supplied to f sequentially from the *beginning* of the sequence rather than from the *end* of the sequence.

In the case when the function f to be inserted is associative, the designer has the option of using the *associative insert* functional form. The semantics of this form allows the parallel evaluation of the insert in a tree fashion. Hence, it is not necessary to sequentialize the iteration and a parallel evaluation strategy can be used. However, unless the structure of the input (that is, the number of elements in the input sequence) can be deduced prior to execution, this cannot be transformed to a GMB owing to the static nature of the GMB. If it *is* possible to deduce the length of the input sequence a priori, a GMB can be created that is similar to that created for the *construct* form. This is shown in figure 5.10.

The *apply-to-all* functional form

$$\& f$$

can similarly be expanded only if the structure of the input is known before execution. Basically what the transform does is to take an *apply-to-all* and convert it into a *construct* with the appropriate number of functions. The GMB for this form is shown in figure 5.11. The similarity to the *construct* form shown in figure 5.6 should be obvious.

5.6 The GMB Refinement Procedure

Now that the correspondence between $\nu\mathcal{FP}$ forms and GMBs have been described, the refinement procedure for GMBs can be demonstrated. Suppose that a GMB at a particular level of abstraction is given and that the interpretation domain language is $\nu\mathcal{FP}$. The control graph

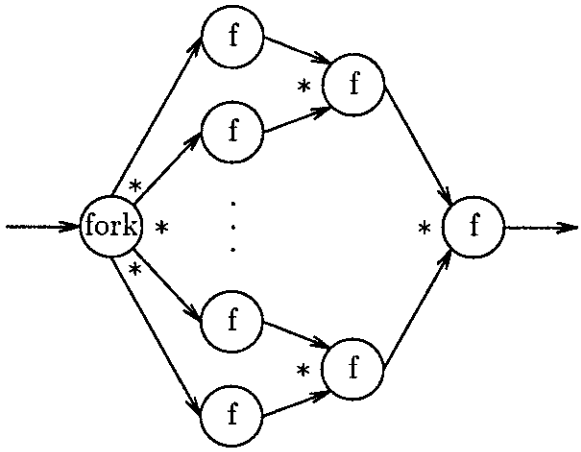


Figure 5.10: Associative Insert

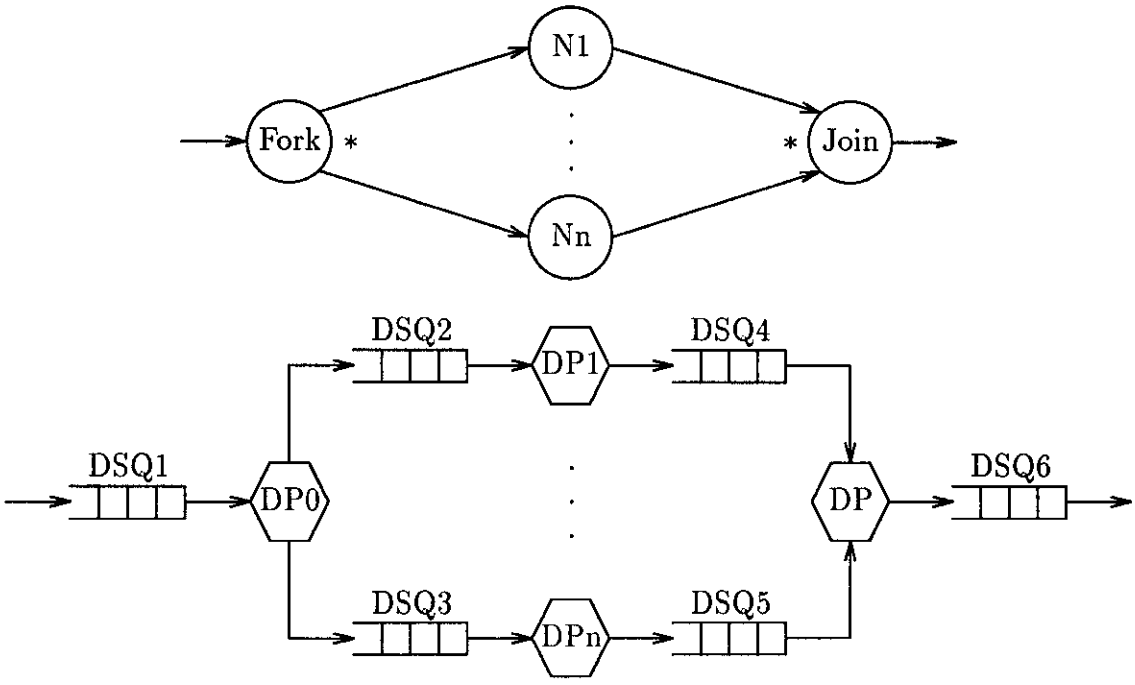


Figure 5.11: Apply-to-all

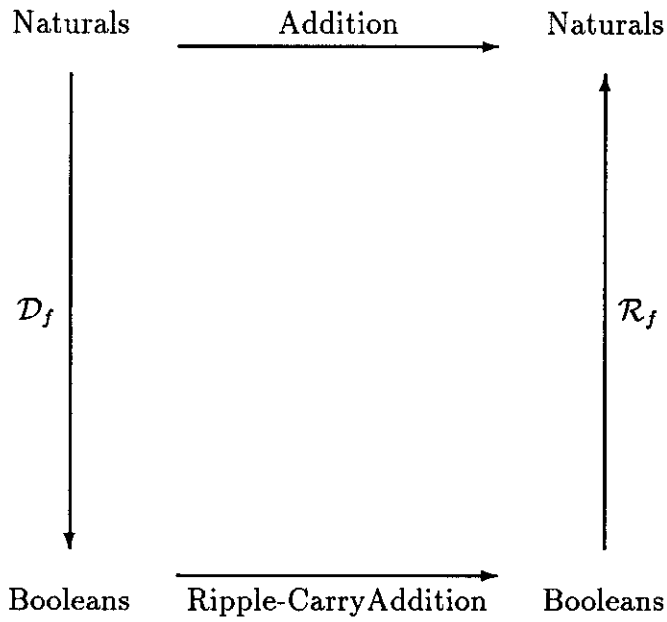


Figure 5.12: Commutative Diagram for a Ripple-carry Adder

is to be modified to represent the model at a lower level of abstraction. Usually this is because the designer wants to explicitly model, in the control domain, more of the concurrency present in the model. Most often, this is done to permit control-flow analysis on the resultant GMB. It should be noted at this point that this procedure does not give designers any *net* refinement or abstraction. Rather, it allows them to conveniently choose what they want to model in the control domain and what they want to model in the data domain. Moreover, it allows them to change this view in a smooth manner as they go along. First, users should use the algebra of $\nu\mathcal{FP}$ programs to transform their interpretation domain to the level of abstraction they intend. Since this is done using the algebra of $\nu\mathcal{FP}$ programs, the resultant interpretation domain program is guaranteed to be equivalent to the original in behavior. Now that the interpretation is at the lower level of abstraction the transformation rules shown in section 5.5 can be used to transform the required subsets of the interpretation domain program into equivalent GMBs.

As mentioned above, this procedure does not result in net refinement. For example, consider the problem of proving that a ripple-carry adder implements the addition function. The solution consists of three steps. The first step is to prove that the ripple-carry adder works on bit-vectors. The second is to show the mapping from natural numbers to bit-vectors. The third is to show the reverse mapping from bit-vectors to natural numbers. This scheme is often demonstrated using a commutative diagram as in figure 5.12. The proof for the ripple-carry

adder can be carried out using the techniques described above. However, the above techniques do not say anything about the mapping from naturals to booleans and vice-versa. In other words, the techniques of this chapter refer to the refinement of the algorithm and are not concerned with the refinement of the domain and range of the algorithm being refined. Chin [94] proposes methods for performing such refinements using transformations. These methods can be directly applied to the $\nu\mathcal{FP}$ framework.

5.7 The GMB Abstraction Procedure

As has been mentioned before, abstraction is the opposite of refinement. The previous section uses the transformations of section 5.5 to translate $\nu\mathcal{FP}$ expressions to equivalent GMBs. Abstraction requires that these same transformations be applied in the reverse direction to translate known sub-GMBs to the appropriate $\nu\mathcal{FP}$ constructs in the interpretation domain of a control node. As will be noticed, all the transforms in section 5.5 produce properly-terminating SESX GMBs. If users want to perform an abstraction on a particular GMB, they first scan the GMB for graph patterns that match the GMBs mentioned in the transformations. If such a pattern is found, then that sub-GMB can be replaced by an equivalent control node whose interpretation is given by the corresponding $\nu\mathcal{FP}$ expression. Obviously, not all GMBs can be abstracted in this manner. In particular, GMBs often contain control graphs that model mutual exclusion. Unfortunately, general mutual exclusion cannot be directly translated into $\nu\mathcal{FP}$ expressions. However, for the special case when mutual exclusion is used to serialize the execution of two or more processes (as shown in figure 5.13), it is possible to translate that particular GMB to $\nu\mathcal{FP}$ by simply choosing one particular sequence of execution for the concurrent processes.

5.8 Optimizations

Naive applications of the rules in section 5.5 will produce GMBs that are correct, but which may be very inefficient. This section discusses some simple optimizations that can be performed to alleviate some of the inefficiency. In this context, "inefficiency" means that there is unnecessary serialization of computations that could possibly run concurrently. There are two classes of optimizations that will be discussed. The first kind are those that are based on $\nu\mathcal{FP}$ laws and thus can be performed on either the $\nu\mathcal{FP}$ program itself or on the resultant GMBs. Since it is easier to manipulate the $\nu\mathcal{FP}$ program, it is suggested that these optimizations be performed on that representation. The other class of optimizations are those that make use of

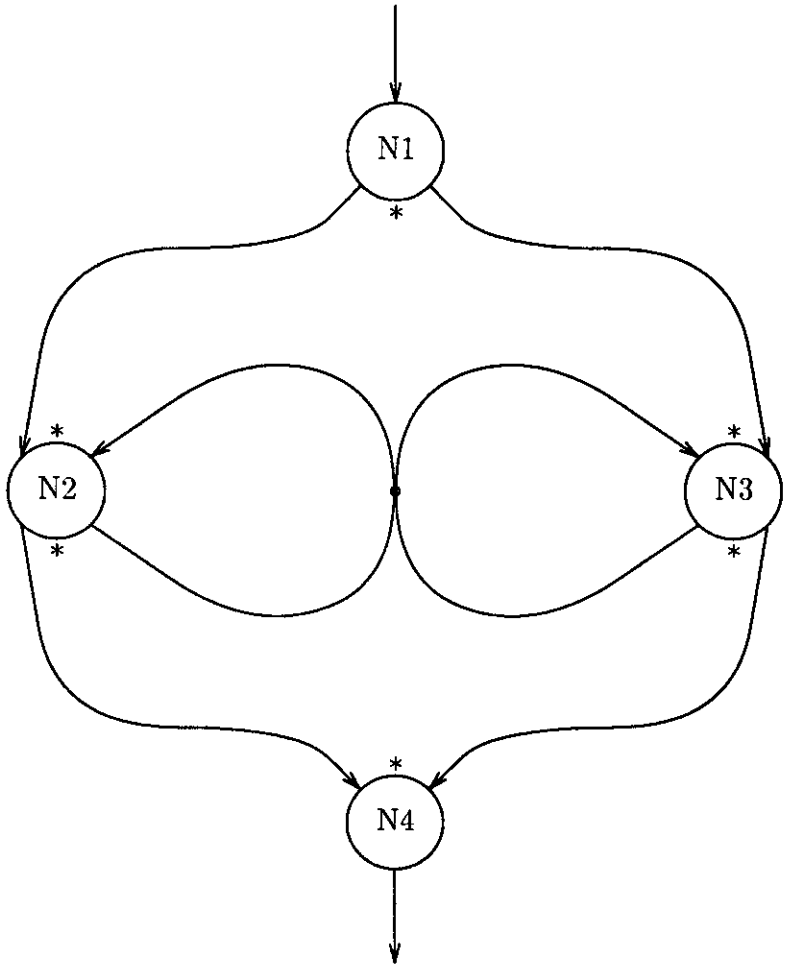


Figure 5.13: Simple mutual exclusion of processes

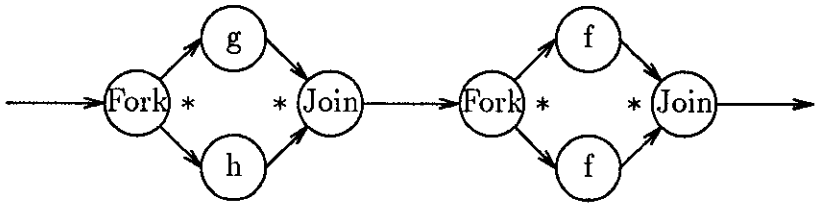


Figure 5.14: GMB corresponding to $\&f \circ [g,h]$

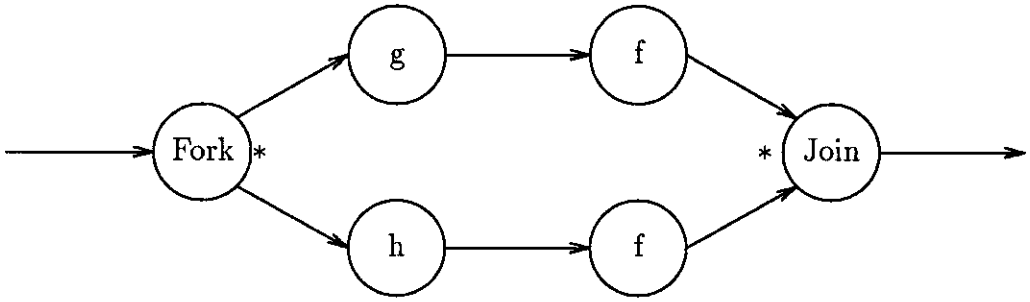


Figure 5.15: GMB corresponding to $[f \circ g, f \circ h]$

the semantics or other known properties of the GMB.

As an example of the first class of optimizations, consider the following $\nu\mathcal{FP}$ law.

$$\& f \circ [g, h] \equiv [f \circ g, f \circ h]$$

The corresponding GMB for that law is shown in figures 5.14 and 5.15. If the user performed a naive translation of the left hand side of the above equation, they might end up with a GMB that is like the one shown in figure 5.14. As can be seen, there is an unnecessary serialization (synchronization) performed by the *Join* node because it is immediately followed by a *Fork* node. It would be preferable, in general, to use the GMB shown in figure 5.15 There are other simple optimizations based on the $\nu\mathcal{FP}$ laws such as this one.

Very often, in $\nu\mathcal{FP}$, it is necessary to use a succession of selectors to get at a particular item of interest. An example would be

$$1 \circ 4 \circ 3$$

This means that you first take the third element of the sequence, then extract the fourth element of *that*, and then the first element of *that*. A naive translation, would lead to a GMB as shown in figure 5.16. However, since the data arcs of GMBs have the capability of specifying that only parts of the dataset (or dataset queue) should be accessed, a sequence of composed selectors can be replaced by just one data arc as shown in figure 5.17.

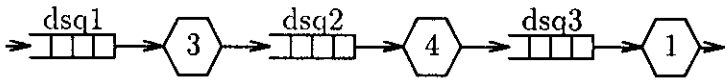


Figure 5.16: A sequence of selectors

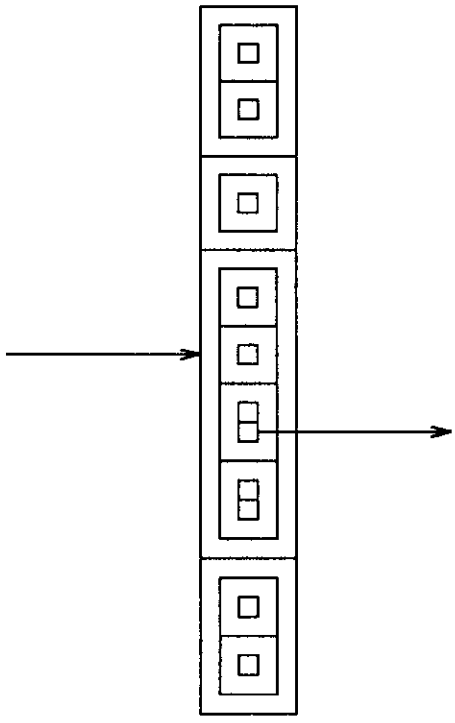


Figure 5.17: Selective data arc access

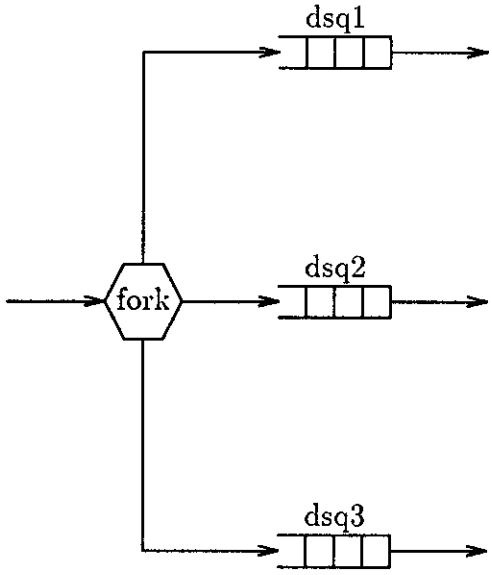


Figure 5.18: Fork node using dataset queues

In many cases, dataset queues can be replaced by simple datasets if it can be determined that there will only be at most element in the dataset queue at any time. There are two methods of insuring that this condition holds. One is by analysis, and the other by introducing explicit concurrency control mechanisms with extra control arcs and control tokens.

For example, if we know that the output dataset queues of a *Fork* controlled processor are going to have at most one data element in each, we can replace the GMB in figure 5.18, by the GMB shown in figure 5.19. Not only do we replace dataset queues by the simpler datasets, but, since datasets can be read multiple times without having their contents destroyed like dataset queues, it is possible to reduce the number of dataset queues when the same value is fed to more than one other data processor.

In many cases, control flow analysis can be used to show that control arcs will have at most one token on them. If these arcs are associated with dataset queues (as they will be for all the transformations shown in section 5.5), then those dataset queues can be replaced by datasets. Another way of insuring that at most one token will ever be on a control arc at any time is to use an explicit acknowledgment scheme that will allow a node to output a token only when the previous token (if any) has been removed by the succeeding node. This is accomplished by adding extra control arcs as shown in figure 5.20.

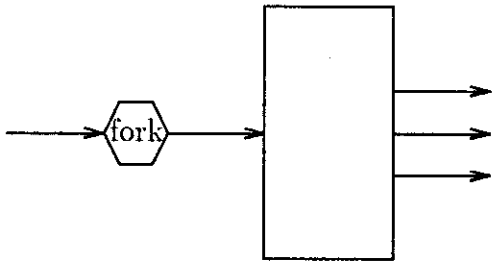


Figure 5.19: Fork node using datasets

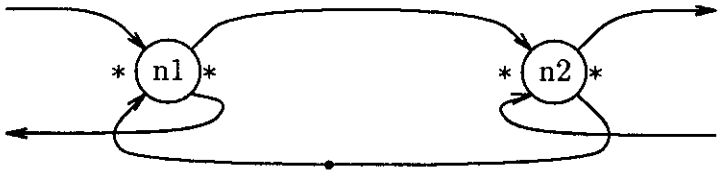


Figure 5.20: Scheme to incorporate explicit acknowledgments

5.9 Summary

This chapter demonstrates different ways in which $\nu\mathcal{FP}$ can be used during the design process. It first details the methods and conditions (section 5.2) under which $\nu\mathcal{FP}$ can be integrated into the SARA design system as an interpretation domain language for the GMB. By itself, this has two major benefits. The first benefit is that it provides the GMB with a concurrent interpretation domain language. This language is formal and analyzable enough that it can be used to prove properties about the data domain transformations it specifies. The second benefit is that, to the extent that this dissertation demonstrates a method for synthesizing $\nu\mathcal{FP}$ expressions, it provides a path for the realization of GMBs in silicon.

The introduction of $\nu\mathcal{FP}$ as an interpretation domain language in the GMB permits a procedure for formally proving refinement in the GMB during the process of design. Providing designers with an equivalence between GMB templates and $\nu\mathcal{FP}$ expressions allows them to move smoothly between the GMB's different domains. They can thus choose the most appropriate domain to model the items of interest and still be able to transform the representation into another domain later when their focus changes. The transformation can be carried out mechanically without having to re-verify that the resultant GMB still maintains the original semantics. These transformations also can be used in proving that refined GMB models have behavior equivalent to the unrefined ones (section 5.6). The opposite process is used for abstraction (section 5.7).

Attempting to incorporate a small, formal language like $\nu\mathcal{FP}$ into a richer, unstructured system like the GMB highlighted problem areas in both. On the GMB side, it showed that there are still some aspects of the GMB semantics that are not sufficiently formal or well understood. Appendix B clarifies some of these aspects. Also, the proposed methods for refinement show that restricting the types of GMBs used can lead to more formal and hence provable structures. On the $\nu\mathcal{FP}$ side, this research showed the limitations of an applicative language in representing concepts like mutual exclusion and non-determinism.

Chapter 6

Applicative Languages in the Synthesis of Algorithms

Most approaches of performing high-level synthesis from behavior to structure [68] use some kind of directed graph as an intermediate form to represent behavior. Sometimes, more than one graph is used to model different aspects of the behavior. Synthesis consists of mapping this graph into an implementation via function scheduling and resource allocation. Hardware modules (e.g., functional units, registers, and memory) and communication paths (e.g., buses, switches) between the modules are allocated to the nodes and edges of the data flow graph respectively [95, 72, 96, 97, 98]. In order to implement the control structure, the nodes of the control graph are mapped onto states of a control automaton or to instances in time. This is called scheduling. Instead of scheduling the control graph only in time, researchers have also looked at scheduling in space-time [74, 75]. This results in formal approaches that can be used to reason about the behavior as a specification and as implementation.

$\nu\mathcal{FP}$ takes a transformational approach to synthesis. Transformations have been used for the synthesis of combinational circuits before. $\nu\mathcal{FP}$ extends the domain of transformational synthesis to sequential circuits using space/time duality.

Recall that the behavior and structure are described by the same expressions in $\nu\mathcal{FP}$. Thus, the same expression that describes the behavior can be viewed in a structural manner, resulting in the initial structural implementation. Starting from the initial solution, $\nu\mathcal{FP}$ expressions are then transformed using pre-defined and proven substitutions to generate a structural implementation that consists only of primitives. These primitives can be at any level. In the current system, the primitives are gates. Therefore, the synthesis produces a binary-level logic design.

Next, *space synthesis* is performed. This consists of generating planar topology via symbolic

evaluation of the $\nu\mathcal{FP}$ expression. If direct implementation takes too much space, parts thereof can be transformed via space/time transformations into equivalent sequential implementations. This is called *time synthesis.* If this is satisfactory, the descriptions are converted to a form acceptable to a back-end system that performs the layout.

6.1 Overview of the Synthesis Process

This section provides an overview of the synthesis process in $\nu\mathcal{FP}$ and specifies exactly which $\nu\mathcal{FP}$ constructs may be used at which point in the synthesis process.

6.1.1 Classification of $\nu\mathcal{FP}$ Constructs

The set of all $\nu\mathcal{FP}$ constructs can be organized into three subsets. The first is the set of all available $\nu\mathcal{FP}$ functions and forms. The second subset is the subset that is implementable. The third is the subset of the second that are actually used in the generated topological layout.

All $\nu\mathcal{FP}$ functions: This is the full set of $\nu\mathcal{FP}$ functions and functional forms described in chapter 2. This set is as powerful in describing algorithms as FP. These functions can be used to specify algorithms at the high level.

Implementable $\nu\mathcal{FP}$ functions: Not all $\nu\mathcal{FP}$ functions can be implemented because of the method (symbolic evaluation) used in the system for extracting topology. Functions whose output structure depends on input value (like `iota` and `pick`) cannot be used for generating topological layouts. In addition, the description of the algorithm must be such that the predicate of any conditional is evaluable in terms of structure only. Also, the second and third arguments of a `sw` must have the same structure. All other functions are implementable.

Layout Management Functions: Out of the set of implementable functions, the functional forms are only used to guide the layout process and do not have any physical realization themselves. They are only used to specify the connectivity between the modules corresponding to their arguments. Only the set of functions that are implementable and are not layout management functions (i.e., primitives) are used for the physical layout.

6.1.2 The Steps in Synthesis

The synthesis process consists of the following steps.

1. Specify the algorithm in terms of the behavior.

2. Debug the specification using test data until you are convinced that the specification represents your intent.

3. Generate a top-level structural description. This is essentially a trivial step because, in $\nu\mathcal{FP}$, structure and behavior have the same description.

4. Transform the description, using substitutions, to successively lower levels of abstraction until the modules used are those that are implementable. This means that all un-implementable functions must be transformed out. For example, an `iota` could be used at a high level to model a counter, but at lower levels, it must be transformed into an implementation of a counter using other implementable primitives. Similarly, a `pick` can be transformed into a series of `sw`'es. Visual and performance feedback is used to guide this synthesis. This process may need iteration and modification of previous design steps.

5. Generate a direct space implementation topology. If this is satisfactory, then the topology can be fed to the back-end for mask layout generation. If this topology occupies too much space or if the implementation is supposed to work on sequential data, the designer must go to the next step.

6. Generate a time implementation of the algorithm using space/time transforms. Optimizations to the resultant description can be performed to meet the design goals.

7. Balance the delays in the circuit by moving all the inverse delays introduced in previous steps out to the edges of the circuit.

8. Generate a time-domain topology of the circuit.

9. Send the generated topology to the back-end system for physical layout generation.

6.2 Specification and Design of Algorithms

Designers first specify the algorithm in $\nu\mathcal{FP}$. They are free to choose whichever level is "best" for their current purposes. At a lower level of abstraction, where the structure of a function is to be considered, the definition given earlier (figure 2.2, page 28) would suffice to describe the behavior. For example, at some higher level of abstraction, the structure of the *FullAdder* may not be relevant. The *FullAdder* could be defined in terms of *HalfAdder*s. The algebra of

$\nu\mathcal{FP}$ programs may be used to reason about the algorithm. In addition, the specification may be executed with sample data to debug the program.

```
defun FullAdder(a,b,Cin)
      [1 org 2, 3] ∘ apndl ∘
      [1, HalfAdder ∘ [2,3]] ∘
      apndr ∘ [HalfAdder ∘ [a,b], Cin]
enddef

defun HalfAdder(a,b) [a andg b, a xorg b] enddef
```

Substitutions may be used to refine the program to whatever level of detail is required. In this way it is possible to specify the algorithm at a level of abstraction that is high enough to aid understanding and debugging and then refine it to the level necessary for implementation. In the current implementation, the circuit is transformed down to the gate level, and, in particular, until the only gates used in the circuits are those found in the cell library. There is no inherent reason to stop at this level. The framework allows the designer to stop refinement at either a lower (e.g., transistor) level or at a higher (e.g., macro-cell) level.

6.3 Space Domain Implementations of $\nu\mathcal{FP}$ Algorithms

A $\nu\mathcal{FP}$ algorithm can be mapped into a structure corresponding to a combinational network by passing symbolic inputs to functions which in turn generate symbolic outputs. The unit of information represented by a symbolic atom reflects objects at the level of abstraction. Thus, a symbolic atom may represent a wire, a set of wires, a bit vector, or an integer, as required. An acyclic computation graph with $\nu\mathcal{FP}$ primitives as nodes is obtained by tracing the application of a function to a symbolic input. This computation graph can be transformed into a layout in two steps.

1. The computation graph will contain many edges that go nowhere. This is because $\nu\mathcal{FP}$ is a strict language and therefore may compute intermediate results that are not needed for the computation of the final output. These dangling wires are removed using a pruning algorithm that works by traversing the graph backwards from the edge of the graph.

2. After this pruning, a one-dimensional compaction is performed to get the wires and modules as close together as possible.

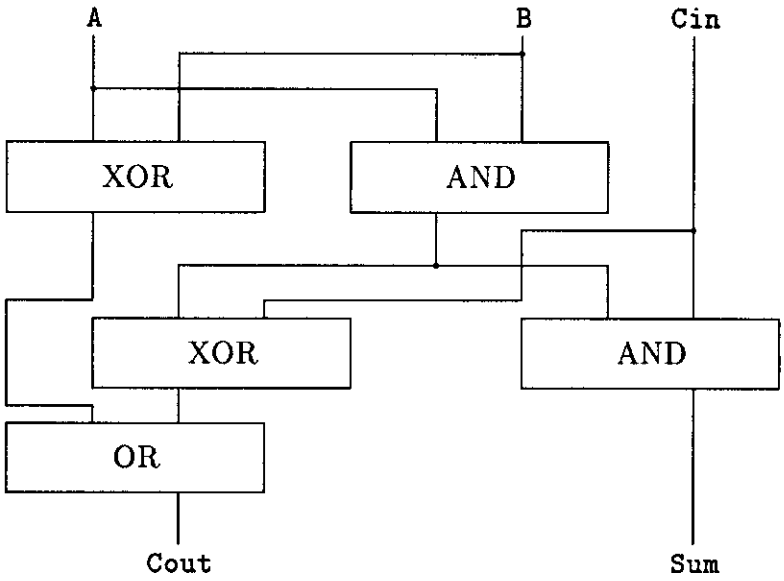


Figure 6.1: *FullAdder* with `andg`, `org`, `xorg`

Details of these techniques can be found in [55]. By tagging the appropriate functions (see chapter 4), the layout may be generated at any desired hierarchical level. For example, figure 6.1 shows a layout of a *FullAdder* using *and*, *or*, and *xor* gates. This corresponds to tagging each individual gate which is the default. Figure 6.2 shows the same *FullAdder* as being composed of *HalfAdder*s. This corresponds to tagging the *HalfAdder* and the *or* gate.

Structural iterations over the input of the circuit can be handled by the *insert* and *apply-to-all* functional forms. Other types of *structural recursions* are allowed in $\nu\mathcal{FP}$ since the *conditional* functional form is treated as a *structural* form for the purposes of layout. Figure 6.3 gives an example of a function that would not be possible to describe if only structural iterations were allowed. Depending on the value of the predicate of the conditional, either the consequent or the alternate part will be evaluated symbolically for its structure but no structure will be generated for the predicate part. A new primitive called `sw` (for switch) is provided in $\nu\mathcal{FP}$ which corresponds to the conditional form in μFP. This primitive takes three arguments. If the first is `true` then the output is the second argument; if it is `false` then the output is the third argument; else the result is $\perp$. In addition, it is required that the *structures* of the second and third arguments be the same.

A $\nu\mathcal{FP}$ description of a circuit can be generic in the sense that the description is independent of the input dimensions of the circuit. For example, there needs to be only one description of a decoder. This same description works for a decoder independent of the number of inputs.

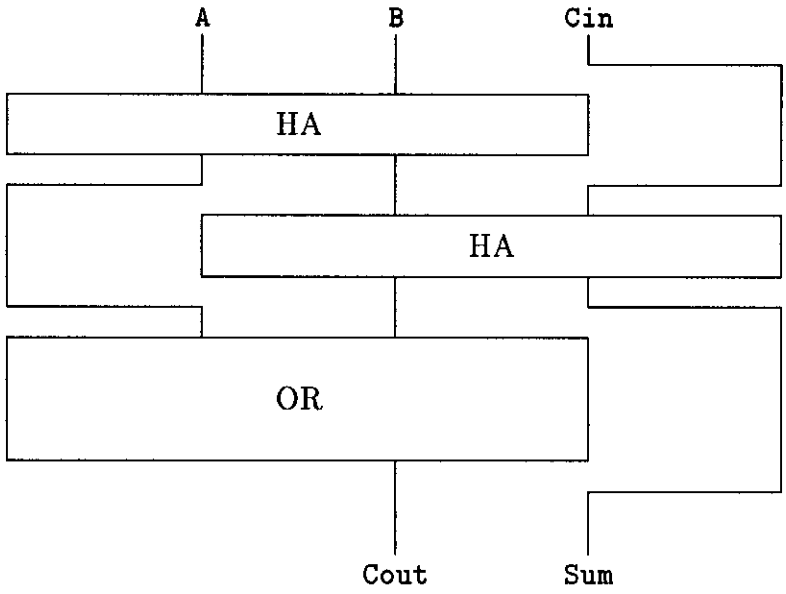


Figure 6.2: *FullAdder* with *HalfAdders*

```
defun gand
      if = ∘ [ length, %1]
      then 1
      else if = ∘ [ length, %2]
      then and2
      else if even ∘ length
      then gand ∘ & and2 ∘ pair
      else gand ∘ apndl ∘ [1, gand ∘ & and2 ∘ pair ∘ tl]
      fi fi fi
enddef
```

Figure 6.3: A generalized AND using structural recursion

The 3-to-8 decoder shown in figure 6.4 is obtained by evaluating the description of the generic decoder with a symbolic argument of size 3. Figure 6.5 shows how the generic iterative decoder is formed by first applying 1-to-2 decoders (*Dec1*) to the inputs and then inserting the function *DecStage*. *DecStage* takes an n-to-2^n decoder and a new input to make a $(n+1)$-to-2^{n+1} decoder.

```
defun Decoder !DecStage ∘ &Dec1 enddef

defun DecStage &andg ∘ concat ∘ &distl ∘ distr enddef

defun Dec1 [notg,id] enddef
```

As mentioned in chapter 4, these implementations may be evaluated to get speed/area estimates, but now, since routing is taken into account, a better estimate of area can be provided.

Cell iterative networks are combinational circuits which are formed by interconnecting a particular cell in a regular pattern. Although combinational circuits without feedback can be described in $\nu\mathcal{FP}$ using the forms inherited from FP, some additional functional forms are provided to give designers more control over exactly how cell iterative networks are to be laid out. These networks are thus readily described in $\nu\mathcal{FP}$ by invoking the form (corresponding to the interconnection pattern) on the function (corresponding to the cell). For example, figure 6.6 shows the **seq** functional form pictorially. Sometimes it is necessary to have two inputs to the function F at each stage and to have one of those inputs come in from the x direction and the other from the y direction. This is accomplished by the **seqxy** functional form shown in figure 6.7. Though both forms result in the same computation graph, their layout is different.

6.3.1 Restrictions

Though any function in $\nu\mathcal{FP}$ can be simulated with value inputs to debug or test it, not all $\nu\mathcal{FP}$ functions can be laid out. This is not because they are inadmissible in an applicative framework, but rather because $\nu\mathcal{FP}$ has chosen to extract structure from the $\nu\mathcal{FP}$ algorithm via the symbolic evaluation of the algorithm. Other methods for structure extraction (e.g., partial evaluation as used in μFP) may be able to relax these restrictions.

The mapping from a $\nu\mathcal{FP}$ algorithm to a combinational network is allowed under the following restrictions. Functions like `iota` or `pick`, whose output *structure* depends on its input *value*, cannot be laid out. This is because, during symbolic evaluation, `iota` will be passed a symbolic argument and it needs a *value* in order to determine the number of elements in (and hence the structure of) its output result. For the same reasons, the predicate part

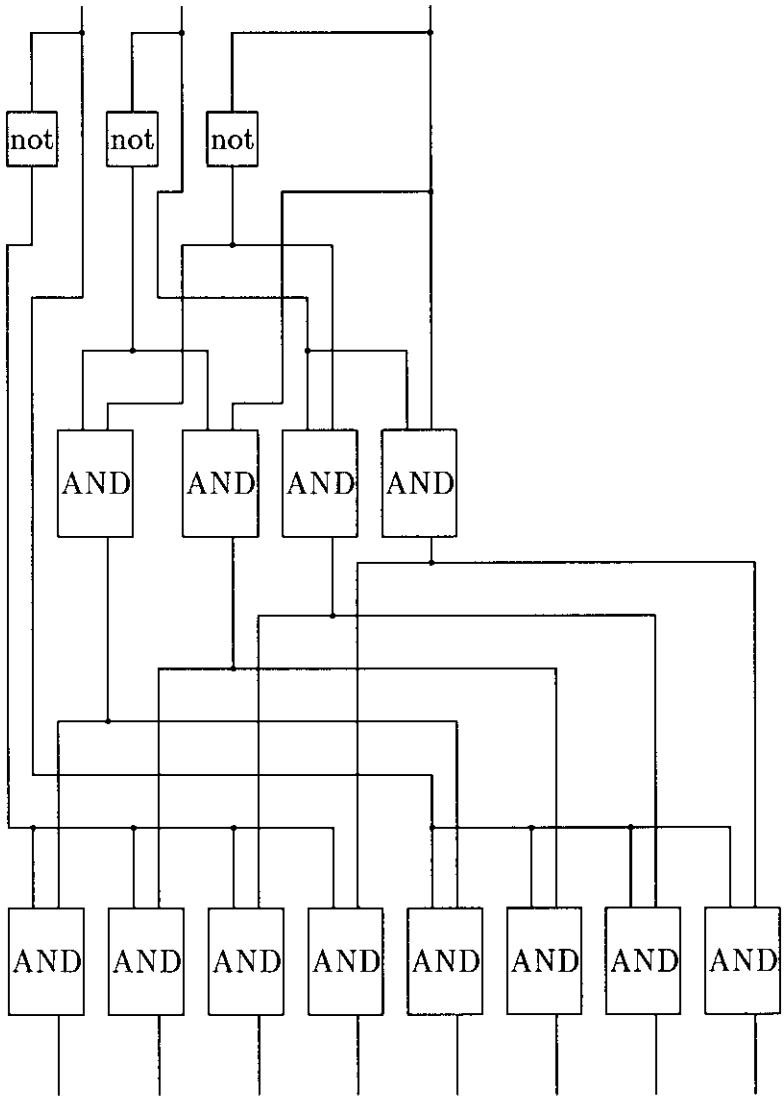


Figure 6.4: 3-to-8 Decoder

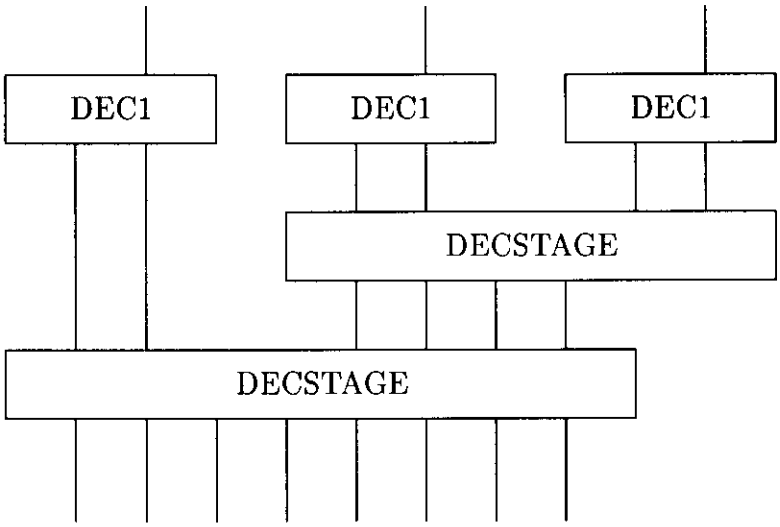


Figure 6.5: Generic Decoder

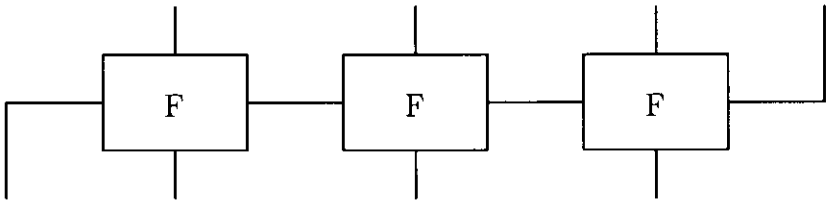


Figure 6.6: The **seq** form

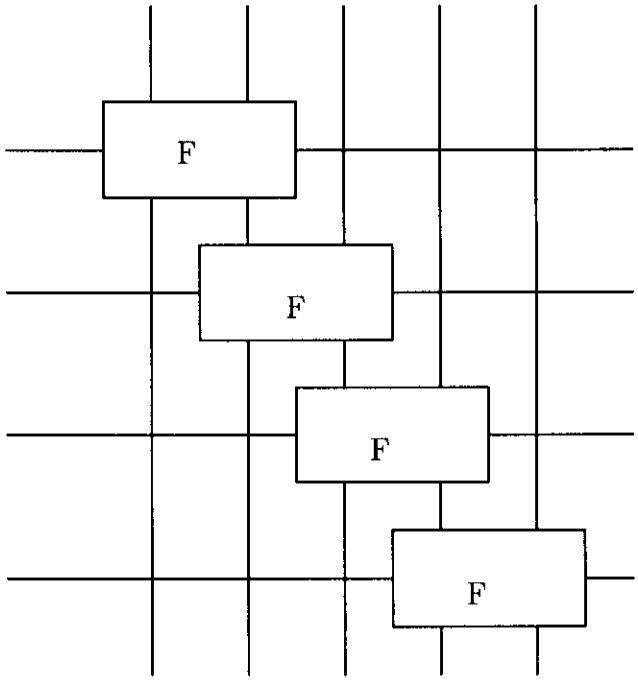


Figure 6.7: The **seqxy** form

of a conditional must be evaluable to a boolean. This implies that the predicate must be a structural predicate (i.e., it must be evaluable using only structural information). For example,

```
!  and  °& null
```

or

```
=  °[ length, %4 ]
```

are allowed predicates, whereas

```
=  °[ 3 , %4 ]
```

is not. The last restriction is that the second and third arguments of the **sw** must have the same structure.

6.4 Time Domain Implementations of $\nu\mathcal{FP}$ Algorithms

As was shown in section 6.3.1, virtually all $\nu\mathcal{FP}$ programs can be laid out. However, since iterations are unfolded in space, the resulting layout might occupy more area than is available. Also, the inputs to the circuit might be coming in serially along the same wire(s), rather than in parallel on separate wires. In both these cases, it is convenient to implement the circuit (or parts of it) as a sequential circuit rather than a combinational circuit. In $\nu\mathcal{FP}$ this is done by replacing certain space-domain functional forms by their time-domain duals.

$$
\begin{aligned}
\&f &\equiv \mathcal{D}^{-1} \circ \mathrm{POSI} \circ \&^T f \circ \mathrm{SOPI} \\
!f &\equiv \mathcal{D}^{-1} \circ !^T f \circ \mathtt{apndr} \circ [\mathrm{SOPI} \circ \mathtt{tlr}, \mathtt{last}] \\
\mathbf{seq}f &\equiv \mathcal{D}^{-1} \circ \mathtt{apndl} \circ [1, \mathrm{POSI} \circ \mathtt{tl}] \circ \mathbf{seq}^T f \circ \mathtt{apndr} \circ [\mathrm{SOPI} \circ \mathtt{tlr}, \mathtt{last}]
\end{aligned}
$$

In these equalities SOPI is a *parallel-out-serial-in* shift register, SOPI is a *serial-out-parallel-in* shift register, and $\&^T$, $!^T$ and $\mathbf{seq}^T$ are the time-domain duals of the corresponding space-domain functional forms introduced earlier. Figures 6.8 and 6.9 show the layout of the time domain duals of the *apply-to-all* and *right insert* functional forms respectively. The $!^T$ and $\mathbf{seq}^T$ forms use the first element of their input sequence as the initial value of the internal register (**REG**). This makes it possible to specify the initial state of a sequential system in $\nu\mathcal{FP}$. $\mathcal{D}^{-1}$ is a phantom element that corresponds to an inverse time delay. It is used to keep track of the number of clock pulses by which the output is going to be delayed. This information is needed by the *construct* functional form to synchronize its components since the semantics of the construct require that the outputs of its elements appear together. Generally

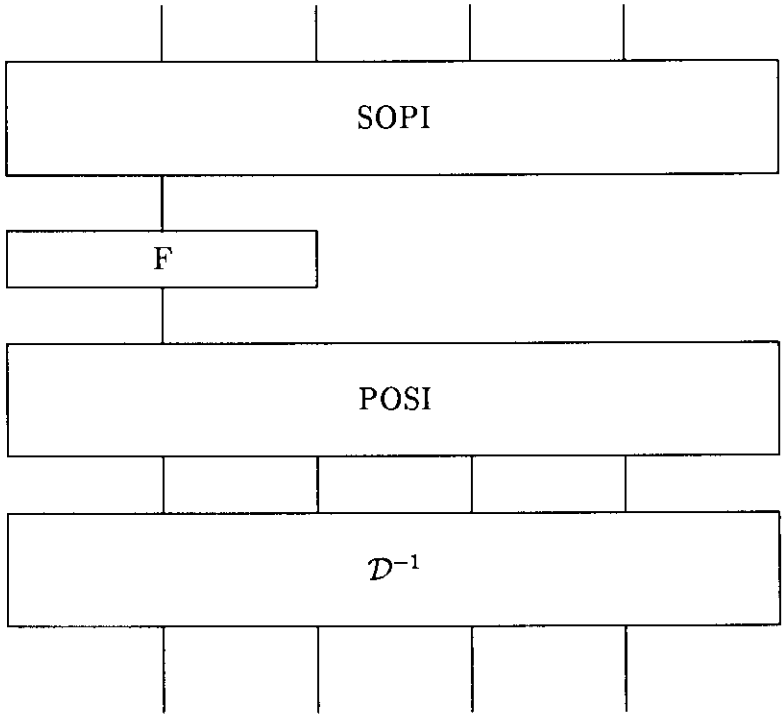


Figure 6.8: Time domain *apply-to-all*

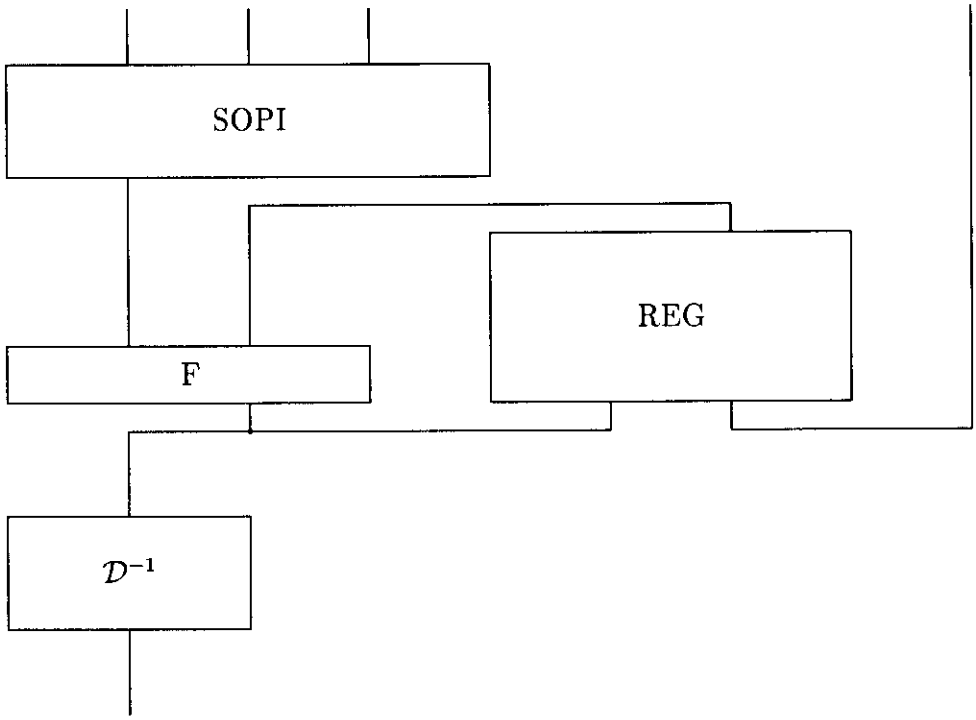


Figure 6.9: Time domain *right insert*

the $\mathcal{D}^{-1}$ elements are moved, via transformations, to the outputs of the circuit where they serve to denote the delay.

When elements of a sequence are available serially in time along the same wire(s), it is necessary to know when each element is valid. This is accomplished, during symbolic simulation, by having each symbolic item carry the name of a clock with it. It is assumed that its value will be stable before every tick of the named clock. The system will automatically widen the intervals between clock ticks to ensure that this is true. Initially, all the inputs are associated with the same clock. Each combinational element will assign to its output the clock associated with its input. If there are n elements to the input sequence of a SOPI, then each of its output elements will be clocked by the clock nC_k; and conversely for a POSI. A clock named nC_k denotes a clock which has n clock ticks in between consecutive ticks of the clock named C_k. Though the description of a SOPI or POSI is generic, the value of n (the number of elements in the sequence) must be known at layout time.

As an example, consider a time-domain implementation of an inner-product algorithm

$$! \quad + \circ \& \quad *$$

The straightforward implementation of the algorithm, using the equations given above, would result in the layout shown in figure 6.10. Since there are two $\mathcal{D}^{-1}$ elements in the layout, the output will be delayed by two clock ticks from the input.

However, using the identities

$$\mathcal{D}^{-1} \circ \mathrm{POSI} \circ f \circ \mathrm{SOPI} \equiv \mathtt{apndr} \circ [\mathcal{D}^{-1} \circ \mathrm{POSI} \circ \&^T f \circ \mathrm{SOPI} \circ \mathtt{tlr}, f \circ \mathtt{last}]$$
$$f \circ \mathcal{D}^{-1} \equiv \mathcal{D}^{-1} \circ f$$

$$\mathrm{POSI} \circ \mathcal{D}^{-1} \circ \mathrm{SOPI} \equiv \mathrm{SOPI} \circ \mathcal{D}^{-1} \circ \mathrm{POSI} \equiv \mathtt{id}$$
$$[\mathtt{tlr}, \mathtt{last}] \circ \mathtt{apndr} \equiv \mathtt{apndr} \circ [\mathtt{tlr}, \mathtt{last}] \equiv \mathtt{id}$$

the program may be transformed into the following

$$\mathcal{D}^{-1} \circ !^T + \circ \mathtt{apndr} \circ [\&^T * \circ \mathrm{SOPI} \circ \mathtt{tlr}, * \circ \mathtt{last}]$$

whose layout is shown in figure 6.11. The single $\mathcal{D}^{-1}$ element denotes that the output is delayed only one clock tick from the input.

Note, however that there is an extra multiplier used only once at initialization. Domain knowledge can be used here to eliminate this multiplier. Since 0 is the identity for addition,

$$! \quad + \circ \& \quad * \equiv ! \quad + \circ \& \quad * \circ [\ \mathtt{apndr} \circ [\ \mathtt{id}, \ \%0, \ \%0]\ ]$$

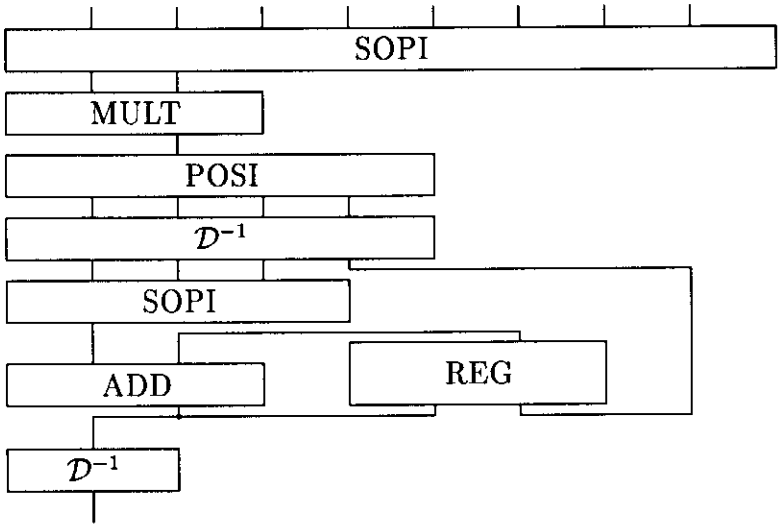


Figure 6.10: Inner Product

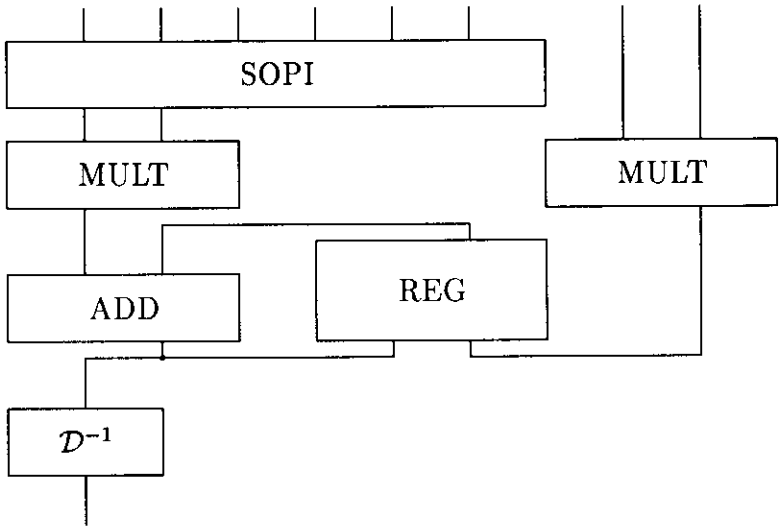


Figure 6.11: Improved Inner Product

It is easy to show that

$$[\ f \circ \texttt{tlr},\ g \circ \texttt{last}]\ \circ \texttt{apndr} \circ [\ p,\ q\ ] \equiv [\ f \circ p,\ g \circ q\ ]$$

The above equations and the fact that

$$* \circ [\ \%0,\ \%0\ ] \equiv \%0$$

can be used to further improve the inner product algorithm into

$$\mathcal{D}^{-1} \circ !^T + \circ \texttt{apndr} \circ [\&^T * \circ \text{SOPI} \circ \texttt{tlr}, \%0]$$

which does not require an extra multiplier. This exercise is a very good example of the benefit of incorporating initial conditions into the framework. The ability to reason about and manipulate the initial value of the register allowed us to optimize out the multiplier.

6.4.1 Restrictions and Extensions

This implementation accepts all its inputs simultaneously and eventually gives its result. It will only work for input sequences of one particular length, since only fixed size SOPIs can be laid out. However, if each element of the input sequence was input serially to the implementation, a corresponding POSI could be introduced at the input and then used to transform out the SOPI that exists in the current implementation. This would make the implementation generic in the sense that it would be able to handle arbitrary length sequences as its inputs.

Applying space/time transformations to a purely combinational circuit introduces $\mathcal{D}^{-1}$ elements into the description. Obviously, these cannot be implemented. Therefore they have to be transformed out to the edges of the circuit where they represent the latency of the circuit. Care must be taken to make sure that the delays in each arm of a *construct* are balanced (i.e., each arm of the *construct* must have the same delay).

6.5 Formalizing Sequential Systems in $\nu\mathcal{FP}$

Section 6.4 shows how space-domain implementations may be replaced by their time-domain duals using space/time transformations. This section establishes a formal basis for talking about time in an applicative framework and provides a method for proving the correctness of the transformations in section 6.4. An example demonstrates how such a proof may be carried out. Other useful identities are listed.

6.5.1 Formal Domains

Assume that each $\nu\mathcal{FP}$ object is tagged with an index. This index is taken from the set of all Dewey-decimal numbers. To be more formal, the index set, $\mathcal{I}$, is the the set of all finite sequences of natural numbers.

$$\mathcal{I} \equiv \bigcup_{n \in \mathbf{N}} \{\langle d_0, d_1, \ldots, d_n \rangle \mid 0 \leq i \leq n \wedge d_i \in \mathbf{N}\}$$

Assume that $i \in \mathcal{I}$ is a typical index.

$$i = \langle d_0, d_1, \ldots, d_n \rangle,\ d_k \in \mathbf{N}$$

Then, we can also represent i as the Dewey decimal number

$$i = d_0.d_1 \ldots d_n$$

The tag of a $\nu\mathcal{FP}$ object is represented by a subscript on the left hand side of the object. As a short-hand, if all the elements of a sequence are tagged by the same index, then the sequence itself is tagged by that index and the indices are removed from the individual elements.

$${}_i\langle x_n, \ldots, x_0 \rangle \equiv \langle {}_i x_n, \ldots, {}_i x_0 \rangle$$

These indices will be used to specify temporal sequencing among $\nu\mathcal{FP}$ objects. The indices have a natural alphabetic sort defined on them. This ordering specifies the temporal ordering of the objects that the indices tag.

$$\begin{array}{ll} i < i.j & \forall i \in \mathcal{I};\ j \in \mathbf{N} \\ i.j < i.k & \forall i \in \mathcal{I};\ j,k \in \mathbf{N};\ j < k \end{array}$$

In particular, note that

$$i < i.0$$

All objects in FP can be deemed to be tagged by just 0. This is the base case. If a particular $\nu\mathcal{FP}$ object O_1 is tagged with an index i_1 and another object O_2 is tagged with i_2 and $i_1 < i_2$ then it means that O_1 will be evaluated before O_2. To be more precise

$$i_1 < i_2 \Longrightarrow \text{time}({}_{i_1}O_1) < \text{time}({}_{i_2}O_2)$$

6.5.2 Time Primitives and Axioms

Let the following be defined as axioms:

$$\begin{aligned}
\mathrm{SOPI}_{n/k} : {}_i\langle X_n, \ldots, X_0\rangle &\equiv \langle {}_{i.n+k}X_?, \ldots, {}_{i.n}X_n, \ldots, {}_{i.0}X_0\rangle \\
\mathrm{POSI}_{n/k} : \langle {}_{i.n+2k}X_?, \ldots, {}_{i.n+k}X_n, \ldots, {}_{i.k}X_0\rangle &\equiv {}_{i+1}\langle X_n, \ldots X_0,\rangle \\
\mathcal{D} : {}_{i.j}X &\equiv {}_{i.j+1}X \\
i &\in \mathcal{I} \\
j, n &\in \mathbf{N}
\end{aligned}$$

$\mathrm{SOPI}_{n/k}$ is a function that converts a space sequence to a time sequence. In the case where $k = 0$, the function is represented by just SOPI. In this situation, each element of the space sequence is available at the same spatial location but at increasing moments in time. If $k \neq 0$ then it means that the time sequence is padded with k arbitrary elements, $X_?$, at the end. It should be noted that $\mathrm{SOPI}_{n/k}$ is only valid when applied to a sequence and results in a sequence in which the index of each element of the sequence is incremented by another Dewey-decimal level.

In a similar fashion, $\mathrm{POSI}_{n/k}$ is a converter from time to space. When $k = 0$, an unadorned POSI is used. The space sequence will be available after all the elements of the time sequence have arrived and so the outer Dewey-decimal index is incremented. In the case when $k \neq 0$, though it will accept $n + k$ elements, only the first n of them are used for the output. In addition, the first input element is skewed by a delay of k.

The delay operator, $\mathcal{D}$, defined above is used to define the *inverse delay* operator, $\mathcal{D}^{-1}$. In addition, note that both $\mathcal{D}$ and $\mathcal{D}^{-1}$ work on single elements or sequences. Obviously, an inverse delay cannot be implemented, but it will be useful in transformations and will provide a notion of the latency of the implemented circuit.

$$\begin{aligned}
\mathcal{D} : \langle X_n, \ldots, X_0\rangle &\equiv \langle \mathcal{D} : X_n, \ldots, \mathcal{D} : X_0\rangle \\
\mathcal{D}^k &\equiv \underbrace{\mathcal{D} \circ \ldots, \mathcal{D}}_{k \text{ times}} \\
\mathcal{D}^{k+1} \circ \mathcal{D}^{-1} &\equiv \mathcal{D}^k \\
\mathcal{D}^{-k} &\equiv \underbrace{\mathcal{D}^{-1} \circ \ldots, \mathcal{D}^{-1}}_{k \text{ times}} \\
\mathcal{D}^{-1} : \langle X_n, \ldots, X_0\rangle &\equiv \langle \mathcal{D}^{-1} : X_n, \ldots, \mathcal{D}^{-1} : X_0\rangle
\end{aligned}$$

6.5.3 Example Proof

Given the axioms above and the definition

$$\&^T f : \langle {}_nX_n, \ldots, {}_0X_0\rangle \equiv \langle {}_nf : X_n, \ldots, {}_0f : X_0\rangle \equiv \langle {}_nY_n, \ldots, {}_0Y_0\rangle$$

we prove the following space-time transformation for the *apply-to-all* functional form. $\&^T$ is the time-domain equivalent of the space-domain "&" form.

$$\&\&f \equiv \mathcal{D}^{-1} \circ \text{SOPI} \circ \mathcal{D}^{-1} \circ \text{SOPI} \circ \&^T f \circ \text{POSI} \circ \text{POSI}$$

The proof makes reference to figure 6.12. Let the input (point 1 in figure 6.12) be

$$_i\langle\langle X_{n,m}, \ldots, X_{n,0}\rangle, \ldots, \langle X_{0,m}, \ldots, X_{0,0}\rangle\rangle$$

After the first SOPI (point 2) it gets transformed to

$$\langle_{i.n}\langle X_{n,m}, \ldots, X_{n,0}\rangle, \ldots, _{i.0}\langle X_{0,m}, \ldots, X_{0,0}\rangle\rangle$$

After the second SOPI (point 3) it is

$$\langle\langle_{i.n.m} X_{n,m}, \ldots, _{i.n.0} X_{n,0}\rangle, \ldots, \langle_{i.0.m} X_{0,m}, \ldots, _{i.0.0} X_{0,0}\rangle\rangle$$

After the function $\&^T f$ (point 4) it becomes

$$\langle\langle_{i.n.m} Y_{n,m}, \ldots, _{i.n.0} Y_{n,0}\rangle, \ldots, \langle_{i.0.m} Y_{0,m}, \ldots, _{i.0.0} Y_{0,0}\rangle\rangle$$

After the first POSI (point 5) the result is

$$\langle_{i.n+1}\langle Y_{n,m}, \ldots, Y_{n,0}\rangle, \ldots, _{i.0+1}\langle Y_{0,m}, \ldots, Y_{0,0}\rangle\rangle$$

After the first $\mathcal{D}^{-1}$ (point 6) it is

$$\langle_{i.n}\langle Y_{n,m}, \ldots, Y_{n,0}\rangle, \ldots, _{i.0}\langle Y_{0,m}, \ldots, Y_{0,0}\rangle\rangle$$

Passing it through the second POSI (point 7) results in

$$_{i+1}\langle\langle Y_{n,m}, \ldots, Y_{n,0}\rangle, \ldots, \langle Y_{0,m}, \ldots, Y_{0,0}\rangle\rangle$$

Finally, after the second $\mathcal{D}^{-1}$ (point 8) it is

$$_i\langle\langle Y_{n,m}, \ldots, Y_{n,0}\rangle, \ldots, \langle Y_{0,m}, \ldots, Y_{0,0}\rangle\rangle$$

which is the same result as passing the original sequence through $\&\&f$.

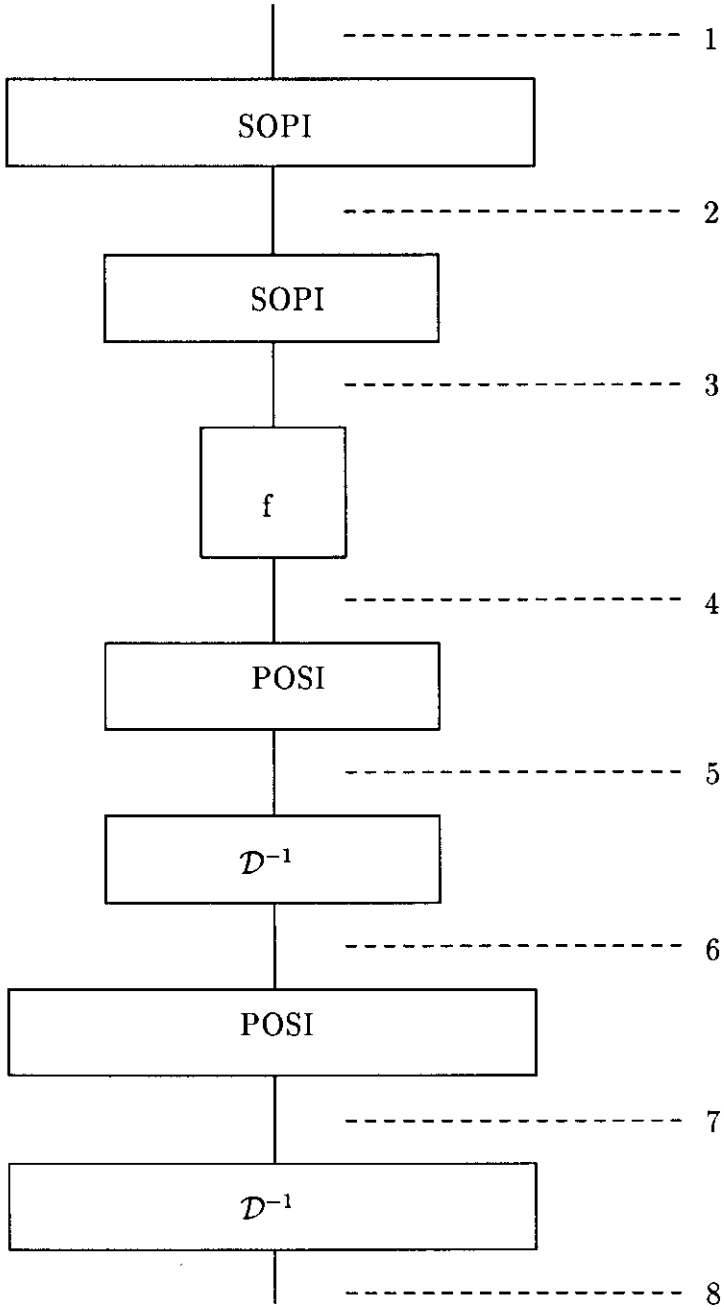


Figure 6.12: Nested Time-domain Apply-to-all

6.5.4 Identities

Some other identities that can be proved similarly are

$$\begin{aligned} id &\equiv \mathcal{D}^{-1} \circ \text{POSI} \circ \text{SOPI} && (6.1)\\ &\equiv \text{SOPI} \circ \mathcal{D}^{-1} \circ \text{POSI} && (6.2)\\ &\equiv \text{POSI} \circ \text{SOPI} \circ \mathcal{D}^{-1} && (6.3)\\ &\equiv \mathcal{D} \circ \mathcal{D}^{-1} && (6.4) \end{aligned}$$

Let f_k be a function that delays its output by k,

$$_{i.j+k}Y \equiv f_k : {}_{i.j}X$$

Then, we can show that

$$\text{POSI} \circ \mathcal{D}^{-k} \circ \&^T f_k \circ \text{SOPI} \equiv \text{POSI}_{n/k} \circ \&^T f_k \circ \text{SOPI}_{n/k} \quad (6.5)$$

The proof is straightforward. First, consider the RHS. Let a generic input sequence be

$$_i\langle X_n, \ldots, X_0 \rangle$$

Passing this sequence through $\text{SOPI}_{n/k}$ yields

$$\langle {}_{i.n+k}X_?, \ldots, {}_{i.n}X_n, \ldots, {}_{i.0}X_0 \rangle$$

by the definition of $\text{SOPI}_{n/k}$. f_k converts this sequence to

$$\langle {}_{i.n+2k}Y_?, \ldots, {}_{i.n+k}Y_n, \ldots, {}_{i.k}Y_0 \rangle$$

By the definition of $\text{POSI}_{n/k}$, this results in

$$_{i+1}\langle X_n, \ldots, X_0 \rangle$$

which is the same result that would be produced by the LHS on the same input sequence.

This theorem is useful because it allows us to absorb $\mathcal{D}^{-1}$s into SOPI/POSI pairs and thus eliminate them from the circuit. In theory, this transformation could be applied only in the case when it was proven that the function f_k delayed its output by k. However, in practice, since the user is not allowed to introduce arbitrary SOPIs or POSIs in the program, it will always be the case. An earlier method of converting space sequences to time sequences was reported by Meshkinpour in [99, 64].

6.6 Interface to Layout Tools

The preceding sections have shown how combinational and sequential circuits are generated from $\nu\mathcal{FP}$ descriptions. This section describes one method of converting the topology derived in section 6.3 to descriptions acceptable to layout tools. Wu, in [100], describes a different approach to interfacing the topological description to the VIVID [101, 102] layout system. The symbolic grid paradigm of VIVID is a good match to the topological information provided by the $\nu\mathcal{FP}$ system. This section describes another approach, using the same information, but interfacing to the LagerIV [47, 103] silicon assembly system. The tool of interest in the LagerIV system is the `flint` layout generator [103]. `flint` takes a net-list description of the connectivity of a circuit and then places and routes the connections automatically. It also has capabilities for allowing human interaction to improve the placement of cells and routing of nets. The connectivity information is provided in a language called SDL (Structure Description Language) and the placement and routing information is specified in a language called FDL (Floorplan Description Language). In the absence of a FDL description, `flint` is capable of generating a placement, but this placement often leaves a lot to be desired. With the topological information available from a $\nu\mathcal{FP}$ description it is possible to provide placement information to `flint` that improves its placement efficiency.

6.6.1 Cross-sections

The topological information derived from a $\nu\mathcal{FP}$ description consists of a list of *cross-sections*. Each cross-section consists of a list of *cross-section elements*. This section provides an overview of the format using an example. The complete description is provided in [55].

A cross-section is a horizontal slice of a topological description. Each cross-section element can be one of three types:

1. *A free wire* is a symbol other than `$`, `*`, `+`, or `^`

2. *A crossing* of the form

   ```
   (* w * u1 u2 ... un)
   ```

   where the wire `w` is a horizontal track that crosses vertical wires. At least one of the `u`'s must be a `+` or a `^`. `+` denotes an upper connection to the horizontal track, and `^` denotes a lower connection to the track. `u`'s that are not `^` or `+` denote wires that feed through the cross section. For example,

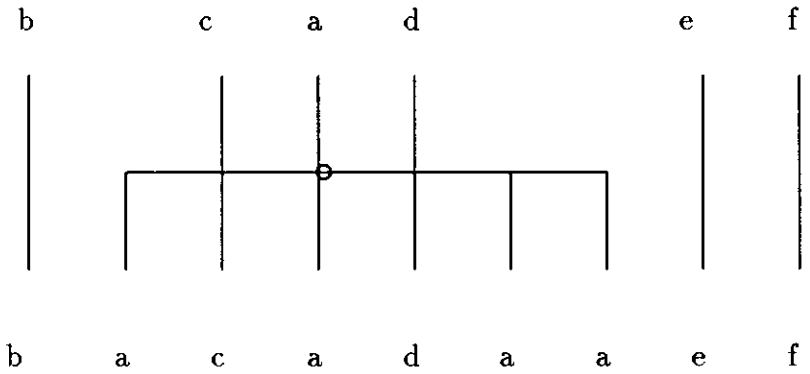


Figure 6.13: A *crossing* cross-section element

```
(* a * b ^ c + ^ d ^ ^ e f)
```

represents the crossing shown in figure 6.13.

3. *A cell* of the form

```
($ level $ ht uid label wd $ i1 i2 ... in $ o1 o2 ... op)
```

where `level` is one of `f` (first), `i` (intermediate), or `l` (last) cross-sections of a cell. `level` could also be `b` in the case where the cross-section includes the whole cell. `ht`, `uid`, `label`, and `wd` are the height, unique identifier, label, and width of the cell respectively. The `i`'s and `o`'s are the ordered inputs and outputs of the cell.

Figure 6.14, adapted from [55], shows the layout of an XNOR gate generated from the following $\nu\mathcal{FP}$ expression

```
defun xnor
      notg ° nandg ° & nandg
       °  [[1,2],[2,3]]  °  [1,nandg,2]
enddef
```

and whose cross-sectional representation is given in figure 6.15.

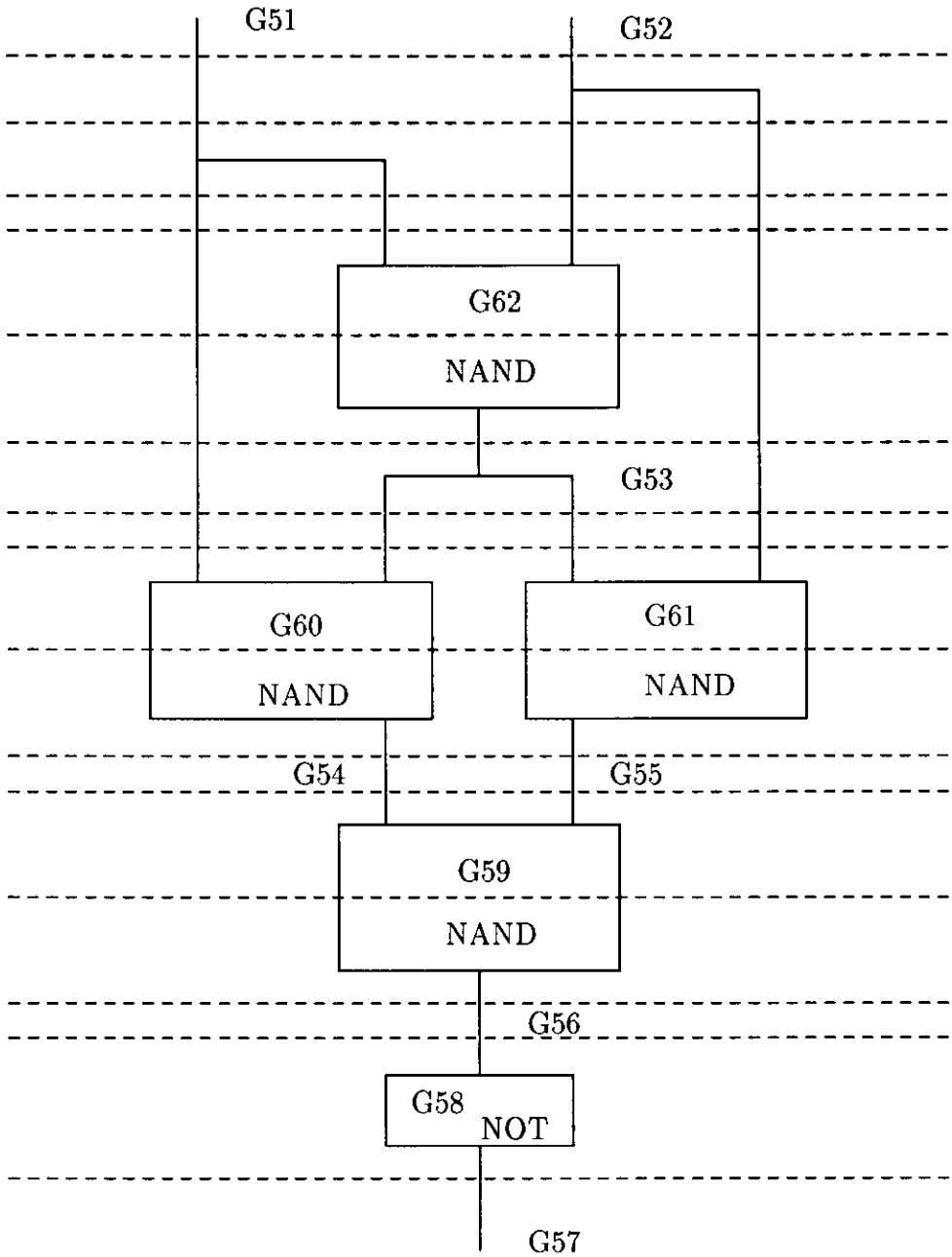


Figure 6.14: Layout of a XNOR gate

```
((G00051 G00052)
 (G00051 (* G00052 * + ^ ^))
 ((* G00051 * + ^ ^) G00052 G00052)
 (G00051 G00051 G00052 G00052)
 (G00051 ($ F 2 G00062 NAND 3 $ G00051 G00052 $ G00053 $)
         G00052)
 (G00051 ($ L 2 G00062 NAND 3 $ G00051 G00052 $ G00053 $)
         G00052)
 (G00051 (* G00053 * + ^ ^) G00052)
 (G00051 G00053 G00053 G00052)
 (($ F 2 G00060 NAND 3 $ G00051 G00053 $ G00054 $)
  ($ F 2 G00061 NAND 3 $ G00053 G00052 $ G00055 $))
 (($ L 2 G00060 NAND 3 $ G00051 G00053 $ G00054 $)
  ($ L 2 G00061 NAND 3 $ G00053 G00052 $ G00055 $))
 (G00054 G00055)
 (($ F 2 G00059 NAND 3 $ G00054 G00055 $ G00056 $))
 (($ L 2 G00059 NAND 3 $ G00054 G00055 $ G00056 $))
 (G00056)
 (($ B 1 G00058 NOT 2 $ G00056 $ G00057 $))
 (G00057))
```

Figure 6.15: Cross-sections of a XNOR gate

6.6.2 Generating the Connectivity Information

It is now easy to extract the netlists from the cross-sectional representation. The method is to look at only the cross-section elements that refer either to `f` or `b` levels (i.e., the inputs to cells). For each such element, add the inputs and outputs of this cell to the appropriate netlists. For example, consider line 5 of figure 6.15 corresponding to the fifth cross-section of figure 6.14. Traversing this cross-section shows that the NAND gate labeled `G00062` has `G00051` and `G00052` as inputs and `G00053` as an output. Hence, `(G00062 IN1)`, `(G00062 IN2)`, and `(G00062 OUT1)` are added to the netlists for `G00051`, `G00052`, and `G00053` respectively. The power and ground connections for each cell also have to be provided. In addition, it is a simple matter to deduce the inputs and outputs of the composed circuit by using the first and last cross-section respectively. The SDL file generated using this algorithm is shown in figure 6.16.

There is one detail that needs to be taken care of. In many instances, there will be cells that have feed-through wires in them. The most common example is that of a one-bit decoder shown in figure 6.17, whose $\nu\mathcal{FP}$ description is

```
defun onedec [id,notg] enddef
```

In such cases, because of symbolic evaluation, the label of an output of the cell will be the same as one of the input labels. The correctness of the netlist extraction algorithm mentioned above depends on the input and output labels of a cell being distinct. This problem can be avoided by making sure that such a case does not arise. The simplest way of achieving this is to force the system into generating a new label for each output by passing each output through an AND gate whose second input is set to `true` shown in figure 6.18 and whose $\nu\mathcal{FP}$ description is shown below.

```
defun onedec & andone° [id,notg] enddef

defun andone andg° [id,%1] enddef
```

Since only the *onedec* is tagged, the symbolic interpreter will only generate topology for it and will not generate any for *andone*. In the case of SOPI's and POSI's, the *andone* has to be on the inputs and not on the outputs.

6.6.3 Generating the Placement Information

Generating the placement information required by `flint` is only slightly more complex than generating the SDL file. Each cross-section is first collapsed into a list of cells and channels. Because of the information already in the cross-section element, it is easy to figure out in which

```
(parent-cell "xnor" )
(layout-generator Flint11)
(subcells ( NOT G00058 ) ( NAND G00059 )
  ( NAND G00061 ) ( NAND G00060 ) ( NAND G00062 ))

(net G00056 (NETTYPE SIGNAL) ( (G00059 OUT1) (G00058 IN1)))
(net G00055 (NETTYPE SIGNAL) ( (G00061 OUT1) (G00059 IN2)))
(net G00054 (NETTYPE SIGNAL) ( (G00060 OUT1) (G00059 IN1)))
(net G00053 (NETTYPE SIGNAL) ( (G00062 OUT1) (G00060 IN2)
       (G00061 IN1)))
(net G00057 (NETTYPE SIGNAL) ( (parent OUT1) (G00058 OUT1)))
(net G00052 (NETTYPE SIGNAL) ( (parent IN2)  (G00062 IN2)
       (G00061 IN2)))
(net G00051 (NETTYPE SIGNAL) ( (parent IN1)  (G00062 IN1)
       (G00060 IN1)))

(net Vdd! (NETTYPE SUPPLY) ( (parent Vdd!) (G00058 Vdd!)
   (G00059 Vdd!) (G00061 Vdd!) (G00060 Vdd!) (G00062 Vdd!)))
(net GND! (NETTYPE SUPPLY) ( (parent GND!) (G00058 GND!)
   (G00059 GND!) (G00061 GND!) (G00060 GND!) (G00062 GND!)))
(terminal OUT1 (TERM_EDGE BOTTOM) (TERMTYPE SIGNAL))
(terminal IN2 (TERM_EDGE TOP) (TERMTYPE SIGNAL))
(terminal IN1 (TERM_EDGE TOP) (TERMTYPE SIGNAL))
(terminal Vdd! (TERMTYPE SUPPLY))
(terminal GND! (TERMTYPE GROUND))
(end-sdl)
```

Figure 6.16: SDL of XNOR gate

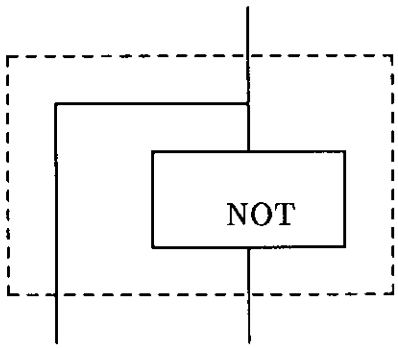


Figure 6.17: One-bit decoder with feed-through

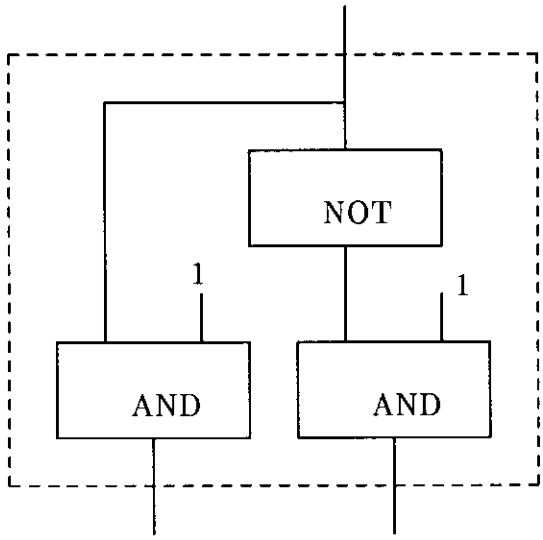


Figure 6.18: Modified one-bit decoder

cross-section a cell starts and in which one it ends. Since this information is not directly available for channels, it is derived by examining adjacent cross-sections. First, the corresponding cells of adjacent cross-sections are lined up. Consider two consecutive cells that match. If there is a channel between them in both cross-sections then that channel is a continuing channel. If there were no channels between them in the previous cross-section, but there is a channel in the current cross-section, then it is the start of a new channel. If there were a channel between them in the previous cross-section, but there is no channel between them in the current cross-section, then the old channel has ended. Now that it is determined where the channels start and end, another pass through all the cross-sections can be used to generate the adjacency information required by FDL. In practice, these three passes can be merged into one.

6.7 Generating a Layout

Once the connectivity and placement is specified, the LagerIV tool `flint` is used to perform the final placement and routing of the connections. At the end of this stage, a file suitable for Magic [3] is produced. If everything is satisfactory at this stage, the mask layout information is produced in Caltech Intermediate Form (CIF) [104]. The CIF layout for the XNOR gate is shown in figure 6.19. Its correspondence to the topology in figure 6.14 should be obvious.

6.8 An Example: A Conditional-Sum Adder

A conditional-sum adder [105] is a fast adder that reduces carry propagation delays by simultaneously generating conditional sums corresponding to both input carry values. When the distant carries have been determined, they are used to select either one of the two conditional sums resulting in the true sum. This treatment is adapted from [100] which should be consulted for details.

The conditional-sum adder takes an input of the form

$$\langle\langle x_{n-1}, \ldots, x_0\rangle, \langle y_{n-1}, \ldots, y_0\rangle, c_{-1}\rangle$$

and returns a result

$$\langle c_n, s_{n-1}, \ldots, s_0\rangle$$

The addition is performed in three steps:

1. Arrange the input into bit pairs.

2. Generate conditional sums.

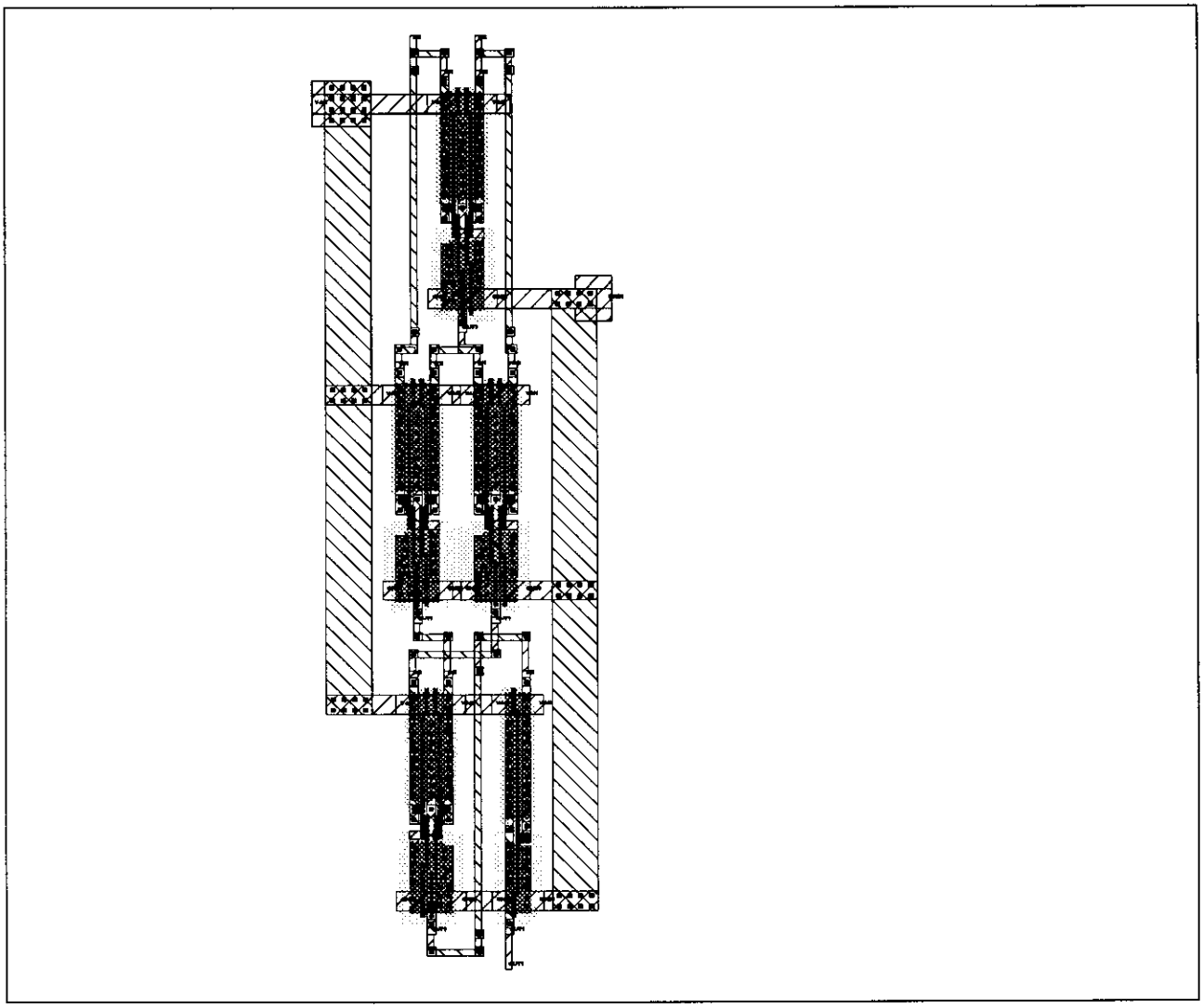


Figure 6.19: XNOR gate CIF Layout

3. Iterate over the conditional sums to get the true sums.

```
defun csa
      combine ° genSums ° setup
enddef
```

The function *setup* takes an input of the form

$$\langle\langle x_{n-1},\ldots,x_0\rangle,\langle y_{n-1},\ldots,y_0\rangle,c_{-1}\rangle$$

and re-arranges it to form the output

$$\langle\langle\langle x_{n-1},y_{n-1}\rangle\ldots\langle x_1,y_1\rangle\rangle\langle x_0,y_0,c_{-1}\rangle\rangle$$

```
defun setup ( a b c )
      [ tlr ° 1, apndr ° [ last ° 1, 2]] ° [ trans ° [ a , b ], c]
enddef
```

The function *genSums* generates the conditional sums and carries for each of the bit pairs. A true sum and carry is generated for the the least significant bit pair. The format of its output is:

$$\langle\langle\langle c_n^1,s_{n-1}^1\rangle\langle c_n^0,s_{n-1}^0\rangle\rangle,\ldots,\langle\langle c_1,s_0\rangle\rangle\rangle$$

c^i (s^i) is a conditional carry (sum) assuming the carry-in is i.

```
defun genSums
      apndr ° { & provSumGen , [ FA ] }
enddef
```

FA is a full adder, and *provSumGen* is a function that generates the conditional sums and conditional carries for a bit pair. The input is

$$\langle x_i,y_i\rangle$$

and the output is

$$\langle\langle c_{i+1}^1,s_i^1\rangle,\langle c_{i+1}^0,s_i^0\rangle\rangle$$

```
defun provSumGen
      [ [ org, xnorg ] , HA ]
```

xnorg is an XNOR gate and *HA* is a half adder.

The function *combine* takes a list of conditional sums/carries and iteratively combines them into a true sum.

```
defun combine
      if = ° [ length, %1 ]
      then id
      else combine ° group fi
enddef
```

The function *group* pairs off a collection of conditional sums from the least-significant bit and combines them together. The input is of the form

$$\langle\langle\langle c_n^1, s_{n-1}^1\rangle, \langle c_n^0, s_{n-1}^0\rangle\rangle, \ldots, \langle\langle c_1, s_0\rangle\rangle\rangle$$

and the output is

$$\langle\langle\langle c_n^1, s_{n-1}^1, s_{n-2}^1\rangle, \langle c_n^0, s_{n-1}^0, s_{n-2}^0\rangle\rangle, \ldots, \langle\langle c_1, s_0\rangle\rangle\rangle$$

The details of *group*'s implementation are not shown here. The interested reader is referred to [100].

The topological description of a conditional-sum adder is shown in figure 6.20 and the corresponding CIF is shown in figure 6.21. This adder could not be generated without hierarchy because doing so resulted in more cells than could be handled by the layout tools. Hence the adder was hierarchically composed out of 2-to-1 multiplexors, a full adder and the cell called *CHA* that corresponds to the function *provSumGen*.

The topological layout of the *CHA* cell is shown in figure 6.22, and the CIF generated is shown in figure 6.23.

The exercise of generating the layout for the conditional-sum adder brought out a problem with the interface to the layout system. In $\nu\mathcal{FP}$, functions are generic and can describe cells of different bit-lengths. In this particular example, a generic multiplexor was used. To get the layout in figure 6.20 the $\nu\mathcal{FP}$ function corresponding to the multiplexor was tagged. However, in the resultant topology there are two 2-bit multiplexors and one 3-bit multiplexor. $\nu\mathcal{FP}$ considers both of them to be of the same type but the layout system considers them to be of two different types. This mismatch was resolved by having the $\nu\mathcal{FP}$ system check the number of inputs to each instance of a generic multiplexor and have it generate a request for the appropriately-sized layout cell. However, having the system perform the check implies that it has to have knowledge of the cells which are generic. If the layout system provides the capability of describing parameterizable cells, an alternative would be to have the $\nu\mathcal{FP}$ system generate instances of the parameterized cell with the appropriate size parameter.

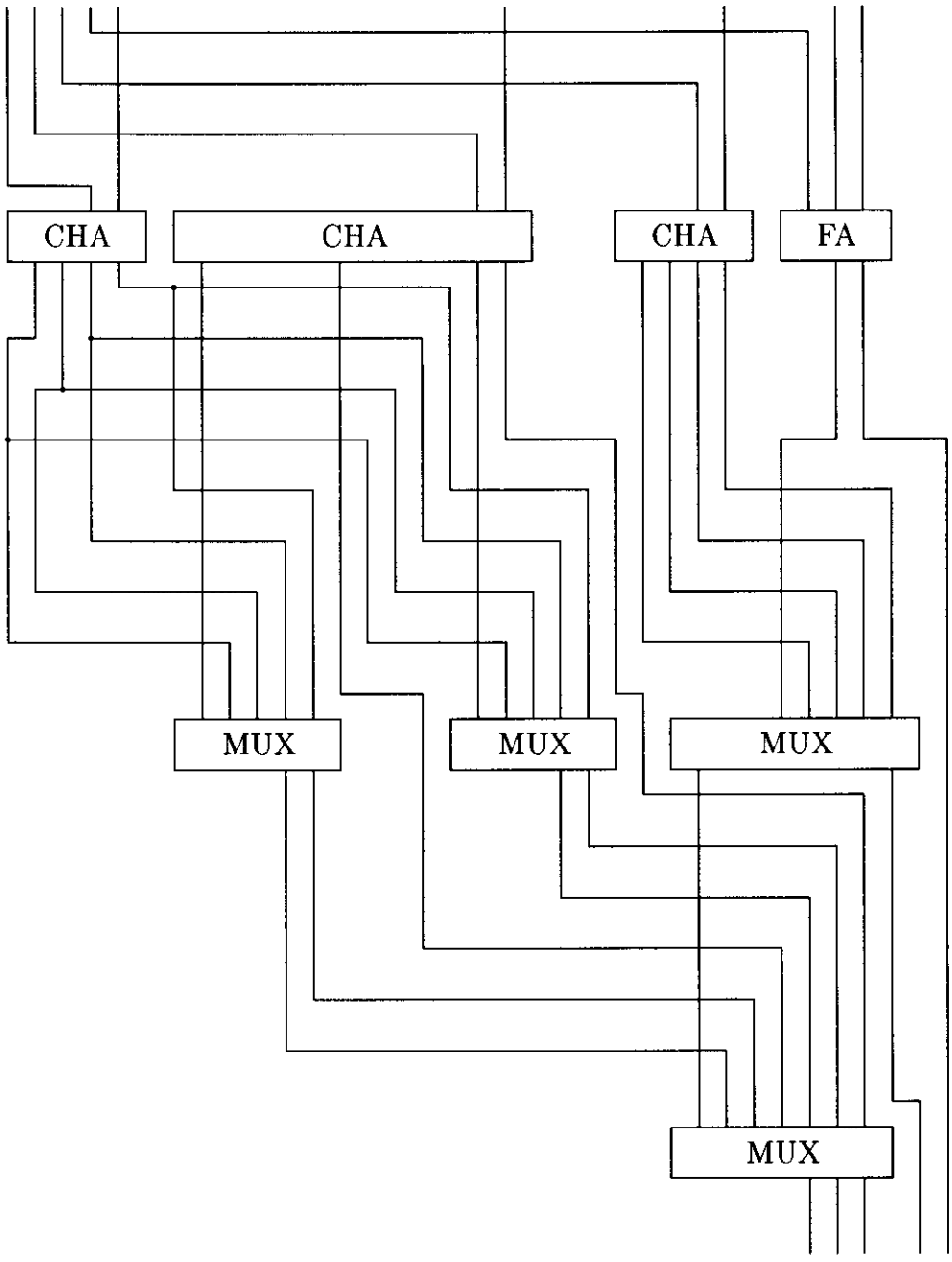


Figure 6.20: Topology of 4-bit Conditional-Sum Adder with carry-out

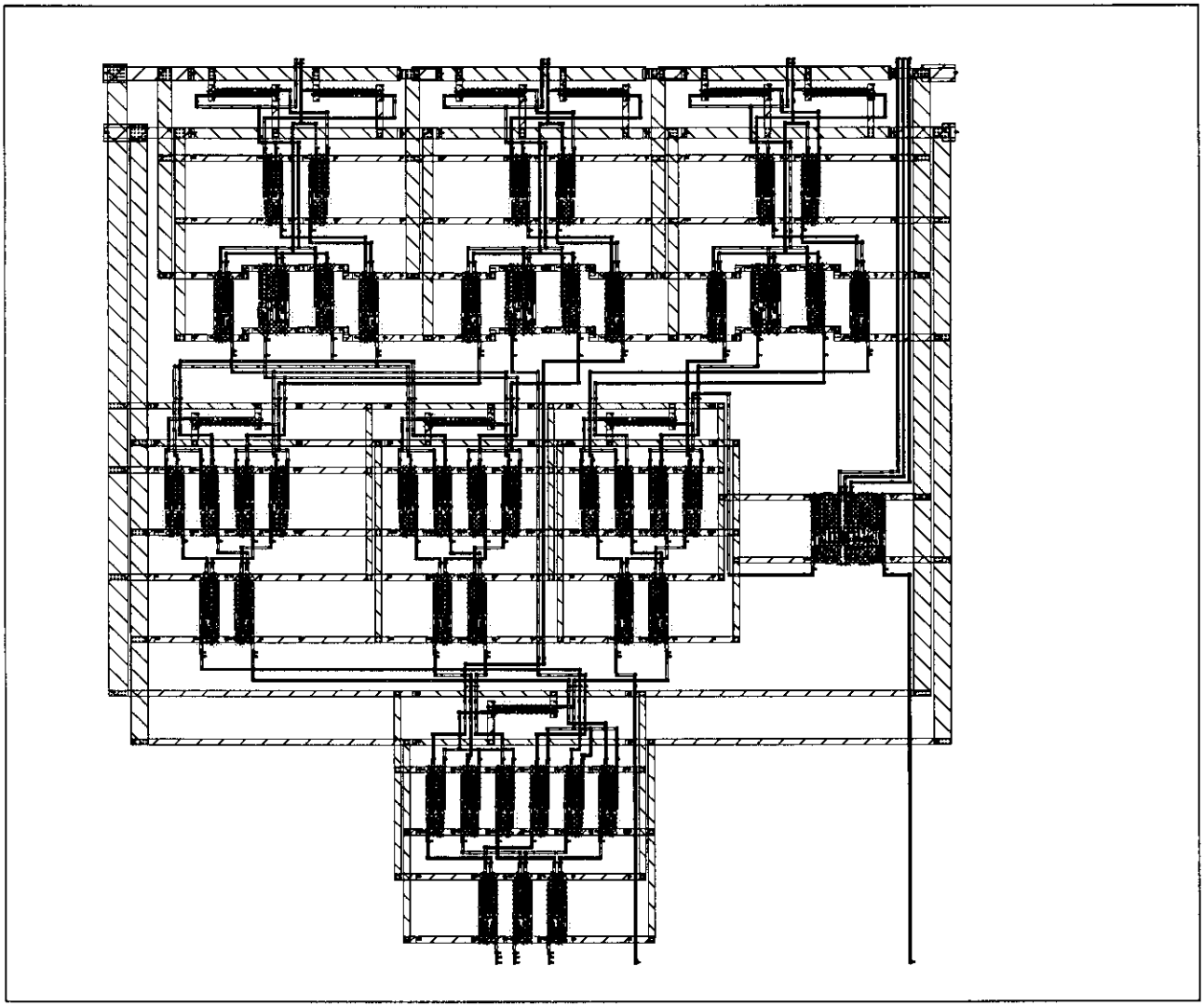


Figure 6.21: 4-bit Conditional-Sum Adder Layout

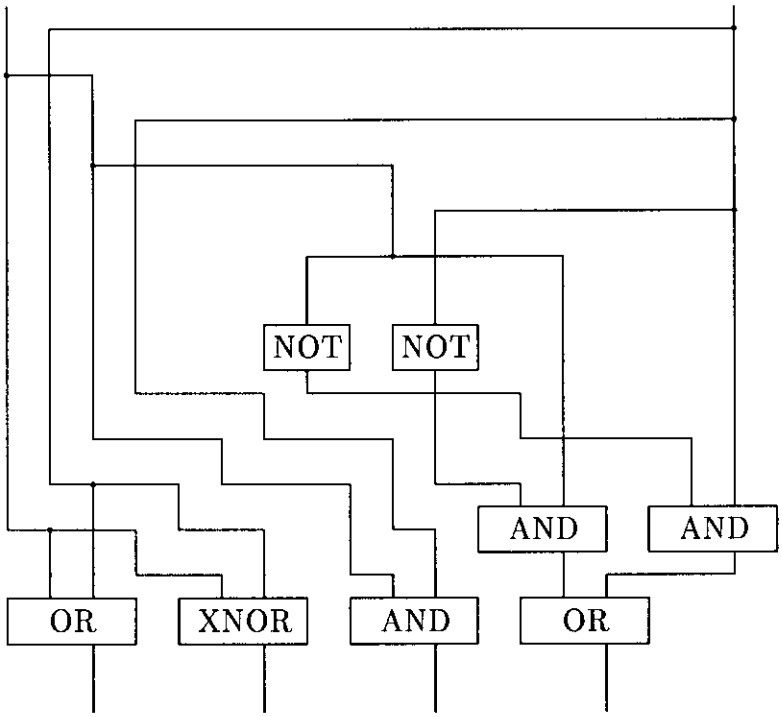


Figure 6.22: Topological Layout of *provSumGen*

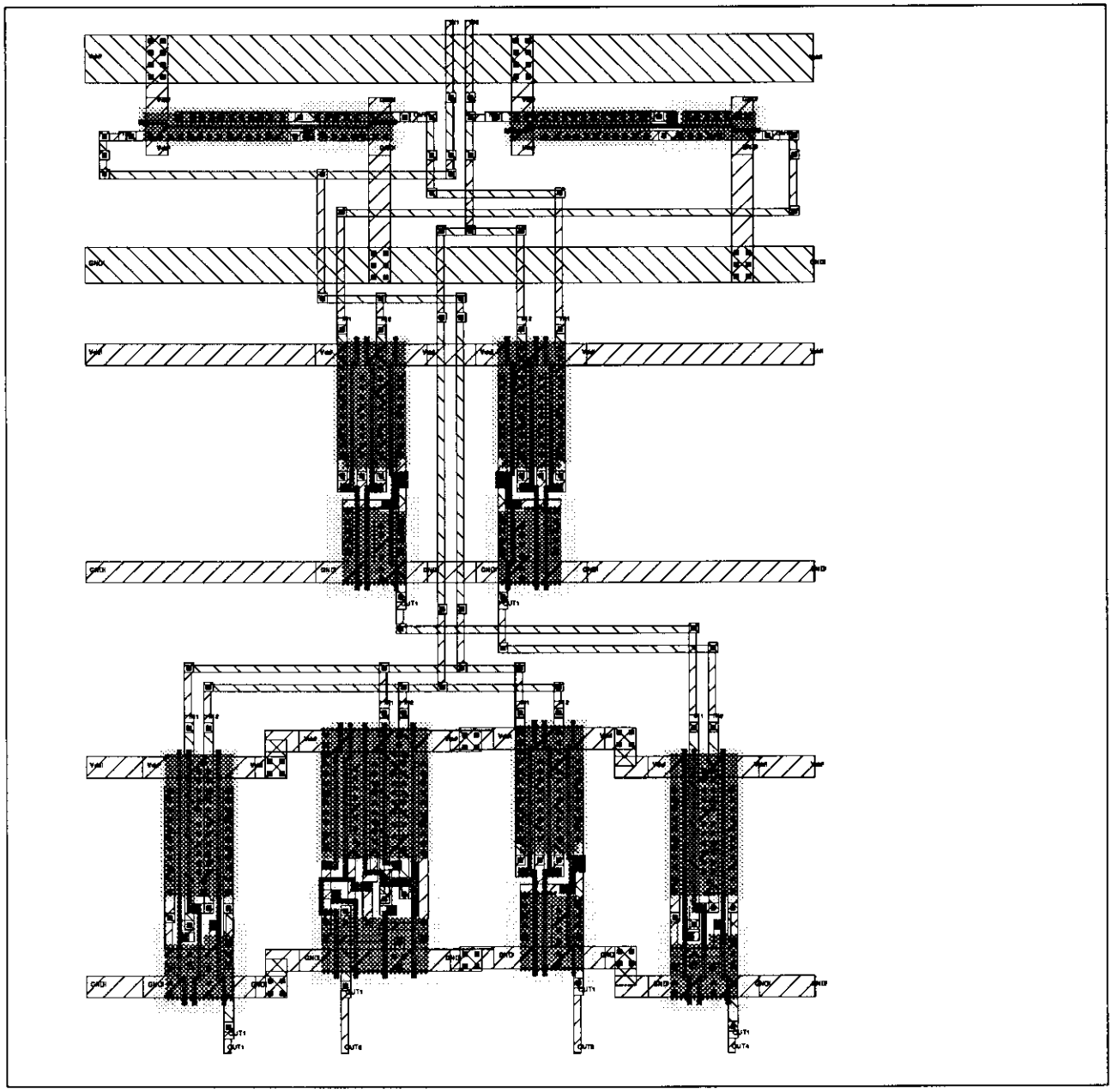


Figure 6.23: Conditional Sum Generator Layout

6.9 Summary

There are two methods for high-level synthesis of structure from behavioral descriptions. One method is to develop a computation graph from the behavioral specification and to map the computation nodes to physical hardware (the process of *allocation*) and to states of a control automaton (the process of *scheduling*). The other is to use semantics-preserving substitutions to convert a specification at the high level down to one only consisting of primitives whose implementation is known. $\nu\mathcal{FP}$ takes the latter approach.

In the $\nu\mathcal{FP}$ synthesis method, substitutions are used to transform a high-level behavioral description down to an equivalent description that only uses the primitives that are available for implementation. Currently this process is guided manually. It is the designer that decides which transformations to apply and the order in which to apply them. After applying the transformations there should be no constructs (like `iota`) whose output structure depends on its input value. In addition, all conditional predicates must be evaluable with only the knowledge of the structure of the input. Under these conditions, the $\nu\mathcal{FP}$ expression is symbolically evaluated and the resultant computation flow graph (actually a tree) is laid out. This is called *space synthesis.*

If this fully expanded circuit takes up too much space on the chip, the designer can choose parts of the circuit to be transformed from a space-domain layout to a time-domain layout. These transformations make use of space-time duality to guarantee that applying these transformations will not change the semantics of the description. This is called *time synthesis.* After all the required sub-circuits have been converted to their time-domain implementations, the delays in the circuit have to be balanced, by moving the $\mathcal{D}^{-1}$ elements to either the inputs or outputs of the circuit. Other optimizing transformations may also be applied at this step. At this point the only $\mathcal{D}^{-1}$ elements will be at the input or output of the circuit where they denote the latency of the circuit. The circuit description is now ready to be shipped out to a convenient back-end system for the generation of the layout, routing of input/output pads, and power routing. Currently, the interface is to the LagerIV system. Though a layout can be produced automatically, it may be rather inefficient. LagerIV allows human intervention to improve the quality of the layout.

Chapter 7

Comparison with μFP

μFP [18] and $\nu\mathcal{FP}$ [19] are two, independently developed systems that use variants of Backus' FP [60] to specify algorithms and then synthesize circuits directly from those descriptions. Both these systems are similar in that they make use of the correspondence between programs written in FP and the connectivity of the primitives that make up the program. For the most part, they are equivalently powerful in that they are able to describe the same class of circuits. However, they differ in their handling of sequential circuits and generation of layout. To gain a better understanding of the strengths and weaknesses of each, this chapter attempts to define precisely the fundamental differences between the two systems.

7.1 Handling Sequential Circuits

μFP extends the semantics of FP to stream semantics. That is, instead of each μFP function operating on a single argument as in FP, μFP extends the semantics of the functions to operate on a infinite sequence (stream) of input arguments.

$$f \text{ (in } \mu\text{FP)} \equiv \&f \text{ (in FP)}$$

$\nu\mathcal{FP}$ considers an algorithm independently of its implementation. Most valid $\nu\mathcal{FP}$ programs, however, have a straightforward implementation in space. That is, all sequences in the algorithm are laid out spatially so that each element of the sequence occupies a different spatial location in the implementation. Under conditions mentioned in section 6.3.1, sequences laid out spatially, can be transformed, using time-space duality, into sequences in time. In this case, successive elements of the sequence occupy the same spatial location, but arrive at that location at different instances of time.

μFP works *only* on infinite streams—finite streams cannot be handled. On the other hand, $\nu\mathcal{FP}$ works on finite streams, but infinite streams can be incorporated by first assuming that the input sequence is finite and then removing the sequentializing and parallelizing constructs (SOPI and POSI) from the beginning and end respectively (see section 6.4.1).

7.2 Handling Initial State

On page 9 in [18] it is assumed that the initial state of the "latch" that holds the state is "don't care".

$$\textbf{if } f : \langle\langle x_1, ?\rangle, \langle x_2, s_1\rangle, \ldots\rangle = \langle\langle o_1, s_1\rangle, \langle o_2, s_2\rangle, \langle o_3, s_3\rangle, \ldots\rangle$$
$$\textbf{then } \mu f : \langle x_1, x_2, x_3, \ldots\rangle = \langle o_1, o_2, o_3, \ldots\rangle$$

This is also corroborated by equation VIII on page 16 of [18]. This would imply that μFP is only capable of describing circuits that are "self-starting"[1].

However, on page 87 Sheeran says

> ... the user must provide the *initial* state ... because the shape of the state of a μ cannot always be deduced from the context ...

This initial state is provided by the user during the evaluation of the μFP program and does not constitute a part of the program. In addition to providing the "shape of the state of a μ", the *value* of the initial state is used to calculate the result of a μFP program (see figure 6.4 on page 91 of [18]). This specification of the initial state by the user allows μFP to describe circuits that are not self-starting. On the other hand, in $\nu\mathcal{FP}$, the initial state of every state register is explicitly specified as one of the elements of the input sequence. In the case of the left insert and left **seq**, it is the first element of the sequence. In the case of the right insert and right **seq**, it is the last element. The cost of incorporating initial state into the description can be seen in the proof in section 7.6.3. The corresponding proof in μFP is significantly shorter.

7.3 Extracting Layout

The μFP system takes a μFP program and then replaces all instances of the *insert* and *apply-to-all* forms by *composes* and *constructs*. To do this, it makes use of the shape and value of the

[1] A *self-starting* circuit is one that can be forced into a predetermined state by using a finite sequence of inputs, independent of the initial state.

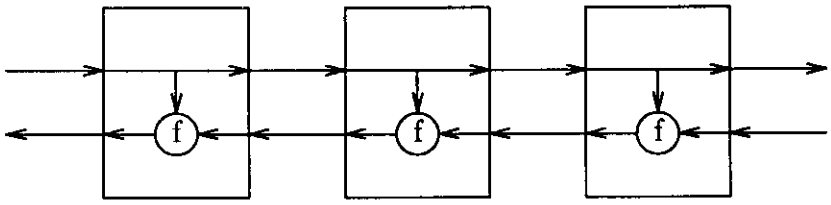


Figure 7.1: Pseudo-bidirectional data flow

input and initial states provided by the user. At the end of this phase, it has transformed the μFP program into a simple data-flow graph of the computation with storage at the μ nodes. This graph is then laid out using the functional geometry [36] system.

On the other hand, $\nu\mathcal{FP}$ uses symbolic interpretation of the program to extract topology of the circuit (see section 6.3 and [55]). This is then pruned and passed through a compactor to get the layout.

7.4 Conditionals

In $\nu\mathcal{FP}$, there are two types of conditionals. The first is *if-then-else-fi*. This is used to implement *structural recursion*. For a circuit containing this type of conditional to be laid out, the condition must be evaluable using only the structure of the argument. This form does not result in any actual layout, but is used to control what is laid out. The other type of conditional is the `sw` and corresponds to a multiplexor in terms of layout.

μFP always expands conditionals into what $\nu\mathcal{FP}$ calls `sw`'s (see page 88, and to a lesser extent equation V on page 16 of [18]). This restricts the ability of the system to use structural recursion to describe complex circuits. A later extension [106] to μFP removes this restriction.

7.5 Bidirectional Flow

On page 44 of [18] Sheeran describes a horizontal composition form called "$\leftrightarrow$" that allows bidirectional communication between modules. In the general case, as shown in equation XIII on page 44 of [18], this could lead to a "deadlock on instability in calculating the output". However, in all given examples, even though the module may look as if it is performing bidirectional communication, the data flow is unidirectional as shown in figure 7.1 below. A later extension [106] to μFP removes this restriction.

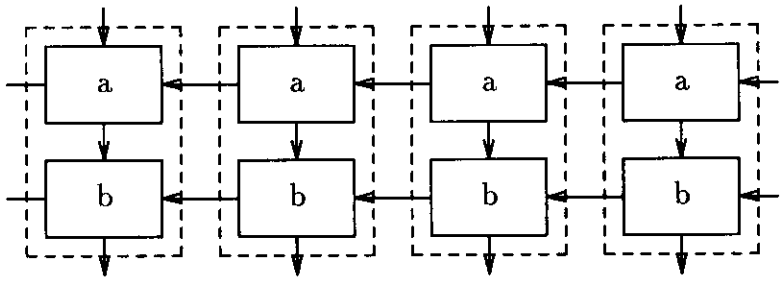


Figure 7.2: Composition of two sequential machines

7.6 Combining Sequential Machines

In μFP there is an identity that relates the combination of two sequential machines. The machines are connected in such a way that the output of the first is connected to the input of the second. In μFP , the identity can be stated as:

$$\mu[f,g] \circ \mu[h,j] = \mu[f \circ [h \circ [1, 2 \circ 2], 1 \circ 2],\ [g \circ [h \circ [1, 2 \circ 2], 1 \circ 2], j \circ [1, 2 \circ 2]]] \quad (7.1)$$

This section develops the corresponding identity for $\nu\mathcal{FP}$. The formal proof of this identity (section 7.6.3) is an excellent example of how transformations can be used to prove that two descriptions in $\nu\mathcal{FP}$ have the same input-output behavior.

7.6.1 The Combinational Theorem

First consider the space implementation of a **seq**. Figure 7.2 shows how two sequential machines can be combined with the output of the first connected to the input of the second.

As can be seen in figure 7.2, the two functions **seq** a and **seq** b are laid out so that the output of **seq** a is fed to the inputs of **seq** b. However, the dotted lines show that we can consider the combined function to be a **seq** of the function inside each dotted box.

Considering the same layout in these two different ways gives us the identity we will be proving.

Theorem 7.1

```
seq([[1 ∘ P, 1 ∘ Q], 2 ∘ Q]) ≡ apndl ∘ [[1 ∘ X, 1 ∘ Y], tl ∘ Y]
                          where X ≡ seqa ∘ apndr ∘ [tlr, 1 ∘ last]
                                Y ≡ seqb ∘ apndr ∘ [tl ∘ X, 2 ∘ last]
                                P ≡ a ∘ [1, 1 ∘ 2]
                                Q ≡ b ∘ [2 ∘ P, 2 ∘ 2]
```

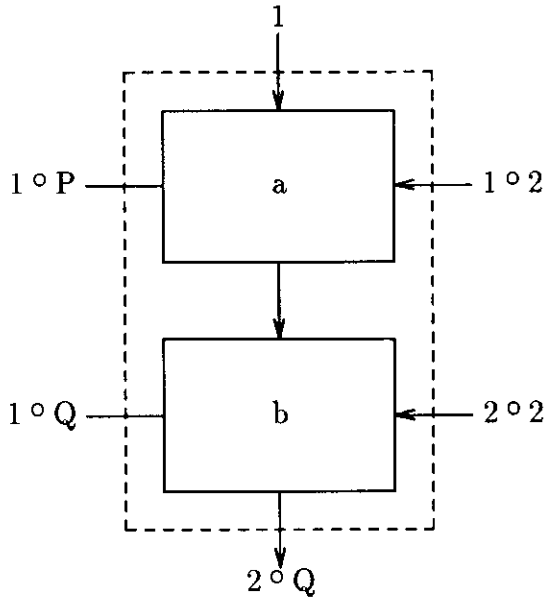


Figure 7.3: A single stage of the combined machine

7.6.2 Informal Proof

First consider each stage of the combined **seq** as shown in figure 7.3. It consists of a function of two inputs, the second of which is a pair. The output is a pair, the first element of which is itself a pair.

Consider the output of the function a. a takes the first element and the first element of the second element as its inputs and gives a pair as output. This is represented by the intermediate function P.

Now consider the output provided by the function b. Its inputs come from the second element of P's output and from the second element of the second element of the original input. This is represented by the intermediate function Q.

Now we can see that the first element of the output of the whole function is a pair, the first element of which is the first element of P's output and the second element is the first element of Q's output. The second element of the whole function is simply the second element of Q's output.

This is the left hand side of theorem 7.1.

Now for the right hand side. Consider X to be the function computed by the first **seq** only. Its input consists of just its initial state, which is the first element of the last element, appended on the right of the rest of the input argument (see figure 7.4).

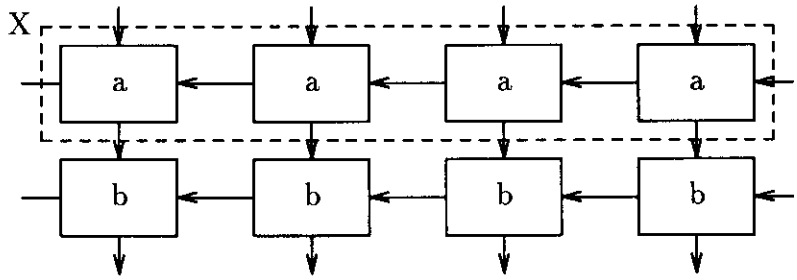


Figure 7.4: Definition of X

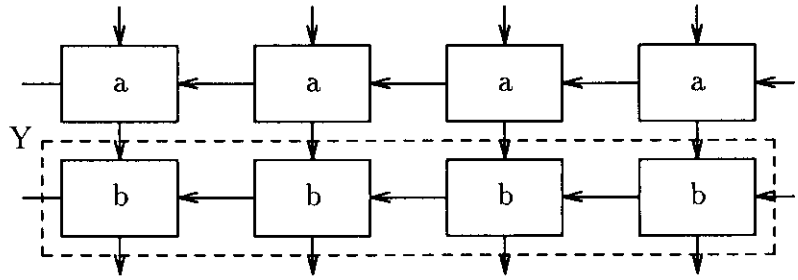


Figure 7.5: Definition of Y

Similarly, let the function computed by the second **seq** be Y. Y's input is the second element of the last element of the input appended to the right of the tail of X's output (see figure 7.5).

Now, we can see that the total function is formed by prepending the tail of Y's output with the pair consisting of the first element of X's output and the first element of Y's output.

This is the right hand side of theorem 7.1.

7.6.3 Formal Proof

The formal proof uses the recursive definition of the **seq** functional form.

$$\mathbf{seq}\, f = \begin{cases} \mathtt{concat} \circ [f \circ [1, 1 \circ 2], \mathtt{tl} \circ 2] \circ [1, \mathbf{seq}\ f \circ \mathtt{tl}] & \text{if } \mathit{arglength} > 2 \\ f & \text{if } \mathit{arglength} = 2 \\ \bot & \text{otherwise} \end{cases} \tag{7.2}$$

For proving the base case, assume that the input argument consists of just two elements. In that case,

$$\mathbf{seq}\, f = f \tag{7.3}$$

We will also use the following identities

$$\texttt{tlr} = [1] \tag{7.4}$$
$$\texttt{tl} = [2] \tag{7.5}$$
$$\texttt{last} = 2 \tag{7.6}$$
$$\texttt{apndr} \circ [[f], g] = [f, g] \tag{7.7}$$

which are true for the case of an input with two arguments. The following identities can be easily proven to be true for suitable arguments. For example, (7.11) and (7.12) assume that f returns a two-element sequence.

$$1 \circ \texttt{concat} \circ [f, g] = 1 \circ f \tag{7.8}$$
$$1 \circ \texttt{apndr} \circ [\texttt{tlr}, 1 \circ \texttt{last}] = 1 \tag{7.9}$$
$$1 \circ 2 \circ [1, f] = 1 \circ f \tag{7.10}$$
$$\texttt{tl} \circ \texttt{concat} \circ [f, g] = \texttt{apndl} \circ [2 \circ f, g] \tag{7.11}$$
$$1 \circ \texttt{apndr} \circ [\texttt{tl} \circ f, g] = 2 \circ f \tag{7.12}$$

To prove the base case of an input with two elements, we first show that

$$X = P \text{ and } Y = Q$$

Lemma 7.1

$$x = P$$

Proof:

$$\begin{aligned} X &= \textbf{seq}\,\mathrm{a} \circ \texttt{apndr} \circ [\texttt{tlr}, 1 \circ \texttt{last}] && \textit{defn of } X \\ &= \textbf{seq}\,\mathrm{a} \circ \texttt{apndr} \circ [[1], 1 \circ 2] && \textit{by (7.4), (7.6)} \\ &= \textbf{seq}\,\mathrm{a} \circ [1, 1 \circ 2] && \textit{by (7.7)} \\ &= \mathrm{a} \circ [1, 1 \circ 2] && \textit{by (7.3)} \\ &= P && \textit{defn of } P \end{aligned}$$

Lemma 7.2

$$Y = Q$$

Proof:

$$\begin{aligned} Y &= \textbf{seq}\,\mathrm{b} \circ \texttt{apndr} \circ [\texttt{tl} \circ X, 2 \circ \texttt{last}] && \textit{defn of } Y \\ &= \textbf{seq}\,\mathrm{b} \circ \texttt{apndr} \circ [[2 \circ X], 1 \circ 2] && \textit{by (7.5), (7.6)} \end{aligned}$$

$= \mathbf{seq}\,\mathrm{b} \circ [2 \circ X, 1 \circ 2]$ *by (7.7)*
$= \mathrm{b} \circ [2 \circ X, 1 \circ 2]$ *by (7.3)*
$= Q$ *defn of Q*

We can now use the previous two lemmas to prove theorem 7.1 for the base case.

Proof:

$\mathtt{apndl} \circ [[1 \circ X, 1 \circ Y], \mathtt{tl} \circ Y]$
$= \mathtt{apndl} \circ [[1 \circ P, 1 \circ Q], \mathtt{tl} \circ Q]$ *by lemmas 7.1, 7.2*
$= \mathtt{apndl} \circ [[1 \circ P, 1 \circ Q], [2 \circ Q]]$ *by (7.5)*
$= [[1 \circ P, 1 \circ Q], 2 \circ Q]$ *defn of* apndl
$= \mathbf{seq}([[1 \circ P, 1 \circ Q], 2 \circ Q])$ *by (7.2)*

Now we are ready to handle the inductive case. First, however, we need to prove some lemmas.

Lemma 7.3

$$\mathtt{tl} \circ \mathtt{apndr} \circ [\mathtt{tlr}, 1 \circ \mathtt{last}] = \mathtt{apndr} \circ [\mathtt{tlr}, 1 \circ \mathtt{last}] \circ \mathtt{tl}$$

This is easy to prove for the case where the input is more than two elements and is left as an exercise.

Lemma 7.4

$$\mathtt{tl} \circ \mathtt{apndr} \circ [\mathtt{tl} \circ X, 2 \circ \mathtt{last}] = \mathtt{apndr} \circ [\mathtt{tl} \circ X, 2 \circ \mathtt{last}] \circ \mathtt{tl}$$

This, too, is an easy exercise to prove for the case when the input has more than two elements.

Lemma 7.5

$$1 \circ a \circ [1, 1 \circ X \circ \mathtt{tl}] = 1 \circ X$$

Proof:

$1 \circ X = 1 \circ \mathbf{seq}\,\mathrm{a} \circ \mathtt{apndr} \circ [\mathtt{tlr}, 1 \circ \mathtt{last}]$ *defn of X*
$= 1 \circ \mathtt{concat} \circ [\mathrm{a} \circ [1, 1 \circ 2], \mathtt{tl} \circ 2] \circ [1, \mathbf{seq}\,\mathrm{a} \circ \mathtt{tl}]$
$\quad \circ \mathtt{apndr} \circ [\mathtt{tlr}, 1 \circ \mathtt{last}]$ *by (7.2)*
$= 1 \circ \mathrm{a} \circ [1, 1 \circ 2] \circ [1, \mathbf{seq}\,\mathrm{a} \circ \mathtt{tl}]$
$\quad \circ \mathtt{apndr} \circ [\mathtt{tlr}, 1 \circ \mathtt{last}]$ *by (7.8)*
$= 1 \circ \mathrm{a} \circ [1, 1 \circ 2] \circ [1, \mathbf{seq}\,\mathrm{a} \circ \mathtt{tl} \circ \mathtt{apndr} \circ [\mathtt{tlr}, 1 \circ \mathtt{last}]]$ *by (7.9)*
$= 1 \circ \mathrm{a} \circ [1, 1 \circ 2] \circ$

$$[1, \mathtt{seq}\,\mathrm{a} \circ \mathtt{apndr} \circ [\mathtt{tlr}, 1 \circ \mathtt{last}] \circ \mathtt{tl}] \qquad \textit{by lemma 7.3}$$
$$= 1 \circ \mathrm{a} \circ [1, 1 \circ \mathtt{seq}\,\mathrm{a} \circ \mathtt{apndr} \circ [\mathtt{tlr}, 1 \circ \mathtt{last}] \circ \mathtt{tl}] \qquad \textit{by (7.10)}$$
$$= 1 \circ \mathrm{a} \circ [1, 1 \circ X \circ \mathtt{tl}] \qquad \textit{defn of } X$$

Lemma 7.6

$$1 \circ Y = 1 \circ b \circ [2 \circ a \circ [1, 1 \circ X \circ \mathtt{tl}], 1 \circ Y \circ \mathtt{tl}]$$

Proof:

$$1 \circ Y = 1 \circ \mathtt{seq}\,\mathrm{b} \circ \mathtt{apndr} \circ [\mathtt{tl} \circ X, 2 \circ \mathtt{last}] \qquad \textit{defn of } Y$$
$$= 1 \circ \mathtt{concat} \circ [\mathrm{b} \circ [1, 1 \circ 2], \mathtt{tl} \circ 2] \circ [1, \mathtt{seq}\,\mathrm{b} \circ \mathtt{tl}] \circ \mathtt{apndr} \circ [\mathtt{tl} \circ X, 2 \circ \mathtt{last}] \qquad \textit{by (7.2)}$$
$$= 1 \circ \mathrm{b} \circ [1, 1 \circ 2] \circ [1, \mathtt{seq}\,\mathrm{b} \circ \mathtt{tl}] \circ \mathtt{apndr} \circ [\mathtt{tl} \circ X, 2 \circ \mathtt{last}] \qquad \textit{by (7.8)}$$
$$= 1 \circ \mathrm{b} \circ [1, 1 \circ 2] \circ [2 \circ X, \mathtt{seq}\,\mathrm{b} \circ \mathtt{tl} \circ \mathtt{apndr} \circ [\mathtt{tl} \circ X, 2 \circ \mathtt{last}]] \qquad \textit{by (7.12)}$$
$$= 1 \circ \mathrm{b} \circ [2 \circ X, 1 \circ \mathtt{seq}\,\mathrm{b} \circ \mathtt{apndr} \circ [\mathtt{tl} \circ X, 2 \circ \mathtt{last}] \circ \mathtt{tl}] \qquad \textit{by lemma 7.4}$$
$$= 1 \circ \mathrm{b} \circ [2 \circ \mathrm{a} \circ [1, 1 \circ X \circ \mathtt{tl}], 1 \circ Y \circ \mathtt{tl}] \qquad \textit{defn of } Y\textit{, lemma 7.5}$$

Lemma 7.7

$$[1 \circ X, 1 \circ Y] = [1 \circ P, 1 \circ Q] \circ [1, [1 \circ X, 1 \circ Y] \circ \mathtt{tl}]$$

Proof:

$$[1 \circ P, 1 \circ Q] \circ [1, [1 \circ X, 1 \circ Y] \circ \mathtt{tl}]$$
$$= [1 \circ \mathrm{a} \circ [1, 1 \circ 2], 1 \circ \mathrm{b} \circ [2 \circ \mathrm{a} \circ [1, 1 \circ 2], 2 \circ 2]] \circ [1, [1 \circ X, 1 \circ Y] \circ \mathtt{tl}] \qquad \textit{defn of } P, Q$$
$$= [1 \circ \mathrm{a} \circ [1, 1 \circ X \circ \mathtt{tl}], 1 \circ \mathrm{b} \circ [2 \circ \mathrm{a} \circ [1, 1 \circ X \circ \mathtt{tl}], 1 \circ Y \circ \mathtt{tl}]] \qquad \textit{by expansion}$$
$$= [1 \circ X, 1 \circ Y] \qquad \textit{by lemmas 7.5, 7.6}$$

Lemma 7.8

$$\mathtt{apndl} \circ [2 \circ Q \circ [1, [1 \circ X, 1 \circ Y] \circ \mathtt{tl}], \mathtt{tl} \circ Y \circ \mathtt{tl}] = \mathtt{tl} \circ Y$$

Proof:

$$\mathtt{tl} \circ Y = \mathtt{tl} \circ \mathtt{seq}\,\mathrm{b} \circ \mathtt{apndr} \circ [\mathtt{tl} \circ X, 2 \circ \mathtt{last}] \qquad \textit{defn of } Y$$
$$= \mathtt{tl} \circ \mathtt{concat} \circ [\mathrm{b} \circ [1, 1 \circ 2], \mathtt{tl} \circ 2] \circ [1, \mathtt{seq}\,\mathrm{b} \circ \mathtt{tl}]$$

$$
\begin{aligned}
& \circ \texttt{apndr} \circ [\texttt{tl} \circ X,\ 2 \circ \texttt{last}] && \textit{by (7.2)} \\
= \ & \texttt{apndl} \circ [2 \circ \texttt{b} \circ [1,1 \circ 2],\ \texttt{tl} \circ 2] \ \circ [1,\ \texttt{seqb} \ \circ \texttt{tl}] \\
& \circ \texttt{apndr} \circ [\texttt{tl} \circ X,\ 2 \circ \texttt{last}] && \textit{by (7.11)} \\
= \ & \texttt{apndl} \circ [2 \circ \texttt{b} \circ [1,1 \circ 2],\ \texttt{tl} \circ 2] \\
& \circ [2 \circ X,\ \texttt{seqb} \ \circ \texttt{tl} \circ \texttt{apndr} \circ [\texttt{tl} \circ X,\ 2 \circ \texttt{last}]] && \textit{by (7.12)} \\
= \ & \texttt{apndl} \circ [2 \circ \texttt{b} \circ [1,1 \circ 2],\ \texttt{tl} \circ 2] \\
& \circ [2 \circ X,\ \texttt{seqb} \ \circ \texttt{apndr} \circ [\texttt{tl} \circ X,\ 2 \circ \texttt{last}] \circ \texttt{tl}] && \textit{by lemma 7.4} \\
= \ & \texttt{apndl} \circ [2 \circ \texttt{b} \circ [1,1 \circ 2],\ \texttt{tl} \circ 2] \circ [2 \circ X,\ Y \circ \texttt{tl}] && \textit{defn of } Y \\
= \ & \texttt{apndl} \circ [2 \circ \texttt{b} \circ [2 \circ X,\ 1 \circ Y \circ \texttt{tl}],\ \texttt{tl} \circ Y \circ \texttt{tl}] && \textit{by expansion}
\end{aligned}
$$

Now we gather all the lemmas together to show the final result.

Proof:

$$
\begin{aligned}
\text{LHS} = \ & \texttt{concat} \circ [[[1 \circ P,\ 1 \circ Q],\ 2 \circ Q] \ \circ [1,\ 1 \circ 2],\ \texttt{tl} \circ 2] \\
& \circ [1,\ \texttt{apndl} \circ [[1 \circ X,\ 1 \circ Y],\ \texttt{tl} \circ Y] \ \circ \texttt{tl}] && \textit{by theorem 7.1, (7.2)} \\
= \ & \texttt{concat} \circ [[[1 \circ P,\ 1 \circ Q],\ 2 \circ Q] \ \circ [1,\ [1 \circ X,\ 1 \circ Y] \ \circ \texttt{tl}], \\
& \texttt{tl} \circ Y \circ \texttt{tl}] && \textit{by expansion} \\
= \ & \texttt{apndl} \circ [[\ 1 \circ P,\ 1 \circ Q] \ \circ [1,\ [1 \circ X,\ 1 \circ Y] \circ \texttt{tl}], \\
& \texttt{apndl} \circ [2 \circ Q \ \circ [1,\ [1 \circ X,\ 1 \circ Y] \circ \texttt{tl}],\ \texttt{tl} \circ Y \circ \texttt{tl}]] && \textit{by expansion} \\
= \ & \texttt{apndl} \circ [[1 \circ X,\ 1 \circ Y],\ \texttt{tl} \circ Y] && \textit{by lemmas 7.7, 7.8} \\
= \ & \text{RHS}
\end{aligned}
$$

7.7 Summary

This chapter points out the differences between μFP and $\nu\mathcal{FP}$—two similar systems for synthesizing VLSI circuits from applicative expressions. The major difference between them is the way they handle sequential circuits. μFP handles sequential circuits by extending FP semantics to infinite streams. Thus μFP is more suited to describing and synthesizing circuits like systolic arrays that work on infinite streams of data, and has difficulty dealing with finite length sequences. On the other hand, $\nu\mathcal{FP}$ incorporates sequential circuits by using space/time duality on fixed length sequences and is thus better suited to synthesizing algorithms that use sequences of fixed length.

A successor to μFP called RUBY [107, 106] uses relations on infinite streams instead of functions which provides greater clarity and symmetry to the descriptions. The μFP restrictions on structural recursions and bi-directional flow are lifted in RUBY.

Another difference between $\nu\mathcal{FP}$ and μFP is the way in which they handle initial conditions. In μFP the initial conditions are specified separately from the circuit description. In $\nu\mathcal{FP}$ the

initial conditions are part of the input and are used by the description. Both μFP and $\nu\mathcal{FP}$ have a law which shows how to transform a sequential composition of two state machines into one higher-level state machine. The proof of the $\nu\mathcal{FP}$ law is provided in this chapter. It serves as an extended example of the use of transformations in proving properties about the representation. However, the $\nu\mathcal{FP}$ proof is much larger than the corresponding μFP proof. The difference in size can be traced to the fact that $\nu\mathcal{FP}$ incorporates the initial conditions into the description whereas μFP does not. It is not clear whether benefits of dealing explicitly with the initial conditions in the description are worth the added complexity of the proofs.

Chapter 8

Conclusions

The objective of this research was to develop a formal high-level language approach to specification, simulation, performance evaluation, floor planning, and chip layout for VLSI systems. A high-level applicative language ($\nu\mathcal{FP}$) and programming style was the basis of the approach. A prototype system was built that can be used for describing algorithms, providing performance estimates, and generating topological layouts at all levels of abstraction. An interface to a back-end VLSI layout system is provided for the generation of mask layouts.

This dissertation covers a part of the design space (see figure 1.1 for the complete spectrum). The highest behavioral abstraction level covered is that dealing with algorithms and their specification. Algorithmic descriptions are then converted into structural designs at the level of interconnected modules. These modules are further refined until the modules represent realizable primitives at the binary level. Topological floor plans are generated from this network of gates and fed to a back-end system for the physical layout level generation of layout masks.

8.1 Design Framework

The major contribution of this work is in providing a coherent framework, based on an applicative language, for the specification, analysis, design, synthesis and layout of VLSI algorithms. The fact that there is one coherent framework, makes the design of tools operating on the representation easier. Additionally, users only need to learn one language to be able to use the system at all the different levels. The disadvantage is that the same language may not be ideally suited to all the levels. In this particular case, our experience shows that $\nu\mathcal{FP}$ is more useful at a higher level of abstraction and tends to be less so at levels lower than the gate level.

One of the benefits of using $\nu\mathcal{FP}$ is that by appropriately choosing the set of primitive

functions, the user can stop the refinement procedure at any convenient level of abstraction. For instance, this dissertation describes a method whose level of realization is that of custom VLSI. There is no reason why a designer could not stop at the gate or macro-cell level if the intended implementation backend was to be gate-arrays.

An important aspect of the $\nu\mathcal{FP}$ framework is the ability to study the design representation at any arbitrary level of abstraction. Designs can be evaluated with respect to performance and visual feedback provided at all abstraction levels. This feedback, the visual and the performance measures, is extremely beneficial in allowing designers to take appropriate action all throughout the design process.

8.2 The $\nu\mathcal{FP}$ Tools

Various design tools were developed as part of this research. Figure 8.1 shows the overall data-flow in the $\nu\mathcal{FP}$ design system. Programs can be written in either of two forms. The `tFP` interpreter accepts programs written in the $\nu\mathcal{FP}$ syntax described herein, the `bFP` interpreter accepts programs written in the Berkeley FP [79] syntax. The `tFP` interpreter, written in **T** [87], performs syntax checks and converts the program into an intermediate form that is an annotated **T** expression. This intermediate form is a prefix parse tree of the original $\nu\mathcal{FP}$ program. The `conv` program converts a program written in Berkeley FP to the intermediate form. The `quick-layout` program symbolically interprets this intermediate form to generate the topological cross-sections described in section 6.6 and [55]. This description is placed in a file with a `.d` extension. The `.d` file is converted to TEX via `xplot`. The `netlist` program converts the `.d` description to input compatible with the LagerIV system and thence to CIF. `conv`, `quick-layout`, and `xplot` were written by Martine Schlag [55]. The translation to VIVID was done by Winthrop Wu [100].

8.3 Sequential Behavior via Space/Time Duality

One of the major problems in applicative languages is the treatment of state in an otherwise state-less system. A major contribution of this work is to provide an alternative approach to introducing sequential (state-oriented) behavior into an applicative framework (section 6.4). The formal basis for this is discussed in section 6.5. Sequential behavior has been introduced into $\nu\mathcal{FP}$ without giving up the power of transformations or referential transparency which are useful properties of applicative languages. Previous approaches to this problem have either used lazy functional languages [62] or have extended the semantics of a strict functional language to

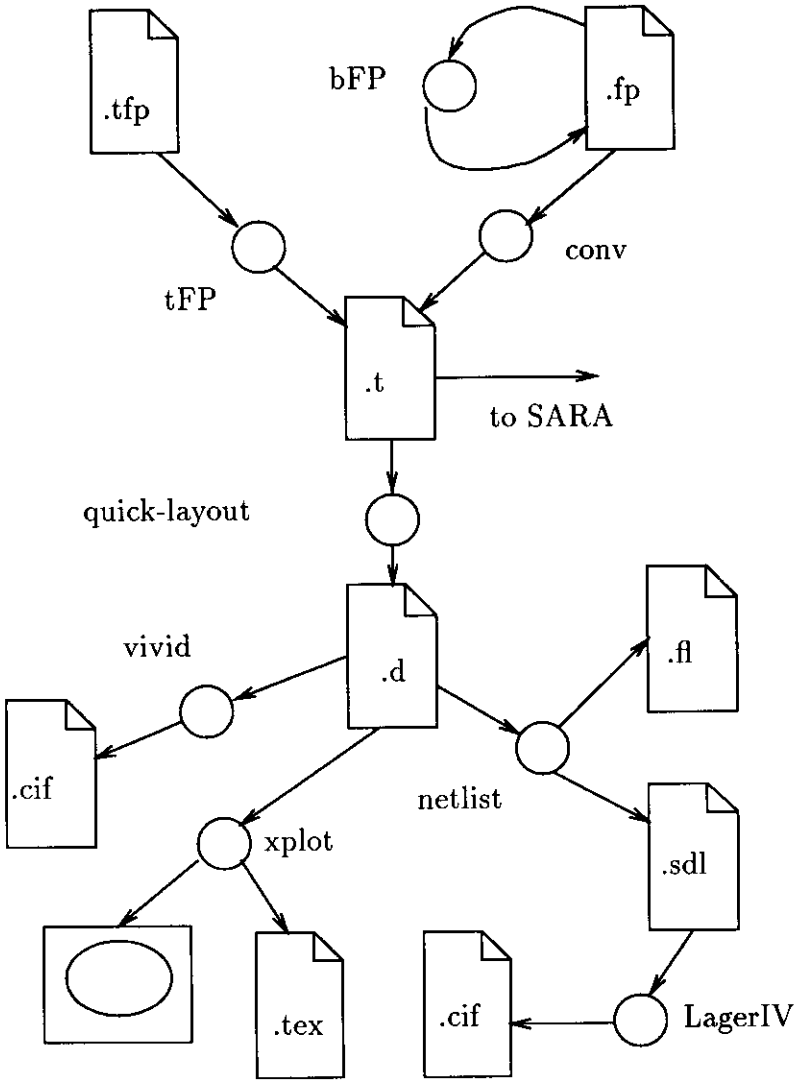


Figure 8.1: The $\nu\mathcal{FP}$ Design System

include streams [18]. This dissertation has taken the approach of applying the duality between space and time to sequences in $\nu\mathcal{FP}$ and using transformations to convert between time and space implementations of functional forms.

There are tradeoffs in this means of introducing sequential behavior (see chapter 7). $\nu\mathcal{FP}$ is better at describing systems that work on fixed length sequences, but has problems describing systems (like systolic arrays) that work on infinite streams. $\nu\mathcal{FP}$ incorporates initialization of state registers into the algorithmic description instead of specifying initialization separately from the behavior. Experience has shown that though this formalizes the initialization process and allows it to be analyzed in the same way as the rest of the behavior, it complicates proofs by a significant amount. More experience is required before a judgment can be made on whether this is a useful tradeoff.

8.4 Synthesis via Transformations

$\nu\mathcal{FP}$ synthesizes circuits via successively applying semantics-preserving substitutions to a high-level description until the description only contains primitive constructs with known implementations. Though the idea of using transformations for synthesis is not new [108, 109, 110, 94, 111], this dissertation extends the idea to the domain of sequential circuits.

8.4.1 Space Synthesis

There are two stages in the $\nu\mathcal{FP}$ synthesis process. The first stage is *space synthesis* which consists of symbolically evaluating the $\nu\mathcal{FP}$ description and generating a computation graph. This graph is then pruned, compacted and laid out using techniques described in section 6.6 and [55]. This synthesis method expands out all iterations, recursions and conditionals spatially. Each instance of a primitive function is represented by an instance of the corresponding primitive cell at some spatial location. Not all computable functions can be synthesized in this way. First, the functions must be representable in $\nu\mathcal{FP}$. This means that, for instance, mutual exclusion and infinite streams cannot be represented. Second, functions like `iota` whose output structure depends on input value are not allowed. Third, the predicate of every conditional must be evaluable with only the knowledge of the structure of its input. The last two restrictions are a direct consequence of using symbolic evaluation to extract structure from a $\nu\mathcal{FP}$ expression. Using other methods for structure extraction may relax those restrictions.

8.4.2 Time Synthesis

The second synthesis stage is performed if the layout obtained from the space synthesis is too large. In that case, parts of the circuit can be transformed, using transformations shown in section 6.4, from their space-domain implementations to their time-domain equivalents. This process may introduce inverse delay elements into the circuit. Since these cannot be implemented, they have to be transformed out to the output of the circuit where they denote the latency of the circuit. Normally, $\nu\mathcal{FP}$ can only represent circuits that use finite sequences. However, in the special case where it is possible to convert a circuit into one that only has a SOPI at the inputs and only a POSI at the outputs, it is possible to remove those elements and be left with a circuit that operates on an infinite stream.

8.4.3 Generating Layouts

Given a planar topology derived from the symbolic evaluation of $\nu\mathcal{FP}$ expressions, section 6.6 provides a method for generating connectivity and placement information that can be supplied to a back-end system for generating layout masks. Layouts of various examples have been generated to demonstrate the validity of the approach. This method is flexible enough to be easily adapted to the different formats required by different back-ends. Two kinds of representative netlists have been generated.

8.5 Transformations for Proving Equivalence

Another contribution of this dissertation is in providing a better understanding of, and tools for, refinement in the design process. In particular, this work has shown how it is possible to aid the process of formally proving that a particular design is a refinement of another. In this context, chapter 5 demonstrates how it is possible to transform a design between the control, data, and interpretation domains within the constraints of a particular design paradigm. Therefore, $\nu\mathcal{FP}$ facilitates the process of refinement and the process of moving between different views of the same design object.

Transformations are used throughout the dissertation to prove equivalence between two expressions. Not all the derivations are shown. Section 7.6.3 contains the proof of a theorem in $\nu\mathcal{FP}$ corresponding to a μFP theorem proved in [18]. This proof also serves as an extended example of the use of transformations in proving equivalences.

8.6 Formalizing Aspects of GMB Semantics

The effort of incorporating $\nu\mathcal{FP}$ into a design system with a radically different design paradigm produced two subsidiary contributions. Since $\nu\mathcal{FP}$ is a formally defined system, its introduction into a less formally defined system lead to the questioning and clarification of the semantics provided by the design system. The results are described in appendix B. Since the design system provided useful features that were substantially different from the $\nu\mathcal{FP}$ paradigm of design it highlighted the limitations of a purely applicative design system.

The concrete contributions are in formalizing the semantics of token removal (section B.3) in the Graph Model of Behavior. Multiple token removal (section B.6) in the presence of the priority operator is also handled. In addition, a formal translation (section B.4) from logic expressions to partially ordered sets is provided. These posets are used as the basis for the token removal semantics. This translation provides two benefits: formalizing the meaning of what a particular logic expression means, and providing a basis for developing an algebra of logic expressions.

8.7 Applicative Specification Languages

The usefulness of applicative languages for algorithm specification is demonstrated in chapter 3. It also shows how $\nu\mathcal{FP}$ can provide abstraction in the specification by using two-dimensional composition that can be used to encapsulate communication between processes.

Chapter 9

Future Work

Though the $\nu\mathcal{FP}$ system provides a research framework for VLSI synthesis, there are some aspects that could benefit from further explorations. In addition, some of the research described here opens up other avenues of fruitful research. Some of these are open research issues whereas others were not completed only because they were not central to this dissertation. The issues are divided into two parts. Those that deal with the synthesis of hardware algorithms and those that deal with the Graph Model of Behavior.

9.1 Issues in Synthesis

This section discusses the issues in the synthesis of hardware algorithms that are raised by this dissertation. The open research topics are presented first.

9.1.1 Transformations

During the process of building this system many transformations were defined and used. However, there are many more useful ones that have already been proposed and many that are surely waiting to be discovered. The first task would be to collect all such transformations together in one place and to generate further transformations.

Currently transformations are classified into three broad groups: structural, tradeoff, and domain-specific. Structural transformations are those that only depend on FP and are usually used to rearrange the structure of a $\nu\mathcal{FP}$ expression. For example,

$$\&f \circ [g,h] \equiv [f \circ g, f \circ h]$$

The most obvious tradeoff transformations are the time/space transforms. For example,

$$\&f \equiv \mathcal{D}^{-1} \circ \text{POSI} \circ \&^T f \circ \text{SOPI}$$

The domain-specific transformations are those that are specific to the current problem domain. It the case of $\nu\mathcal{FP}$, this is boolean logic. An example of this type of transformation would be deMorgan's Law.

$$\texttt{notg} \circ \texttt{andg} \equiv \texttt{org} \circ \&\texttt{notg}$$

These are only three obvious classes. The next step would be to take all the transformations gathered previously and classify them. It is very likely that there could be other useful classes of transforms.

9.1.2 Expert System for Synthesis

The current system provides a framework for synthesis. It does not provide any "smarts" for the actual synthesis process. Currently, the system does not provide a way to automatically synthesize a circuit from an algorithmic description. Though there are transformations available, the selection of the transformations and the order of applying them is left up to the designer.

Given a useful and extensible set of transformations, it should be possible to develop an expert-system that would decide which transformations should be applied to transform a specification to an implementation to meet design constraints. The challenges in such a system would lie in coming up with a scheme that could perform adequate resource allocation and performance prediction to result in layouts that are comparable to those designed by hand. This could be modeled on the way human designers work when they are trained to perform within a transformation-based system.

It would be interesting to speculate whether and how $\nu\mathcal{FP}$ could be used to incorporate synthesis methods developed in other areas. Work has already been done to automate the use of transformations in the synthesis process [108, 109, 110, 94, 111, 112, 73], and it can all be incorporated into the $\nu\mathcal{FP}$ framework. Another approach would be to take the the work done on space-time scheduling, allocation and synthesis [74, 75, 72, 7, 113, 96] and adapt it to the $\nu\mathcal{FP}$ paradigm. Algorithms used for scheduling and allocation of computation graphs without feedback should map into $\nu\mathcal{FP}$ in a straightforward manner. Graphs with feedback may need to be converted to a different form before they could be applied to the $\nu\mathcal{FP}$ framework.

9.1.3 Automated High-Level Test-Case Generation

Conventional test-case generation generates test-cases based on functional blocks at the lowest level of abstraction (usually gates). At this point, all the higher-level information of what these gates do has been lost. In $\nu\mathcal{FP}$, however, all the functional information is available at all the hierarchical levels. It should be possible to use this information to generate better and fewer test-cases to exercise the circuit.

A promising approach would be to use inverse functions to determine what inputs to the circuit will produce the desired inputs to modules inside the circuit. Inverse functions will have to be provided for all primitive functions and some way found to provide inverses for combining forms. Work done in using AHPL for test case generation [114] could be useful in this context. In order to be able to test the circuit module by module, it will have to incorporate some kind of scan-in scan-out registers [115].

9.1.4 Extensions

There are two major limitations of the $\nu\mathcal{FP}$ approach as described in this dissertation. The first is the total absence of side-effects and the second is that it is not possible to describe circuits whose output structure depends on input value. Other applicative languages for VLSI design like ELLA and SILAGE have recently introduced both these imperative features into an otherwise-applicative system [12]. It would be interesting to see whether such features could be introduced into the $\nu\mathcal{FP}$ system without compromising its benefits.

A possible path for incorporating side-effects into $\nu\mathcal{FP}$ would be to encapsulate the side-effecting constructs so that they interact with the rest of the system in a very controlled manner. Allowing circuits (like `while` loops) whose output structure depends on input value might be impossible given the current method of generating topology via symbolic interpretation. This is because the input to this construct will be symbolic and hence the output will have an indeterminate symbolic structure. One possible approach to addressing this problem is to devise a system that uses a combination of symbolic and actual values so that actual values could be used in certain cases and symbolic values otherwise. This hybrid would provide the advantages of both the methods.

The rest of the issues in this section are not fundamental, but are documented here for completeness.

9.1.5 Typing, Scoping, and Extended Definitions

Currently, the names of $\nu\mathcal{FP}$ functions are global in scope. This makes it difficult to use $\nu\mathcal{FP}$ for developing large systems—especially when there is more than one programmer involved. This is because of naming conflicts that may arise. It would be a simple extension to incorporate lexical scoping and/or local definitions into $\nu\mathcal{FP}$ functions. A simple translator could make up unique names for each function to guarantee no name conflicts and then translate this extended $\nu\mathcal{FP}$ to the current $\nu\mathcal{FP}$.

$\nu\mathcal{FP}$, as it stands currently, is type-less. Introducing polymorphic types into $\nu\mathcal{FP}$ would allow earlier and better error checking. It would also allow a much closer compatibility to the AXES [116, 117] specification language and enhance $\nu\mathcal{FP}$'s use as a specification language.

The syntax currently used to specify parameters to functions in $\nu\mathcal{FP}$ was a compromise between ease of implementability and flexibility. It should be possible to implement the extended definition mechanism proposed in [76]. The ability to specify pattern matches in the arguments, in particular, would greatly enhance the readability of programs.

9.1.6 User-specified Attributes

As was mentioned in chapter 4, the $\nu\mathcal{FP}$ system currently only evaluates a fixed set of properties of the algorithm. It would have been useful to have an extensible system so that users could define their own attributes for the base cells and their own algorithms for combining their attributes. An implementation using an attribute grammar pre-processor was unsuccessfully attempted.

Currently, the evaluation system assumes unit delay for each tagged function. Extending this capability to be able to specify arbitrary fixed delays would be useful.

Chapter 4 shows how the evaluation of attributes is performed in an applicative manner within the framework. It is hoped that the ease of incorporating extensions in this framework will encourage users to develop other measures to be evaluated.

9.1.7 General Placement of Inputs/Outputs

Currently the inputs to a circuit are always assumed to enter from the top and the outputs always leave from the bottom. There is no inherent reason why this should be so. It should be possible to describe and use circuits with inputs and outputs on any side.

9.1.8 Different Sequential Structures

The present system has time-domain equivalents of only the three most useful functional forms (*insert*, *apply-to-all*, and *sequential*). There is no reason why this could not be extended to other functional forms or to some primitives.

9.1.9 Different Timing Regimes

The formal timing behavior described in section 6.5 is but one way of implementing sequences in time. This example was chosen as an existence proof that it could be done. However, there is no need to be restricted to it. There may be other timing regimes with different properties that are of greater use in different situations.

9.2 Issues in the GMB

This section addresses issues pertaining to the GMB that were raised by the research completed for this dissertation.

9.2.1 Improving Control Flow Analysis

Currently, control flow analysis of the GMB is carried out using information only from the control domain. Thus, some of the anomalies reported by control flow analysis do not actually exist. A designer has then to go through each reported anomaly and prove that it either is real or not. In many cases, given enough information about the data or interpretation domains, it is possible to automatically determine when a suspected anomaly is real or not.

However, the current interpretation domain does not allow this capability because it is not amenable to analysis. Since $\nu\mathcal{FP}$ has simple formal semantics and is easy to analyze, using $\nu\mathcal{FP}$ as an interpretation domain language would allow the control flow analysis to profit from knowledge about the interpretation domain.

9.2.2 An Algebra for Logic Expressions

The conditions under which a GMB control node will fire are specified via the use of input logic expressions. Output logic expressions specify which control arcs will receive tokens after the control node terminates. Section B.4 shows how input logic expressions can be translated to posets. This provides a method to determine the equality of two logic expressions. Formally, two logic expressions are equivalent if their associated posets are equivalent. This admits the

development of an algebra over logic expressions. Though logic expressions look very much like boolean expressions, boolean logic does not work on them. For example,

```
a * ( a + b )
```

can be reduced to `a` in boolean logic, but can only be reduced to `a + (a * b)` in the case of logic expressions. In addition, laws like

```
( a < b ) + ( b < a ) = a + b
```

can be derived for logic expressions that have no analog in boolean logic.

9.2.3 Impact on Reduction

The formalism for translating input logic expressions to posets currently gives meaning to some peculiar logic expressions, for example,

```
( a < b ) * ( b < a)
```

where it is not not obvious that they should have any meaning at all. A reasonable meaning and justification can be created for such an expression, but it is not clear that a user would ever write such an expression and an alternative would be to declare such expressions as being semantically in error. However, such expressions may be generated automatically during the process of reduction. It is for this reason that they were included as valid at this point. If, however, it is determined that such expressions will not be generated during reduction, there may be compelling reasons to declare them as invalid. In general, the impact of this formalism on reduction merits further study.

Bibliography

[1] Carlo H. Séquin. Managing VLSI complexity: An outlook. *Proceedings of the IEEE*, 71(1):149–166, January 1983.

[2] C. A. Mead. Structural and behavioral composition of VLSI. In F. Anceau and E. J. Aas, editors, *VLSI83—Proceedings of the IFIP TC10/WG10.5 International Conference on Very Large Scale Integration*, pages 3–8, Trondheim, Norway, 16–19 August 1983. North Holland.

[3] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor. Magic: A VLSI layout system. In *Proceedings of the 21st ACM/IEEE Design Automation Conference*, pages 152–159, Albuquerque, New Mexico, June 1984.

[4] Ronald L. Rivest. The 'PI' (placement and interconnect) system. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 475–481, Las Vegas, NV, June 14–16 1982.

[5] Alan T. Sherman. *VLSI Placement and Routing: The PI Project.* Springer-Verlag, New York, NY, 1989.

[6] Jay R. Southard. Macpitts: An approach to silicon compilation. *IEEE Computer*, 16(12):74–82, December 1983.

[7] S. Director, A. Parker, D. Siewiorek, and D. Thomas. A design methodology and computer aids for digital VLSI systems. *IEEE Transactions on Circuits and Systems*, CAS-28(7):634–645, July 1981.

[8] David Johannsen. Bristle blocks: A silicon compiler. In *Proceedings of the 16th ACM/IEEE Design Automation Conference*, pages 310–313, San Diego, CA, June 1979.

[9] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE Software*, 1(4):10–26, October 1984.

[10] D. D. Gajski and R. H. Kuhn. Guest editors' introduction: New VLSI tools. *IEEE Computer*, 16(12):11–14, December 1983.

[11] P. N. Hilfinger. A high-level language and silicon compiler for digital signal processing. In *Proceedings of the 1985 IEEE Custom Integrated Circuits Conference*, pages 213–216, Portland, Oregon, May 20–23 1985.

[12] J. D. Morison, N. E. Peeling, and E. V. Whiting. Sequential programming extensions to ELLA, with automatic transformation to structure. In *1987 IEEE International Conference on Computer Design*, pages 571–576, Rye Brook, NY, October 1987.

[13] J. D. Morison, N. E. Peeling, and T. L. Thorp. Ella: A hardware description language. In *ICCC 82: Proceedings of the 2nd IEEE International Conference on Circuits and Computers*, pages 604–607, New York, N.Y., September 28–October 1 1982.

[14] Ehud Shapiro. Concurrent Prolog: A progress report. *IEEE Computer*, 19(8):44–58, August 1986.

[15] Norihisa Suzuki. Concurrent Prolog as an efficient VLSI design language. In *IEEE Computer* [118], pages 33–40.

[16] George J. Milne. CIRCAL: A calculus for circuit description. Computer Science Department Report CSR-122-82, University of Edinburgh, July 1982.

[17] George J. Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985. Also published as Computer Science Department Report CSR-151-83, University of Edinburgh, November 1983.

[18] Mary Sheeran. *μFP—An Algebraic VLSI Design Language.* PhD thesis, Oxford University Computing Laboratory, Oxford, U.K., November 1983. Available as Technical Monograph PRG-39, September 1984.

[19] Dorab Patel, Martine Schlag, and Miloš Ercegovac. $\nu\mathcal{FP}$: An environment for the multi-level specification, analysis, and synthesis of hardware algorithms. In J.-P. Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 238–255, Nancy, France, September 1985. Springer-Verlag.

[20] Moe Shahdad, Roger Lipsett, Eric Marschner, Kellye Sheehan, Howard Cohen, Ron Waxman, and Dave Ackley. VHSIC hardware description language. In *IEEE Computer* [118], pages 94–103.

[21] Robin Milner. A proposal for standard ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming* [119], pages 184–197.

[22] David MacQueen. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming* [119], pages 198–207.

[23] Mauro Pezzé. Behavioural abstraction and circuit verification using Circal. Computer Science Department Internal Report CSR-251-87, University of Edinburgh, November 1987.

[24] Laurence W. Nagel. SPICE2: A computer program to simulate semiconductor circuits. ERL Memo ERL-M520, Electronics Research Laboratory, University of California at Berkeley, Berkeley, CA, May 1975.

[25] M. C. Chen and C. A. Mead. A hierarchical simulator based on formal semantics. In R. Bryant, editor, *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 207–223. California Institute of Technology, Computer Science Press, March 1983.

[26] Robert Piloty, Mario Barbacci, Dominique Borrione, Donald Dietmeyer, Frederick Hill, and Patrick Skelly. An overview of CONLAN: a formal construction method for hardware description languages. In Simon Lavington, editor, *Information Processing 80: Proceedings of the IFIP Congress 80*, pages 199–204, Tokyo, Japan and Melbourne, Australia, 6–9 and 14–17 October 1980. North Holland.

[27] Ben Moszkowski. A temporal logic for multilevel reasoning about hardware. In *IEEE Computer* [118], pages 10–19.

[28] Ben Moszkowski. *Executing temporal logic programs.* Cambridge University Press, Cambridge, UK, 1986.

[29] Gerald Estrin, Robert S. Fenchel, Rami R. Razouk, and Mary K. Vernon. SARA (System ARchitects Apprentice): Modeling, analysis, and simulation support for design of concurrent systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, February 1986.

[30] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[31] R. L. Blackburn and D. E. Thomas. Linking the behavioral and structural domains of representation in a synthesis system. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference* [120], pages 374–380.

[32] R. L. Blackburn, D. E. Thomas, and P. M. Koenig. CORAL II: Linking behavior and structure in an IC design system. In *Proceedings of the 25th ACM/IEEE Design Automation Conference* [121], pages 529–535.

[33] Telle Whitney. A hierarchical design analysis front end. In Gray [122], pages 217–225.

[34] Jim A. Rowson. *Understanding Hierarchical Design.* PhD thesis, California Institute of Technology, Pasadena, CA, 1980.

[35] Jim Rowson and Martin Newell. Geometry composition—another look. SSP File 3792, Silicon Structures Project of the California Institute of Technology, Pasadena, CA, June 26, 1980.

[36] Peter Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming* [123], pages 179–187.

[37] Luca Cardelli. *An Algebraic Approach to Hardware Description and Verification.* PhD thesis, Department of Computer Science, University of Edinburgh, 1982. Published as Report CST-16-82.

[38] George Johnstone Milne. *A Mathematical Model of Concurent Computation.* PhD thesis, Department of Computer Science, University of Edinburgh, 1978. Published as Report CST-4-78.

[39] Frederick J. Hill. Introducing AHPL. *IEEE Computer*, 7(12):27–44, December 1974.

[40] David S. Harrison, Peter Moore, Rick L. Spickelmier, and A. Richard Newton. Data management and graphics editing in the Berkeley design environment. In *ICCAD-86: Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 24–27, Santa Clara, CA, November 11-13 1986.

[41] Bart Locanthi. LAP: A SIMULA package for IC layout. SSP Display File 1862, Silicon Structures Project of the California Institute of Technology, Pasadena, CA, 24 July 1978.

[42] O.J. Dahl and K. Nygaard. SIMULA—an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

[43] John Batali, Neil Mayle, Howard Shrobe, Gerald Sussman, and Daniel Weise. The DPL/Daedalus design environment. In Gray [122], pages 183–192.

[44] John McCarthy. Recursive functions of symbol expressions and their computation by machine. *Communications of the ACM*, 3(4):184–194, April 1960.

[45] Jay R. Southard. Algorithmic system compilation: Silicon compilation for systems designers. In Gajski [124], chapter 6, pages 153–203.

[46] Peter Denyer and David Renshaw. *VLSI Signal Processing: A Bit-Serial Approach.* Addison-Wesley Publishing Company, Inc., Reading, MA, 1985.

[47] Peter A. Ruetz, Rajeev Jain, Chuen-Shen Shung, Jan M. Rabaey, Gordon M. Jacobs, and Robert W. Brodersen. Automatic layout generation of real-time image processing circuits. In *Proceedings of the IEEE 1986 Custom Integrated Circuits Conference*, pages 111–115, Rochester, N.Y., May 12–15 1986.

[48] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, and F. Catthoor. CATHEDRAL-II: A synthesis system for multiprocessor DSP systems. In Gajski [124], chapter 8, pages 311–360.

[49] Nicolas Halbwachs and Daniel Pilaud. Use of a real-time declarative language for systolic array design and simulation. In Moore et al. [125], pages 81–90.

[50] Geraint Jones and Wayne Luk. Exploring designs by circuit transformation. In Moore et al. [125], pages 91–98.

[51] R. K. Brayton, R. Camposano, G. De Micheli, R. H. J. M. Otten, and J. van Eijndhoven. The Yorktown silicon compiler system. In Gajski [124], chapter 7, pages 204–310.

[52] John Backus. Programming language semantics and closed applicative languages. In *Conference Record of the ACM Symposium on the Principles of Programming Languages*, pages 71–86, Boston, MA, October 1–3 1973.

[53] John Backus. Reduction languages and variable free programming. Research Report RJ 1010, IBM Yorktown Heights, NY, April 1972.

[54] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science University of Edinburgh, March 1986.

[55] Martine Denise Francoise Schlag. *Layout From A Topological Description.* Ph. D. dissertation, UCLA Computer Science Department, Los Angeles, California, July 1986. Available as Technical Report CSD-860039.

[56] Robert N. Mayo and John K. Ousterhout. Pictures with parenthesis: Combining graphics and procedures in a VLSI layout tool. In *Proceedings of the 20th ACM/IEEE Design Automation Conference*, pages 270–276, Miami, FL, 1983.

[57] John Batali and Anne Hartheimer. The Design Procedure Language. A.I. Memo 598, Massachusetts Institute of Technology, September 1980.

[58] Stephen Trimberger. Combining graphics and layout language in a single interactive system. SSP File 3784, California Institute of Technology Silicon Structures Project, Pasadena, CA, June 8, 1980.

[59] Paul J. Asente. Editing graphical objects using procedural representations. Research Report 87/6, DEC Western Research Laboratories, Palo Alto, CA, November 1987.

[60] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[61] D. O. Lahti. Applications of a functional programming language. Master's thesis, UCLA Computer Science Department, Los Angeles, California, April 1981. Available as Technical Report CSD-810403.

[62] Steven Johnson. *Synthesis of Digital Designs from Recursion Equations.* MIT Press, 1984.

[63] Luca Cardelli and Gordon Plotkin. An algebraic approach to VLSI design. In Gray [122], pages 173–192.

[64] F. Meshkinpour and M. D. Ercegovac. A functional language for description and design of digital systems: Sequential constructs. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference* [120], pages 238–244.

[65] Mary Sheeran. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming* [119], pages 104–112.

[66] H. T. Kung and W. T. Lin. An algebra for VLSI design. Technical Report CMU-CS-84-100, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, April 1983.

[67] Lennart Johnsson, Danny Cohen, Uri Weiser, and Alan L. Davis. Towards a formal treatment of VLSI arrays. In Charles L. Seitz, editor, *Proceedings of the Second Caltech Conference on Very Large Scale Integration*, pages 375–398, January 19–21 1981.

[68] Michael C. McFarland S.J., Alice C. Parker, and Raul Camposano. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference* [121], pages 330–336.

[69] M. C. McFarland. The VT: A data base for automated digitial design. Report DRC-01-4-80, Design Research Center, Carnegie Mellon University, Pittsburgh, PA, December 1978.

[70] David W. Knapp and Alice C. Parker. A unified representation for design information. In C. J. Koomen and T. Moto-oka, editors, *Proceedings of the 7th International Symposium on Computer Hardware Description Langauges and Their Applications*, pages 337–353, Tokyo, Japan, August 29–31 1985. Elsevier Science Publishers B.V. (North Holland).

[71] Alex Orailoglu and Daniel D. Gajski. Flow graph representation. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference* [126], pages 503–509.

[72] Alice C. Parker, Jorge "T" Pizarro, and Mitch Milnar. MAHA: A program for datapath synthesis. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference* [126], pages 461–466.

[73] Thaddeus J. Kowlaski. The VLSI design automation assistant: An architecture compiler. In Gajski [124], chapter 5, pages 122–152.

[74] Marina Chien-mei Chen. *Space-Time Algorithms: Semantics and Methodology.* PhD thesis, California Institute of Technology, Pasadena, CA, 1983.

[75] Björn Lisper. *Synthesizing Synchronous Systems by Static Scheduling in Space-Time.* Number 362 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

[76] John Backus. The algebra of functional programs: Function level reasoning, linear equations, and extended definitions. In J. Diaz and I. Ramos, editors, *Proceedings of the International Colloqium on Formalization of Programming Concepts*, number 107 in Lecture Notes in Computer Science, pages 1–43. Springer-Verlag, 1981.

[77] J. Barkley Rosser. Highlights of the history of the lambda calculus. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming* [123], pages 216–225.

[78] John Backus, John H. Williams, and Edward L. Wimmers. FL language manual (preliminary version). Research Report RJ 5339 (54809), IBM Almaden Research Center, San Jose, CA, 7 November 1986.

[79] Scott B. Baden and Dorab R. Patel. Berkeley FP — Experiences with a Functional Programming Language. In *Conference Record COMPCON Spring 1983*, pages 274–277, San Francisco, 1983.

[80] Mark Sausville. Gathering performance statistics on hardware specified in FP. Internal report, University of California at Los Angeles, Los Angeles, CA, March 20 1986.

[81] Rami R. Razouk, Mary Vernon, and Gerald Estrin. Evaluation methods in SARA—the graph model simulator. In *Proceedings of the 1979 Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 189–206, Boulder, CO, August 1979.

[82] Gerald Estrin. A methodology for design of digital systems—supported by SARA at the age of one. In *AFIPS Conference Proceedings*, volume 47, pages 313–324, 1978.

[83] Peter Naur (editor). Report on the algorithmic language ALGOL60. *Communications of the ACM*, 3(5):299–314, May 1960.

[84] Mary Katharine Vernon. *Performance-Oriented Design of Distributed Systems.* PhD thesis, University of California at Los Angeles, Los Angeles, California, 1982. Available as Technical Report CSD-821217 dated December 1982, published August 1983.

[85] Rami R. Razouk. *Computer-Aided Design and Evaluation of Digital Computer Systems.* PhD thesis, University of California at Los Angeles, Los Angeles, CA, 1981. Available as Technical Report CSD-810205 dated February 1981.

[86] Kim Gostelow. *Flow of Control, Resource Allocation, and the Proper Termination of Programs.* PhD thesis, University of California at Los Angeles, Los Angeles, California, 1971. Available as Technical Report UCLA-ENG-7179.

[87] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, LAMBDA: The ultimate software tool. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming* [123], pages 114–122.

[88] Mario Barbacci. Instruction set processor specifications for simulation evaluation, and synthesis. In *Proceedings 16th Design Automation Conference*, pages 64–72, San Diego, CA, June 1979.

[89] William T. Overman. *Verification of Concurrent Systems: Function and Timing.* PhD thesis, University of California at Los Angeles, Los Angeles, California, August 1981. Available as Technical Report CSD-810814.

[90] Ivan Campos. *Multilevel Modeling for Synthesis of Reliable Concurrent Software Systems.* PhD thesis, University of California at Los Angeles, Los Angeles, California, 1977.

[91] Wilson Ruggiero. *A Distributed Data and Control Driven Machine: Programming and Architecture.* PhD thesis, University of California at Los Angeles, Los Angeles, California, November 1978. Available as Technical Report UCLA-ENG-7878.

[92] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[93] Eduardo Aaron Krell. *System ARchitect's Apprentice (SARA) as the Foundation for a Methodology-Oriented Ada Programming Support Environment.* PhD thesis, University of California at Los Angeles, Los Angeles, California, January 1987. Available as Technical Report CSD-870001.

[94] Shiu-Kai Chin and Kevin J. Greene. Verifiable and executable theories of design for synthesizing correct hardware. In *Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 604–610, Rye Brook, NY, 3–5 October 1988.

[95] Donald E. Thomas, Charles Y. Hitchcock III, Thaddeus J. Kowalski, Jayanth V. Rajan, and Robert Walker. Automatic data path synthesis. *IEEE Computer*, 16(12):59–70, December 1983.

[96] Chia-Jeng Tseng and Daniel P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer Aided Design*, CAD-5(3):379–395, July 1986.

[97] Alice Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim. The CMU design automation system. In *Proceedings 16th Design Automation Conference*, pages 73–80, San Diego, CA, June 1979.

[98] Howard E. Shrobe. The data path generator. In *1982 MIT Conference on Advanced Research in VLSI*, pages 175–181, Cambridge, Mass., January 1982.

[99] Farshad Meshkinpour. On specification and design of digital systems using an applicative hardware description language. Technical Report CSD-840046, UCLA Computer Science Department, Los Angeles, California, November 1984.

[100] Winthrop John Wu. FLAG: An FP based VLSI layout generator. Master's thesis, University of California at Los Angeles, Los Angeles, CA, 1989.

[101] C. D. Rogers, S.W. Daniel, and J.B. Rosenberg. An overview of VIVID, MCNC's vertically integrated symbolic design system. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference* [120], pages 62–68.

[102] J. B. Rosenberg. Auto-interactive schematics to layout translation. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference* [120], pages 82–87.

[103] Jan M. Rabaey, Stephen P. Pope, and Robert W. Brodersen. An integrated automated layout generation system for DSP circuits. *IEEE Transactions on Computer Aided Design*, CAD-4(3):285–296, July 1985.

[104] Robert W. Hon and Carlo H. Séquin. A guide to LSI implementation. Technical report, Xrox PARC, Palo Alto, CA, January 1980.

[105] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(2):226–231, June 1960.

[106] Mary Sheeran. Retiming and slowdown in regular array design. In Cordelia Hall, John Hughes, and John T. O'Donnell, editors, *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, pages 161–186, Rothesay, Isle of Bute, August 2-5, 1988. Computer Science Department, University of Glasgow. Also published as Research Report 89/R4, dated February 1989.

[107] Mary Sheeran and Geraint Jones. Relations + higher order functions = hardware descriptions. In Walter E. Proebster and Hans Reiner, editors, *IEEE CompEuro87: Proceedings of the First International Conference on Computer Technology, Systems and Applications*, pages 303–306, Hamburg, W. Germany, 11–15 May 1987.

[108] Aart J. de Geus and William Cohen. A rule-based system for optimizing combinational logic. *IEEE Design and Test of Computers*, 2(4):22–32, August 1985.

[109] David Gregory, Karen Bartlett, Aart de Geus, and Gary Hachtel. SOCRATES: A system for automatically synthesizing and optimizing combinational logic. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference* [126], pages 79–85.

[110] Shiu-Kai Chin and Edward P. Stabler. Synthesis of digital designs by equivalence transformations. In *Proceedings of the 1986 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 558–560, Port Chester, NY, 6–9 October 1986.

[111] John A. Darringer, Jr. William H. Joyner, C. Leonard Berman, and Louise Trevillyan. Logic synthesis through local transformations. *IBM Journal of Research and Development*, 25(4):272–280, July 1981.

[112] Raul Camposano. Structural synthesis in the Yorktown Silicon Compiler. In Carlo H. Séquin, editor, *VLSI'87: VLSI Design of Digital Systems: Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration*, pages 61–72, Vancouver, Canada, 10–12 August 1987. Elsevier Science Publishers B.V. (North Holland).

[113] G. Zimmermann. The MIMOLA design system: a computer aided digital processor design method. In *Proceedings of the 16th ACM/IEEE Design Automation Conference*, pages 53–58, San Diego, CA, June 1979.

[114] Ben M. Huey and Fredrick J. Hill. Fault test generation using a design language. In *Proceedings of the 1975 International Symposium on Computer Hardware Description Langauges and Their Applications*, pages 91–95, N.Y., NY, September 3–5 1975.

[115] E. B. Eichelberger and T. W. Williams. A logic design structure for LSI testability. In *Proceedings of the 14th ACM/IEEE Design Automation Conference*, pages 462–468, New Orleans, LA, June 1977.

[116] M. Hamilton and S. Zeldin. The relationship between design and verification. *The Journal of Systems and Software*, 1:29–56, 1979.

[117] Margaret Hamilton and Saydean Zeldin. Higher Order Software—a methodology for defining software. *IEEE Transactions on Software Engineering*, SE-2(1):9–32, March 1976.

[118] *IEEE Computer*, 18(2), February 1985.

[119] *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, PA, August 1984.

[120] *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, Las Vegas, NV, June 23–26 1985.

[121] *Proceedings of the 25th ACM/IEEE Design Automation Conference*, Anahiem, CA, June 1988.

[122] John P. Gray, editor. *VLSI81: Proceedings of the First International Conference on Very Large Scale Integration.* Academic Press, August 1981.

[123] *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, Austin, TX, August 1982.

[124] Daniel D. Gajski, editor. *Silicon Compilation.* Addison-Wesley Publishing Company, 1988.

[125] Will Moore, Andrew McCabe, and Roddy Urquhart, editors. *Proceedings of the First International Workshop on Systolic Arrays*, Oxford, UK, 2–4 July 1986. Adam Hilger.

[126] *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, Las Vegas, NV, June 29 – July 2 1986.

Appendix A

$\nu\mathcal{FP}$ Semantics

A complete description of the $\nu\mathcal{FP}$ language is provided here.

A.1 Objects

The set of $\nu\mathcal{FP}$ objects (Ω)is the same as those in FP. It consists of atoms and sequences

$$\langle x_0, \ldots, x_k \rangle$$

where

$$\forall i,\ 0 \leq i \leq k,\ x_i \in \Omega$$

Atoms uniquely determine the set of valid objects and consist of numbers and quoted ascii strings ("abcd"). There are three predefined atoms, **T** and **F**, that correspond to the logical values `true` and `false`, and the undefined atom $\perp$. *Bottom* denotes the value returned as the result of an undefined operation, e.g., division by zero. The empty sequence, $\langle\rangle$ is also an atom. The following are examples of valid $\nu\mathcal{FP}$ objects:

$\perp$	3.1415	1234567
ab	"CD"	$\langle 1, \langle 2, 3 \rangle \rangle$
$\langle\rangle$	**T**	$\langle a, \langle\rangle \rangle$

If any element of a sequence is undefined, by definition, the sequence itself is undefined. Hence

$$\langle \ldots, \perp, \ldots \rangle \equiv \perp$$

This property is the "bottom-preserving property" mentioned in [60].

A.2 Application

There is only one operation in $\nu\mathcal{FP}$, application, denoted by the colon (":"). Given a function σ and an object x, $\sigma : x$ is an application and its meaning is the object that results from applying σ to x (i.e. evaluating $\sigma(x)$). σ is the *operator* and x is the *operand.* The following are examples of applications:

$$+ : \langle 8,7\rangle \equiv 15 \qquad \mathtt{tl} : \langle 1,2,3\rangle \equiv \langle 2,3\rangle$$
$$1 : \langle a,b,c,d\rangle \equiv a \qquad 2 : \langle a,b,c,d\rangle \equiv b$$

A.3 Functions

All functions map objects into objects. $\nu\mathcal{FP}$ functions are strict.

$$\sigma : \bot \equiv \bot;\ \forall \sigma \in \mathcal{F}$$
$$\langle \ldots, \bot, \ldots\rangle \equiv \bot$$

A modification of McCarthy's conditional expression [44] is used to describe the semantics of $\nu\mathcal{FP}$ functions and combining forms.

$$p_1 \Longrightarrow e_1;\ \ldots;\ p_n \Longrightarrow e_n;\ e_{n+1}$$

This means that the expression e_1 is evaluated if p_1 is `true`. If not, then e_2 is evaluated if p_2 is `true`. And, so on. If none of the predicates p_i is `true`, then evaluate e_{n+1}. If any of the predicates p_i evaluate to $\bot$, the result of the whole expression is $\bot$.

A.3.1 Selector Functions

For a nonzero integer, μ

$$\mu : x \equiv$$
$$x = \langle x_1, \ldots, x_k\rangle \wedge 0 < \mu \leq k \Longrightarrow x_\mu;$$
$$x = \langle x_1, \ldots, x_k\rangle \wedge -k \leq \mu < 0 \Longrightarrow x_{k+\mu+1};$$
$$\bot$$

$$\mathtt{id} : x \equiv x$$

$$\texttt{last}:x \equiv \begin{array}{l} x = \langle\rangle \Longrightarrow \langle\rangle; \\ x = \langle x_1, \ldots, x_k \rangle \ \wedge \ k \geq 1 \Longrightarrow x_k; \\ \bot \end{array}$$

$$\texttt{first}:x \equiv \begin{array}{l} x = \langle\rangle \Longrightarrow \langle\rangle; \\ x = \langle x_1, \ldots, x_k \rangle \ \wedge \ k \geq 1 \Longrightarrow x_1; \\ \bot \end{array}$$

$$\texttt{tl}:x \equiv \begin{array}{l} x = \langle x_1 \rangle \Longrightarrow \langle\rangle; \\ x = \langle x_1, \ldots, x_k \rangle \ \wedge \ k \geq 2 \Longrightarrow \langle x_2, \ldots, x_k \rangle; \\ \bot \end{array}$$

$$\texttt{tlr}:x \equiv \begin{array}{l} x = \langle x_1 \rangle \Longrightarrow \langle\rangle; \\ x = \langle x_1, \ldots, x_k \rangle \ \wedge \ k \geq 2 \Longrightarrow \langle x_1, \ldots, x_{k-1} \rangle; \\ \bot \end{array}$$

A.3.2 Structure Modifying Functions

$$\texttt{distl}:x \equiv \begin{array}{l} x = \langle y, \langle\rangle \rangle \Longrightarrow \langle\rangle; \\ x = \langle y, \langle z_1, \ldots, z_k \rangle \rangle \Longrightarrow \langle \langle y, z_1 \rangle, \ldots, \langle y, z_k \rangle \rangle; \\ \bot \end{array}$$

$$\texttt{distr}:x \equiv$$

$$x = \langle\langle\rangle, y\rangle \Longrightarrow \langle\rangle;$$
$$x = \langle\langle y_1, \ldots, y_k\rangle, z\rangle \Longrightarrow \langle\langle y_1, z\rangle, \ldots, \langle y_k, z\rangle\rangle;$$
$$\bot$$

$\texttt{apndl} : x \equiv$

$$x = \langle y, \langle\rangle\rangle \Longrightarrow \langle y\rangle;$$
$$x = \langle y, \langle z_1, \ldots, z_k\rangle\rangle \Longrightarrow \langle y, z_1, z_2, \ldots, z_k\rangle;$$
$$\bot$$

$\texttt{apndr} : x \equiv$

$$x = \langle\langle\rangle, z\rangle \Longrightarrow \langle z\rangle;$$
$$x = \langle\langle y_1, \ldots, y_k\rangle, z\rangle \Longrightarrow \langle y_1, y_2, \ldots, y_k, z\rangle;$$
$$\bot$$

$\texttt{trans} : x \equiv$

$$x = \langle\langle\rangle, \ldots, \langle\rangle\rangle \Longrightarrow \langle\rangle;$$
$$x = \langle x_1, \ldots, x_k\rangle \Longrightarrow \langle y_1, \ldots, y_m\rangle;$$
$$\bot$$

where

$$x_i = \langle x_{i,1}, \ldots, x_{i,m}\rangle \ \wedge \ y_j = y_{1,j}, \ldots, y_{k,j}\rangle;\ 1 \leq i \leq k,\ 1 \leq j \leq m$$

$\texttt{reverse} : x \equiv$

$$x = \langle\rangle \Longrightarrow \langle\rangle;$$
$$x = \langle x_1, \ldots, x_k\rangle \Longrightarrow \langle x_k, \ldots, x_1\rangle;$$
$$\bot$$

$\texttt{rotl} : x \equiv$

$$x = \langle\rangle \Longrightarrow \langle\rangle;$$

$$
\begin{aligned}
& x = \langle x_1 \rangle \Longrightarrow \langle x_1 \rangle; \\
& x = \langle x_1, \ldots, x_k \rangle \ \wedge \ k \geq 2 \Longrightarrow \langle x_2, \ldots, x_k, x_1 \rangle; \\
& \bot \\
\texttt{rotr} : x \ \equiv & \\
& x = \langle\rangle \Longrightarrow \langle\rangle; \\
& x = \langle x_1 \rangle \Longrightarrow \langle x_1 \rangle; \\
& x = \langle x_1, \ldots, x_k \rangle \ \wedge \ k \geq 2 \Longrightarrow \langle x_k, x_1, \ldots, x_{k-2}, x_{k-1} \rangle; \\
& \bot \\
\texttt{concat} : x \ \equiv & \\
& x = \langle \langle x_{11}, \ldots, x_{1k} \rangle \ldots \langle x_{m1}, \ldots, x_{mp} \rangle \rangle \ \wedge \ k, m, n, p > 0 \\
& \quad \Longrightarrow \langle x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots x_{m1}, \ldots, x_{mp} \rangle; \\
& \bot
\end{aligned}
$$

Concatenate removes all occurrences of the null sequence:

$$\texttt{concat} : \langle \langle 1,3 \rangle, \langle\rangle, \langle 2,4 \rangle, \langle\rangle, 5 \rangle \rangle \ \equiv \ \langle 1,3,2,4,5 \rangle$$

$$
\begin{aligned}
\texttt{pair} : x \ \equiv & \\
& x = \langle x_1, \ldots, x_k \rangle \ \wedge \ k > 0 \ \wedge \ k \text{ is even} \\
& \quad \Longrightarrow \langle \langle x_1, x_2 \rangle, \ldots, \langle x_{k-1}, x_k \rangle \rangle; \\
& x = \langle x_1, \ldots, x_k \rangle \ \wedge \ k \geq 0 \ \wedge \ k \text{ is odd} \\
& \quad \Longrightarrow \langle \langle x_1, x_2 \rangle, \ldots, \langle x_k \rangle \rangle; \\
& \bot \\
\texttt{split} : x \ \equiv & \\
& x = \langle x_1 \rangle \Longrightarrow \langle \langle x_1 \rangle, \langle\rangle \rangle; \\
& x = \langle x_1, \ldots, x_k \rangle \ \wedge \ k > 1 \\
& \quad \Longrightarrow \langle \langle x_1, \ldots, x_{\lceil k/2 \rceil} \rangle, \langle x_{\lceil k/2 \rceil + 1}, \ldots, x_k \rangle \rangle; \\
& \bot
\end{aligned}
$$

A.3.3 Predicate (Test) Functions

$$\texttt{atom} : x \ \equiv$$

$$x \in \mathit{atoms} \Longrightarrow \texttt{true};$$
$$x \neq \bot \Longrightarrow \texttt{false};$$
$$\bot$$

$$\texttt{eql} : x \equiv$$
$$x = \langle y, z \rangle \wedge y = z \Longrightarrow \texttt{true};$$
$$x = \langle y, z \rangle \wedge y \neq z \Longrightarrow \texttt{false};$$
$$\bot$$

Less than ($<$), greater than ($>$), greater than or equal ($\geq$), less than or equal ($\leq$), not equal (neql) are defined similarly. "=" is a synonym for eql.

$$\texttt{null} : x \equiv$$
$$x = \langle\rangle \Longrightarrow \texttt{true};$$
$$x \neq \bot \Longrightarrow \texttt{false};$$
$$\bot$$

A.3.4 Logical Conjunctions

$$\texttt{and} : \langle x, y \rangle \equiv$$
$$x = \texttt{true} \wedge y \in \{\texttt{true}, \texttt{false}\} \Longrightarrow y;$$
$$x = \texttt{false} \wedge y \in \{\texttt{true}, \texttt{false}\} \Longrightarrow \texttt{false};$$
$$\bot$$
$$\texttt{or} : \langle x, y \rangle \equiv$$
$$x = \texttt{false} \wedge y \in \{\texttt{true}, \texttt{false}\} \Longrightarrow y;$$
$$x = \texttt{true} \wedge y \in \{\texttt{true}, \texttt{false}\} \Longrightarrow \texttt{true};$$
$$\bot$$
$$\texttt{not} : x \equiv$$

$$
\begin{array}{rcl}
 & & x = \texttt{true} \Longrightarrow \texttt{false}; \\
 & & x = \texttt{false} \Longrightarrow \texttt{true}; \\
 & & \bot \\
\texttt{xor} : \langle x, y \rangle & \equiv & \\
 & & x, y \in \{\texttt{true}, \texttt{false}\} \ \wedge \ x = y \Longrightarrow \texttt{false}; \\
 & & x, y \in \{\texttt{true}, \texttt{false}\} \ \wedge \ x \neq y \Longrightarrow \texttt{true}; \\
 & & \bot
\end{array}
$$

A.3.5 Arithmetic Functions

$$
\begin{array}{rcl}
+ : x & \equiv & \\
 & & x = \langle y, z \rangle \ \wedge \ y, z \text{ are numbers} \Longrightarrow y + z; \\
 & & \bot \\
- : x & \equiv & \\
 & & x = \langle y, z \rangle \ \wedge \ y, z \text{ are numbers} \Longrightarrow y - z; \\
 & & \bot \\
* : x & \equiv & \\
 & & x = \langle y, z \rangle \ \wedge \ y, z \text{ are numbers} \Longrightarrow y \times z; \\
 & & \bot \\
/ : x & \equiv & \\
 & & x = \langle y, z \rangle \ \wedge \ y, z \text{ are numbers} \Longrightarrow y \div z; \\
 & & \bot
\end{array}
$$

A.3.6 Circuit (Gate-Level) Primitives

$$
\begin{array}{rcl}
\texttt{andg} : \langle x, y \rangle & \equiv & \\
 & & x = 1 \ \wedge \ y \in \{0, 1\} \Longrightarrow y; \\
 & & x = 0 \ \wedge \ y \in \{0, 1\} \Longrightarrow 0; \\
 & & \bot \\
\texttt{org} : \langle x, y \rangle & \equiv & \\
 & & x = 0 \ \wedge \ y \in \{0, 1\} \Longrightarrow y;
\end{array}
$$

$$
\begin{aligned}
& x = 1 \ \wedge \ y \in \{0,1\} \Longrightarrow 1; \\
& \bot \\
\texttt{xorg} : \langle x, y \rangle \ \equiv & \\
& x, y \in \{0,1\} \ \wedge \ x = y \Longrightarrow 0; \\
& x, y \in \{0,1\} \ \wedge \ x \neq y \Longrightarrow 1; \\
& \bot \\
\texttt{nandg} : \langle x, y \rangle \ \equiv & \\
& x = 1 \ \wedge \ y \in \{0,1\} \Longrightarrow \overline{y}; \\
& x = 0 \ \wedge \ y \in \{0,1\} \Longrightarrow 1; \\
& \bot
\end{aligned}
$$

$$
\begin{aligned}
\texttt{norg} : \langle x, y \rangle \ \equiv & \\
& x = 0 \ \wedge \ y \in \{0,1\} \Longrightarrow \overline{y}; \\
& x = 1 \ \wedge \ y \in \{0,1\} \Longrightarrow 0; \\
& \bot \\
\texttt{notg} : x \ \equiv & \\
& x = 1 \Longrightarrow 0; \\
& x = 0 \Longrightarrow 1; \\
& \bot
\end{aligned}
$$

A.3.7 Mathematical Library Routines

$$
\begin{aligned}
\texttt{sin} : x \ \equiv & \\
& x \text{ is a number} \Longrightarrow \sin(x); \\
& \bot \\
\texttt{asin} : x \ \equiv & \\
& x \text{ is a number} \ \wedge \ \mid x \mid \leq 1 \Longrightarrow \sin^{-1}(x); \\
& \bot
\end{aligned}
$$

$$\texttt{cos} : x \equiv$$
$$x \text{ is a number} \Longrightarrow \cos(x);$$
$$\bot$$

$$\texttt{acos} : x \equiv$$
$$x \text{ is a number} \wedge \mid x \mid \leq 1 \Longrightarrow \cos^{-1}(x);$$
$$\bot$$

$$\texttt{exp} : x \equiv$$
$$x \text{ is a number} \Longrightarrow e^x;$$
$$\bot$$

$$\texttt{log} : x \equiv$$
$$x \text{ is a positive number} \Longrightarrow \ln x;$$
$$\bot$$

$$\texttt{mod} : \langle x, y \rangle \equiv$$
$$x, y \text{ are numbers} \Longrightarrow x - y \times \left\lfloor \frac{x}{y} \right\rfloor;$$
$$\bot$$

$$\texttt{sqrt} : \langle x, y \rangle \equiv$$
$$x \text{ is a positive number} \Longrightarrow \sqrt{x};$$
$$\bot$$

A.3.8 Miscellaneous Functions

$$\texttt{length} : x \equiv$$
$$x = \langle x_1, \ldots, x_k \rangle \Longrightarrow k;$$
$$x = \langle\rangle \Longrightarrow 0;$$
$$\bot$$

$$\texttt{sw} : \langle p, x, y \rangle \equiv$$
$$p = \texttt{true} \wedge x, y \text{ have the same form} \Longrightarrow x;$$

$$p = \texttt{false} \wedge x, y \text{ have the same form } \Longrightarrow y;$$
$$\bot$$

A.4 Combining Forms

A combining form takes functions as parameters and returns a new function that is a combination of its functional parameters. Combining forms manipulate functions, while functions operators manipulate values.

For two functions ϕ and ψ, the form $\phi \circ \psi$ denotes their composition:

$$(\phi \circ \psi) : x \equiv \phi : (\psi : x),\ \forall x \in \Omega$$

The *constant* function takes an object parameter:

$$\%x : y \equiv (y = \bot \Longrightarrow \bot; x),\ \forall x, y \in \Omega$$

The function $\%\bot$ always returns $\bot$.

In the following description of the combining forms, we assume that ξ, ξ_i, σ, σ_i, τ, τ_i are functions and that x, x_i, y, y_i, z_i are objects.

A.4.1 Compose

$$(\sigma \circ \tau) : x \equiv \sigma : (\tau : x)$$

A.4.2 Construct

$$[\sigma_1, \ldots, \sigma_n] : x \equiv \langle \sigma_1 : x, \ldots, \sigma_n : x \rangle$$

A.4.3 Apply-to-All

$$\&\sigma : x \equiv$$
$$x = \langle\rangle \Longrightarrow \langle\rangle$$
$$x = \langle x_1, \ldots, x_k \rangle \Longrightarrow \langle \sigma : x_1, \sigma : x_2, \ldots, \sigma : x_k \rangle$$
$$\bot$$

A.4.4 Conditional

$$\begin{aligned}\textbf{if } \xi \textbf{ then } \sigma \textbf{ else } \tau \textbf{ fi } : x \;\equiv\; & \\ & (\xi : x) = \texttt{true} \implies \sigma : x; \\ & (\xi : x) = \texttt{false} \implies \tau : x; \\ & \bot\end{aligned}$$

A.4.5 Constant

$$\begin{aligned}\%x : y \;\equiv\; & \\ & y = \bot \implies \bot\,;\; x,\; \forall x \in \Omega\end{aligned}$$

This function returns its object parameter as its result.

A.4.6 Right Insert

$$\begin{aligned}!\sigma : x \;\equiv\; & \\ & x = \langle\rangle \implies e_f : x; \\ & x = \langle x_1 \rangle \implies x_1; \\ & x = \langle x_1, \ldots, x_k \rangle \;\wedge\; k \geq 2 \implies \sigma : \langle x_1, !\sigma : \langle x_2, \ldots, x_k \rangle\rangle; \\ & \bot\end{aligned}$$

For example,

$$\begin{aligned} !+ : \langle 4,5,6 \rangle &= 15 \\ !+ : \langle\rangle &= 0 \\ !* : \langle\rangle &= 1 \end{aligned}$$

Currently, identity functions are defined for + (0), − (0), * (1), /(1), `andg`(1), `org`(0), `xorg`(0). All other functions default to bottom($\bot$).

A.4.7 Left Insert

$$
\begin{aligned}
\mathbf{linss}\sigma : x \;\equiv\; & \\
& x = \langle\rangle \implies e_f : x; \\
& x = \langle x_1 \rangle \implies x_1; \\
& x = \langle x_1, \ldots, x_k \rangle \;\wedge\; k \geq 2 \\
& \implies \sigma : \langle \mathbf{linss}\sigma : \langle x_1, \ldots, x_{k-1} \rangle, x_k \rangle; \\
& \bot
\end{aligned}
$$

A.4.8 Associative Insert

If the function being inserted is associative, the *associative insert* form can be used. This form groups elements in pairs and performs a bottom-up traversal of the insert tree.

$$
\begin{aligned}
|\,\sigma : x \;\equiv\; & \\
& x = \langle\rangle \implies e_f : x; \\
& x = \langle x_1 \rangle \implies x_1; \\
& x = \langle x_1, \ldots, x_k \rangle \;\wedge\; k \geq 2 \\
& \implies |\,\sigma \circ \&\sigma \circ \mathtt{pair} : x \\
& \bot
\end{aligned}
$$

A.4.9 Tree Insert

Alternately, the *tree insert* form could be used for inserting an associative function. The tree insert splits the tree into two halves and recursively descends each sub-tree.

$$
\begin{aligned}
\clubsuit\sigma : x \;\equiv\; & \\
& x = \langle\rangle \implies e_f : x; \\
& x = \langle x_1 \rangle \implies x_1; \\
& x = \langle x_1, \ldots, x_k \rangle \;\wedge\; k \geq 2 \\
& \implies \sigma : \langle \clubsuit\sigma : \langle x_1, \ldots, x_{\lceil n/2 \rceil} \rangle, \clubsuit\sigma : \langle x_{\lceil n/2 \rceil + 1}, \ldots, x_k \rangle \rangle; \\
& \bot
\end{aligned}
$$

A.4.10 Right Seq

$$\begin{aligned}\mathbf{seq}\,\sigma : x \;\equiv\; & \\ & x = \langle\rangle \implies e_f : x; \\ & x = \langle x_1\rangle \implies \langle x_1\rangle; \\ & x = \langle x_1, x_2\rangle \implies \sigma : \langle x_1, x_2\rangle; \\ & x = \langle x_1, \ldots, x_k\rangle \;\wedge\; k > 2 \implies \langle y_1, \ldots, y_k\rangle; \\ & \bot\end{aligned}$$

where

$$\begin{aligned}\langle z, y_3, \ldots, y_k\rangle &= \mathbf{seq}\,\sigma : \langle x_2, \ldots, x_k\rangle \\ \langle y_1, y_2\rangle &= \sigma : \langle x_1, z_2\rangle\end{aligned}$$

A.4.11 Left Seq

$$\begin{aligned}\mathbf{seqL}\sigma : x \;\equiv\; & \\ & x = \langle\rangle \implies e_f : x; \\ & x = \langle x_1\rangle \implies \langle x_1\rangle; \\ & x = \langle x_1, x_2\rangle \implies \sigma : \langle x_1, x_2\rangle; \\ & x = \langle x_1, \ldots, x_k\rangle \;\wedge\; k > 2 \implies \langle y_1, \ldots, y_k\rangle; \\ & \bot\end{aligned}$$

where

$$\begin{aligned}\langle y_1, \ldots, y_{k-2}, z_{k-1}\rangle &= \mathbf{seqL}\sigma : \langle x_1, \ldots, x_{k-1}\rangle \\ \langle y_{k-1}, y_k\rangle &= \sigma : \langle z_{k-1}, x_2\rangle\end{aligned}$$

A.4.12 Map

$$\begin{aligned}\{\sigma_1, \ldots, \sigma_k\} : x \;=\; & \\ & x = \langle x_1, \ldots, x_k\rangle \implies \langle \sigma_1 : x_1, \ldots, \sigma_k : x_k\rangle \\ & \bot\end{aligned}$$

A.5 Time Domain Primitives

The functions SOPI, POSI, $\mathcal{D}^{-1}$, are all functions which are semantically equivalent to the $\nu\mathcal{FP}$ function `id`, but correspond to specific constructs in the layout. There are time-domain equivalents to the *Apply-to-All*, *Right Insert*, *Left Insert*, *Right Seq* and *Left Seq* combining forms. These are denoted respectively by, **tapall, tlins, tseq, tseqL**. They are equivalent in terms of meaning to their space-domain counterparts, but they are laid out differently.

A.6 User Defined Functions

An FP definition is entered as follows:

```
defun functionName ( arguments )
    ...enddef
```

where *functionName* is an ascii string consisting of letters, numbers and the underline symbol. The *arguments*, which must be simple names may be omitted. In that case, the parentheses around the *arguments* is also omitted. For example, the functions

```
defun factorial ( n )
      if zero?  ∘n then %1
                   else *∘[ n, factorial∘- ∘[ n , %1]])
enddef

defun zero?
      = ∘[id, %0 ]
enddef
```

form a recursive definition of the factorial function.

Appendix B

Clarifications on Some Aspects of GMB Semantics

B.1 Introduction

Vernon, in [84], formally defines the Graph Model of Behavior (GMB). However, there are a few aspects of the semantics that would benefit from further clarifications. This appendix deals with some of these aspects.

It is assumed that the reader is familiar with the GMB and has at least an overview knowledge of the issues discussed in [84]. As far as possible, this document uses the same symbols as in [84].

B.2 Semantics of Control Node Enabling

Normally, the enabling conditions for a control node are specified by an input logic expression. Formally, it is better to use a marking function that represents a precondition on the firing of the node. This is discussed in section 2.2 of [84], where the marking function L- is defined. Formally,

$$\text{L-} : \text{N} \rightarrow 2^{2^{\text{A}}}$$

where $\text{N} = \{n\}$ is the set of control nodes
$\text{A} = \{a\}$ is the set of control arcs

Note that this is a simplification of the specification given by Vernon in that it is assumed that only one token at a time will ever be removed from an arc. Section B.6 shows how this

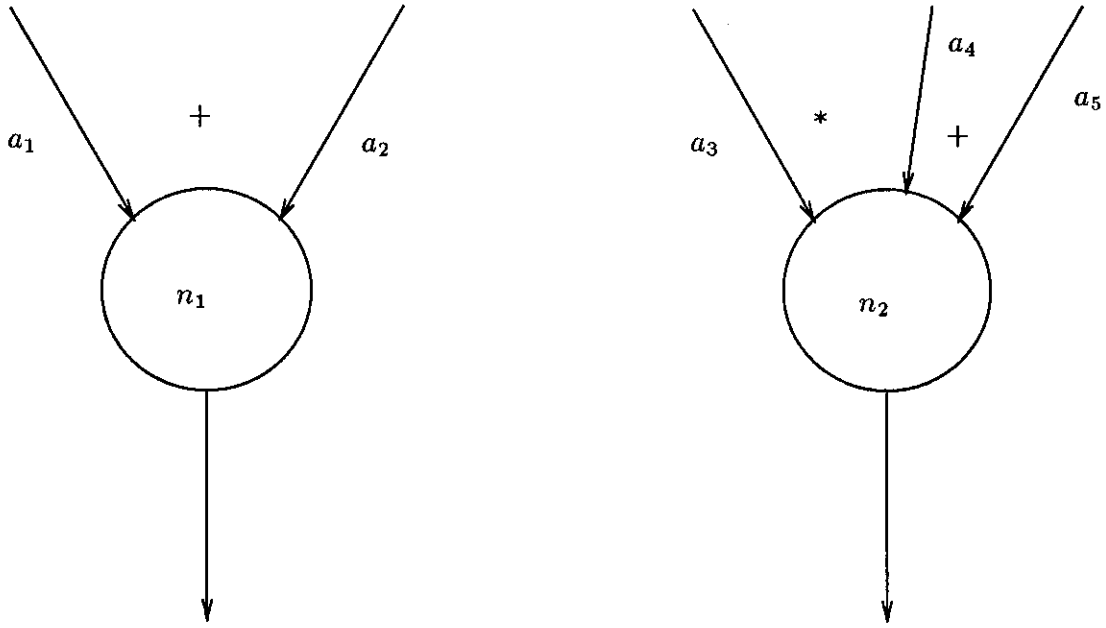


Figure B.1: Sample Control Graph

formalism is extended to handle the case when multiple tokens on an arc are required for enabling a node. Also note that [84] describes the range of L- to be 2^A even though examples indicate that 2^{2^A} is what was intended.

The above definition says that given $n \in$ N, L-(n) returns a set of sets. Each element of the set is a set of control arcs that must have at least one token on each of the constituent control arcs for the node n to be enabled for firing. For example, in figure B.1,

$$\begin{aligned} \text{L-}(n_1) &= \{\{a_1\}, \{a_2\}\} \\ \text{L-}(n_2) &= \{\{a_3, a_4\}, \{a_3, a_5\}\} \end{aligned}$$

Node n_1 is enabled if there is a token on arc a_1 or on arc a_2. Node n_2 is enabled if there are tokens on arcs a_3 and a_4 or on arcs a_3 and a_5.

Section 3.6.1 of [84] mentions that the priority operator defines a partial order on the range set of L-. This is expanded on in section 3.8 of [84]. To make the partial order explicit, L- is redefined to be

$$\text{L-} : \text{N} \rightarrow L' \times \leq$$

where

$$\begin{array}{rcccl} L' & = & \mathcal{D}(\mathcal{R}(\text{L-})) & = & \text{domain of L-'s range} \quad = \quad 2^{2^{\mathbf{A}}} \\ \leq & = & \mathcal{R}(\mathcal{R}(\text{L-})) & = & \text{range of L-'s range} \quad = \quad 2^{(2^{\mathbf{A}} \times 2^{\mathbf{A}})} \end{array}$$

There is no change to the algorithm of the token machine, presented in section 2.3 of [84], to determine whether a control node can be fired or not, except that instead of L-, the first element of L-, L', must be used.

B.3 Semantics of Control Token Removal

Step 2.c.i in section 2.3 of currently states:

> If L-(n_p) is satisfied, randomly select a marking in L-(n_p) which is satisfied, and remove the specified number of tokens for each arc in the set.

This is sufficient in the absence of the priority operator. After the introduction of the priority operator, section 3.6.1 of [84] mentions that the token removal algorithm presented in section 2.3 of [84] should be modified so that

> ...input tokens are removed in priority order, rather than non-deterministic order...

If each element, L'_{n_p}, of L' were totally ordered with a relation $\leq_{n_p} \in \leq$, the algorithm for selecting the appropriate element of L'_{n_p} would be obvious. However, because $\leq_{n_p}$ is a partial order, the algorithm is not obvious and this section presents an algorithm for determining the order in which the elements of L'_{n_p} will be checked for determining which tokens are to be removed. This algorithm should be viewed as a replacement for step 2.c.i in section 2.3 of [84].

- n_p is the control node that is being investigated.
- Let L-(n_p) = $\langle L'_{n_p}, \leq_{n_p} \rangle$.
- X and Y are temporary variables.

1. Set $X = \langle X', X'' \rangle \leftarrow$ L-(n_p). That is, $X' \leftarrow L'_{n_p}$ and $X'' \leftarrow \leq_{n_p}$.
2. Repeat the following steps until X' is $\emptyset$.

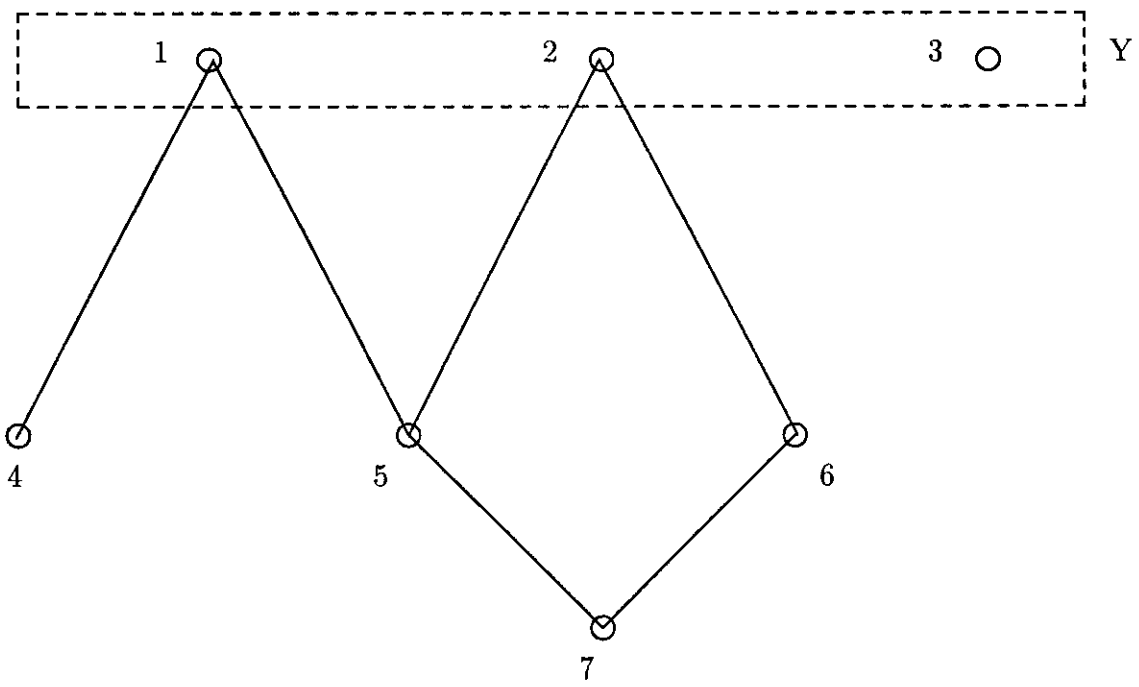


Figure B.2: Initial state of X

(a) Set $Y \leftarrow \aleph_+(L'_{n_p}, \leq_{n_p})$[1].

(b) If at least one element of Y is satisfied by the current token distribution, non-deterministically choose one of them and exit the algorithm. If none of the elements of Y is satisfied by the current token distribution then set $X \leftarrow \langle X' - Y\,,\, X'' - (X''/Y)\rangle$[2]. Repeat.

3. Since X' is $\emptyset$, n_p cannot be currently fired.

For example, assume that the poset[3] of markings for a particular control node is shown by the Hasse diagram in figure B.2. The nodes of the graph represent individual markings and the arcs represent the ordering. If a node, connected by an arc to another node, is above that node, it means that the first node (marking) has priority over the second marking. Initially,

[1] $\aleph_+(P, \leq)$ is the set of all the *maximal members* of P with respect to the partial order $\leq$. An element $y \in P$ is a maximal member of P with respect to a partial order $\leq$ if for no $x \in P$ is $y < x$.

[2] R/S is the restriction of a relation R to the set S and can be defined as $R/S = R \cap (S \times \mathcal{D}(R))$ where $\mathcal{D}(R)$ is the domain of the relation R, and $S \subseteq \mathcal{D}(R)$.

[3] A poset is a set with a partial order relation defined on it.

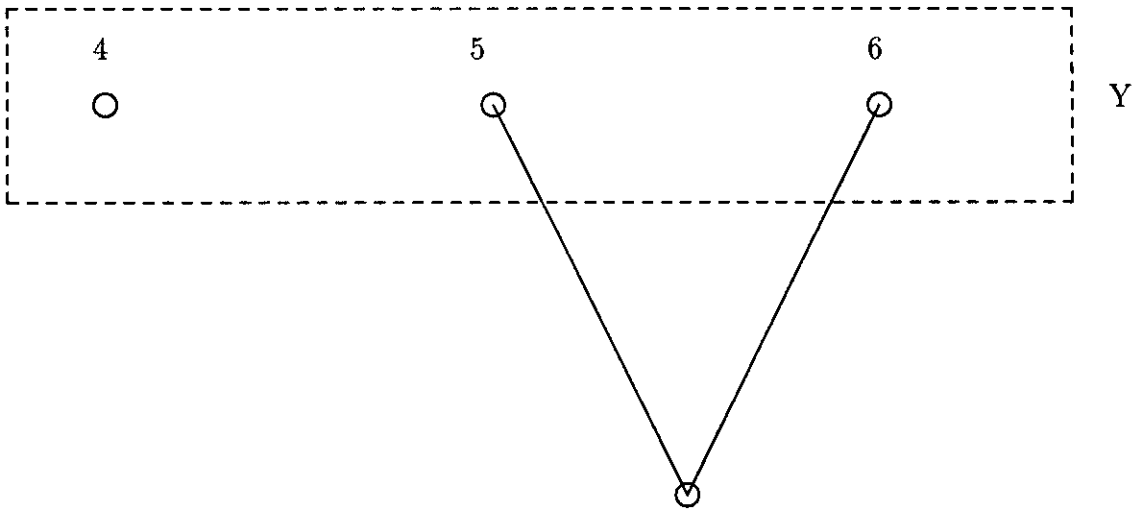


Figure B.3: Next state of X

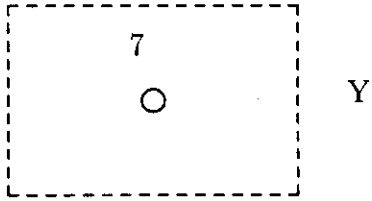


Figure B.4: Final state of X

X is the whole poset, and Y is the set of nodes labeled 1,2, and 3. The algorithm first checks whether the markings corresponding to 1,2, or 3 are satisfied. If so, it non-deterministically chooses any one and is done. If none are satisfied, the nodes in Y are removed from X along with the arcs that they are connected to. The results are shown in figure B.3. Now Y consists of the nodes marked 4,5, and 6. Again, if any of the markings in Y are satisfied, one is chosen non-deterministically. If none of the markings are satisfied, then the nodes in Y are removed resulting in figure B.4. At this stage, there is only one marking, 7, left, and if it is not satisfied, then X is set to $\langle \emptyset, \emptyset \rangle$ and the algorithm stops.

This algorithm insures that markings with higher priorities are examined before markings with lower priorities. In addition, the maximal markings are examined in non-deterministic order.

B.4 Translation of Input Logic Expressions to Markings

Though L-, the input marking poset function, is useful for describing the input logic enabling and token removal semantics, it is not very convenient for practical use by users. Users of the GMB use *input logic expressions* to specify L- indirectly. However, the mapping between an input logic expression and L- has never before been explicated in the GMB literature leading to confusion as to the precise meaning of a particular input logic expression. The source code of the implemented GMB simulator has been the final arbiter of the semantics of a particular logic expression. This section hopes to clarify the situation by providing a formal translation between the input logic expression of a control node and the poset returned by L- for that node.

Another benefit of this formalism is to provide a mechanism for proving equivalences between two logic expressions. This can be used to formulate an algebra of logic expressions. For example, it can be shown that

$$a * (a + b) \equiv a + b$$

or

$$(a > b) + (b > a) \equiv a + b$$

As can be seen from the above examples, this is not the same as boolean algebra. Using boolean algebra would have resulted in a and $a \oplus b$ as the right hand sides of the respective equations mentioned above.

An input logic expression can be described by the following BNF:

$$\begin{aligned} expr = \quad & expr + expr \\ | \quad & expr * expr \\ | \quad & expr > expr \\ | \quad & simpleExpr \end{aligned}$$

The syntax-directed translation of an input logic expression to the corresponding poset is specified by providing the translation for each of the BNF rules. Consider, first, the BNF rule

$$expr = simpleExpr \tag{B.1}$$

where $simpleExpr$ is the name of a control arc. The corresponding poset is

$$\langle \{\{simpleExpr\}\}, \{\langle \{simpleExpr\}, \{simpleExpr\}\rangle\}\rangle \tag{B.2}$$

In other words, the set consists of one element, which is a set just containing the control arc. The relation only contains the reflexive relation from *simpleExpr* to itself.

Next, consider the BNF rule

$$expr = expr_1 + expr_2$$

Let the poset corresponding to $expr_1$ be

$$X_1 = \langle P_1, \leq_1 \rangle$$

and the poset corresponding to $expr_2$ be

$$X_2 = \langle P_2, \leq_2 \rangle$$

Then the poset corresponding to $expr_1 + expr_2$ is specified by

$$\langle P_1 \cup P_2 \, , \, \leq_1 \overset{\leftrightarrow}{\cup} \leq_2 \rangle$$

where

$\overset{\leftrightarrow}{\cup}$ ... is the *poset union* (see section B.7) operator

In other words, having the "or" operator in an input logic expression results in taking the union of each of the elements of the sub-expressions. However, in general, the union of two posets will not be a poset. Hence, after forming the union and its transitive closure, all edges of the resultant graph taking part in a cycle are removed. This is accomplished by taking the set difference with the set corresponding to the symmetric sub-relation. It can be shown (section B.7) that the resulting relation (called the *punion*, or *poset union*) is a partial order.

For the priority operator, the poset corresponding to the expression

$$expr_1 < expr_2$$

can be constructed from its constituents as

$$\langle P_1 \cup P_2 \, , \, \leq_1 \overset{\leftrightarrow}{\cup} \leq_2 \overset{\leftrightarrow}{\cup} X \rangle$$

where

$$X = \{\langle x, y \rangle \mid x \in \aleph_-(P_2, \leq_2) \wedge y \in \aleph_+(P_1, \leq_1)\}$$

$\aleph_-(P, \leq)$ is the set of all minimal members[4] of P relative to the partial order $\leq$. This operation can be graphically illustrated by the use of Hasse diagrams as shown in figures B.5, B.6 and B.7.

As can be seen, combining two sub-expressions with the priority operator consists of combining

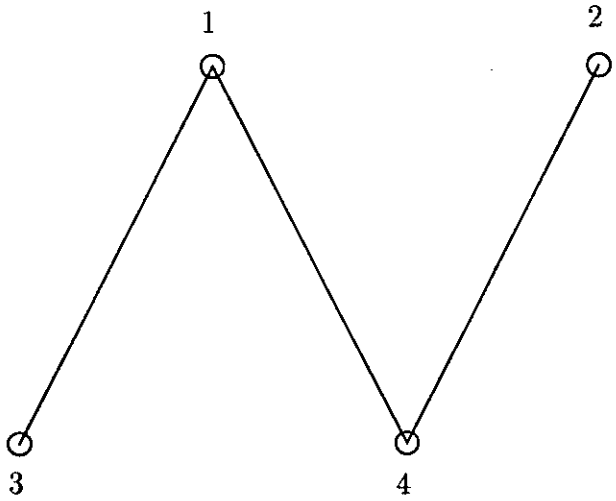


Figure B.5: Hasse diagram of X_1

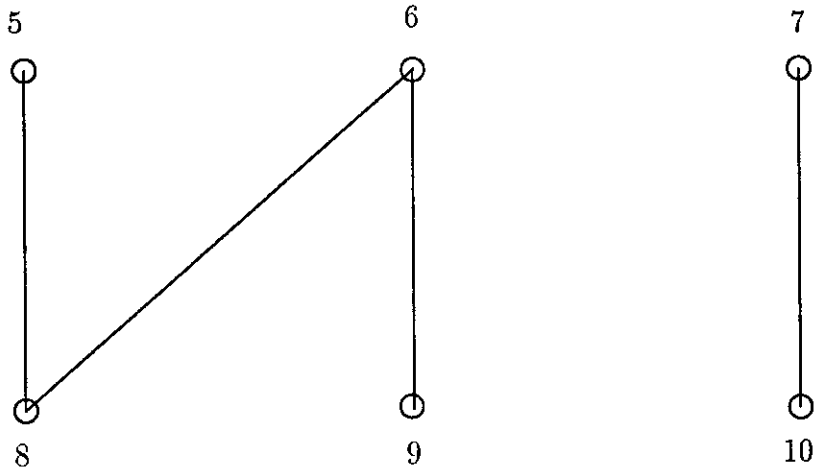


Figure B.6: Hasse diagram of X_2

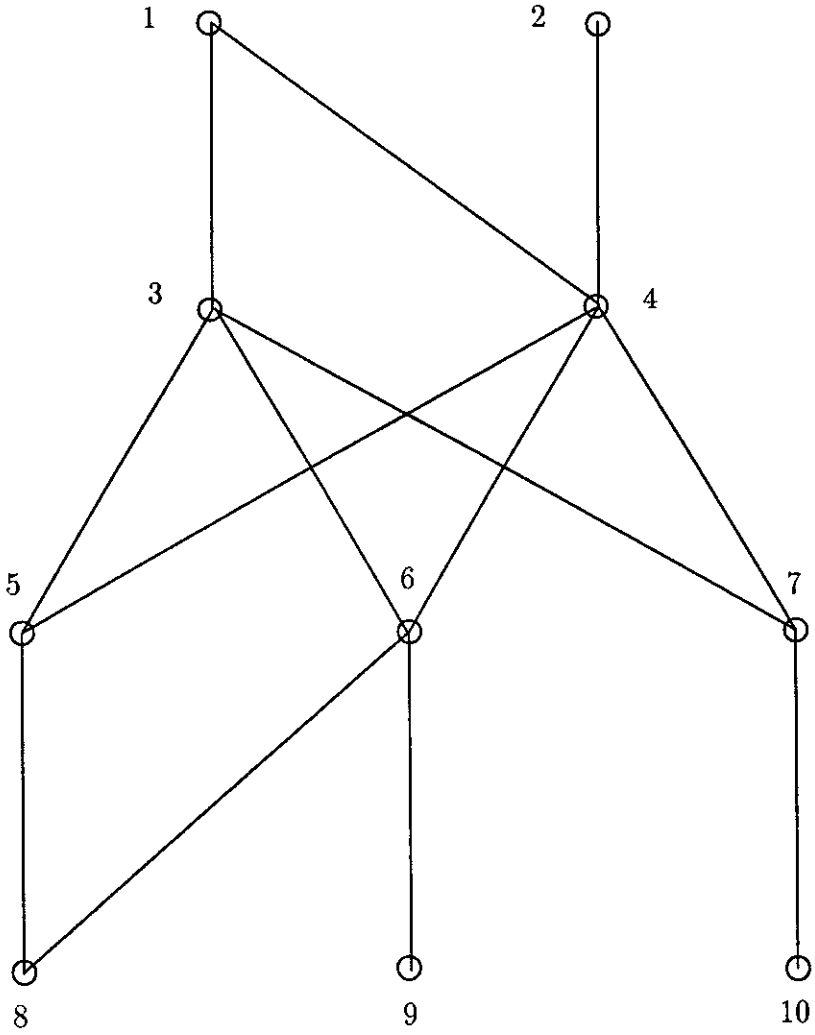


Figure B.7: Hasse diagram of $X_1 < X_2$

their respective posets in such a way that all members of the higher priority poset are ordered higher than the elements of the lower priority poset. Once again, the symmetric sub-relation has to be removed in order to make the resulting relation a partial order.

The poset corresponding to the expression

$$expr_1 * expr_2$$

can be given by

$$\langle P_1 \uplus P_2 \,,\, X^* \rangle$$

where

$$S_1 \uplus S_2 = \{p \cup q \mid p \in S_1 \wedge q \in S_2\}$$

and

$$\begin{aligned} R^* &= R^0 \cup R^+ \ldots \text{ is the } \textit{reflexive transitive closure} \text{ of } R \\ S &= \leq_1 \cup \leq_2 \\ X &= \{\langle x, y \rangle \mid x, y \in S_1 \uplus S_2 \wedge \\ &\quad (2^x \times 2^y) \cap (S - S^0) \neq \emptyset \wedge \\ &\quad (2^y \times 2^x) \cap (S - S^0) = \emptyset\} \end{aligned}$$

The operator $\uplus$ is very similar to the cartesian product of two sets. The difference is that the cartesian product operator takes elements of two sets and makes ordered pairs out of them, whereas the $\uplus$ operator takes the union of the two elements. The idea is that two elements x and y in $S_1 \uplus S_2$ will be related if there is a $p \subseteq x$ and a $q \subseteq y$ such that $\langle p, q \rangle$ is in $\leq_1 \cup \leq_2$. At the same time, there must not be any $p \subseteq x$ and $q \subseteq y$ such that $\langle q, p \rangle$ is in $\leq_1 \cup \leq_2$. Note that the relation X is irreflexive and anti-symmetric, and hence

$$\langle P_1 \uplus P_2, X^* \rangle$$

is a poset. As an example, consider the input logic expression

$$(a > b) * (c > d)$$

The poset corresponding to the sub-expression $a > b$ is shown in figure B.8. The poset corresponding to the sub-expression $c > d$ is shown in figure B.9. And, the poset corresponding to the final logic expression $(a > b) * (c > d)$ is shown in figure B.10.

[4] An element $y \in P$ is called a *minimal member* of P relative to a partial order $\leq$ if for no $x \in P$ is $x < y$.

Figure B.8: Poset corresponding to $a > b$

Figure B.9: Poset corresponding to $c > d$

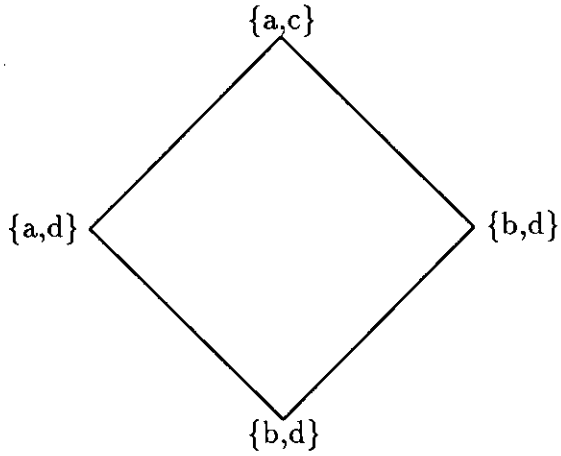


Figure B.10: Poset corresponding to $(a > b) * (c > d)$

B.5 Semantics of Controlled-Read and Priority-Read Data Arcs

Section 3.3 of [84] introduces the "Controlled-Read" data arc and section 3.7 introduces the "Priority-Read" data arc (see section 5.1.2 on page 50 for an explanation). However, some aspects of the semantics in those sections are unspecified. This makes it difficult to generalize the concept to the case which deal with complex input logic expressions. This section attempts to more precisely define the functions, domains, and ranges used in [84].

Following [84], let $DataSet$ be a function that maps a data arc to a set of datasets. Formally,

$$DataSet : \{d\} \rightarrow 2^{Q \cup H}$$

where $d \in D$ is the set of data arcs
Q is the set of datasets
H is the set of sockets

Similarly, let $ioControl$ be a function that maps a data arc to a set of control arcs. Formally,

$$ioControl : \{d\} \rightarrow 2^{A}$$

In [84], the functions $DataSet$ and $ioControl$ return sets that are completely ordered. There is an implicit one-to-one mapping between corresponding elements of the sets returned by these functions. That is, the first element of the set returned by $ioControl$ is implicitly mapped to the first element of the set returned by $DataSet$ for the same data arc argument. A different approach is taken here. Instead of requiring that these functions return completely ordered sets with an implicit map between the elements of the sets, a new function, $\mathcal{N}$, is introduced. $\mathcal{N}$ is a function that, given a data arc, returns a one-to-one correspondence between datasets connected to that data arc and control arcs. Formally,

$$\mathcal{N} : D \rightarrow ((Q \cup H) \rightarrow A)$$

Since $\mathcal{N}(d)$, $d \in D$, is a one-to-one correspondence, its inverse also exists and can be used to associate a dataset, connected to the data arc d, with a given control arc.

For a data arc, d, that is either Controlled-Read or Priority-Read, the the number of source datasets must equal the number of ioControl control arcs. This is because the $\mathcal{N}(d)$ is a one-to-one correspondence. Thus,

$$|ioControl(d)| = |DataSet(d)|$$

Following [84], M_- is a function that maps a data processor to a set of control nodes that can activate it. Consider a particular Controlled-Read or Priority-Read data arc, d and a control node, n, associated with the data processor associated with d. It is required that each mapping in the input logic expression of n shall contain only one of the control arcs contained in $ioControl(d)$. Without this restriction it would be impossible for the token machine to decide which dataset value to provide the data processor when it performs a read operation on the data arc d. Another restriction is that data processors that have Controlled-Read or Priority-Read data arcs connected to them can have only one control node associated with themselves. In other words, $ProcessorSet(d)$ returns a singleton set.

$$\begin{aligned}
d &\in D \\
p &\in ProcessorSet(d) \\
|ProcessorSet(d)| &= 1 \\
n &\in M_-(p) \\
|M_-(p)| &= 1 \\
\text{L-}(n) &= \langle L'_n, \leq_n \rangle \\
S_1, S_2 \in DataSet(d) &\implies \nexists x \in L'_n \text{ such that} \\
&\quad (\mathcal{N}(d))(S_1) \in x \wedge (\mathcal{N}(d))(S_2) \in x
\end{aligned}$$

B.6 Incorporating Multiple Token Removal

So far, it has been assumed that if a control arc is mentioned in the input logic mapping there must be at least one control token on that arc. Additionally, if that mapping is selected for token removal, then only one token will be removed from that arc. As promised in section B.2, this section extends the formalism described above to the case where multiple tokens may be removed on firing of a control node and multiple tokens may be required for the enabling of a control node.

As far as the mapping function, L-, is concerned, the change is very trivial. Instead of A being the set of control arcs, it should now be treated as a set of ordered pairs. The first element of the pair is from the set of control arcs as usual, and the second element of the pair is from the set of natural numbers, **N**. Thus, L- can be redefined as:

$$\text{L-} : \text{N} \rightarrow L' \times \leq$$

where

$$L' = 2^{2^{\mathbf{A} \times \mathbf{N}}}$$

$$\leq \; = \; 2^{2^{\mathbf{A}\times\mathbf{N}}\times 2^{\mathbf{A}\times\mathbf{N}}}$$

The following caveat needs to be added to this formulation. An element of the form

$$\langle a, 0\rangle \; ; \; a \in \mathrm{A}$$

does not mean that there should be 0 or more tokens on the arc a, but instead that the arc a is not considered in this mapping. An alternative solution could be to use the set $\mathbf{N} - \{0\}$ (or $\mathcal{I}^+$, the set of positive integers) instead of $\mathbf{N}$.

Let $\langle a, n\rangle$ mean that n tokens are to be removed from control arc a. Then, the input logic expression

$$\langle a, n\rangle + \langle a, m\rangle$$

means that the control node will fire if there are at least n tokens on arc a, and when it fires, either n tokens will be removed, or if the number of tokens on the arc is currently greater than or equal to m, then, non-deterministically, either n or m tokens will be removed.

The input logic expression

$$\langle a, n\rangle > \langle a, m\rangle$$

means the same as for the "+" case if $n \geq m$, except that if the number of tokens on the arc is greater than or equal to m, then exactly n tokens will be removed.

The input logic expression

$$\langle a, n\rangle < \langle a, m\rangle$$

means the same as for the "+" case if $n \geq m$, except that if the number of tokens on the arc is greater than or equal to m, then exactly m tokens will be removed.

The input logic expression

$$\langle a, n\rangle * \langle a, m\rangle$$

and $n \geq m$, means that the node will fire if there are at least m tokens on the arc a. When it fires, exactly m tokens will be removed from the arc.

If the substitution $m = n = 1$ is made in the above expressions, the semantics described above match the current GMB semantics. In particular, $a * a$ means that exactly one (not two) token will be removed from the arc a.

With this understanding, the previous procedures for translating between input logic expressions and the partially ordered marking set can be reconsidered. The procedures for the simple expression, the "or" and the "priority" case remain the same. However, the procedure for the "and" needs to be modified as follows. Replace all occurrences of $\uplus$ with $\amalg$, where $\amalg$ is defined as:

$$S_1 \amalg S_2 = \{p \sqcup q \mid p \in S_1 \wedge q \in S_2\}$$

This is the same as $\uplus$ except that $\cup$ is replaced by $\sqcup$, and $\sqcup$ is defined as:

$$S_1 \sqcup S_2 = \{\langle p, q\rangle \mid \langle p, n\rangle \in S_1 \wedge \langle p, m\rangle \in S_2 \wedge q = max(m, n)\}$$

under the assumption that

$$\langle p, 0\rangle \in S \iff \langle p, x\rangle \notin S, \forall x \neq 0$$

Note that $\sqcup$ is very similar to the union operator on sets except that it works on sets containing ordered pairs. Additionally, for elements in both sets having the same first element, the *maximum* of the second elements is the second element of the result.

B.7 Punion Theorem

In general, the union of two partial orders will not be a partial order. This is because even though the two constituent relations are anti-symmetric, their union may be symmetric. For example, consider the partial order

$$\{\langle a, a\rangle, \langle b, b\rangle, \langle a, b\rangle\}$$

and the partial order

$$\{\langle a, a\rangle, \langle b, b\rangle, \langle b, a\rangle\}$$

Their union, is the relation

$$\{\langle a, a\rangle, \langle b, b\rangle, \langle a, b\rangle, \langle b, a\rangle\}$$

which is symmetric and hence not a partial order.

However, the *punion* of two partial orders is a partial order. A punion of two partial orders $\leq_1$ and $\leq_2$, denoted by $\leq_1 \overleftrightarrow{\cup} \leq_2$ is defined as:

$$\begin{aligned}
\leq_1 \overleftrightarrow{\cup} \leq_2 &= S^+ - \overleftrightarrow{(S^+)} \\
S &= \leq_1 \cup \leq_2 \\
\overleftrightarrow{R} &= \{\langle x, y\rangle \mid \langle x, y\rangle \in R \wedge \langle y, x\rangle \in R \wedge x \neq y\} \\
&\quad \ldots \text{ is the } \textit{symmetric sub-relation} \text{ on } R \\
R^+ &= R \cup R^2 \cup R^3 \ldots \text{ is the } \textit{transitive closure} \text{ of } R
\end{aligned}$$

A few lemmas need to be proven before proving that the punion of two partial orders is a partial order.

Lemma B.1 The punion of two partial orders is reflexive. Proof:

1. $\leq_1$ is reflexive. *defn of P.O.*
2. $\leq_2$ is reflexive. *defn of P.O.*
3. S is reflexive. *union maintains reflexivity*
4. S^+ is reflexive. *transitive closure maintains reflexivity*
5. $\overleftrightarrow{(S^+)}$ is not reflexive. *defn of* $\overleftrightarrow{\ }$
6. $S^+ - \overleftrightarrow{(S^+)}$ is reflexive. *defn of* $-$
7. $\leq_1 \overleftrightarrow{\cup} \leq_2$ is reflexive. *defn of* $\overleftrightarrow{\cup}$

Lemma B.2 The punion of two partial orders is anti-symmetric. Proof:

1. $\overleftrightarrow{(S^+)}$ is symmetric. *defn of* $\overleftrightarrow{\ }$
2. $S^+ - \overleftrightarrow{(S^+)}$ is anti-symmetric. *defn of* $-$
3. $\leq_1 \overleftrightarrow{\cup} \leq_2$ is reflexive. *defn of* $\overleftrightarrow{\cup}$

Lemma B.3 The punion of two partial orders is transitive. Proof:

1. Assume $\langle x, y\rangle \in S^+ - \overleftrightarrow{(S^+)}$
2. Assume $\langle y, z\rangle \in S^+ - \overleftrightarrow{(S^+)}$
3. $\langle x, z\rangle \in S^+$ *transitivity of* S^+
4. We need to show that $\langle x, z\rangle \notin \overleftrightarrow{(S^+)}$
5. Assume $\langle x, z\rangle \in \overleftrightarrow{(S^+)}$
6. $\langle z, x\rangle \in \overleftrightarrow{(S^+)}$ *since* $\overleftrightarrow{(S^+)}$ *is symmetric.*
7. $\langle z, x\rangle \in S^+$ *since* $\overleftrightarrow{(S^+)} \subset S^+$
8. $\langle x, y\rangle \in S^+$ *by step 1.*
9. $\langle z, y\rangle \in S^+$ *by steps 7 and 8*
10. $\langle z, y\rangle \wedge \langle y.z\rangle \in \overleftrightarrow{(S^+)}$ *by steps 2 and 9.*
11. $\langle y, z\rangle \notin S^+ - \overleftrightarrow{(S^+)}$ *by step 10.*
12. This is contradictory to step 2.
13. $\langle x, z\rangle \notin \overleftrightarrow{(S^+)}$
14. $\langle x, z\rangle \in S^+ - \overleftrightarrow{(S^+)}$ *by steps 3 and 13.*
15. $S^+ - \overleftrightarrow{(S^+)}$ is transitive *by steps 1, 2 and 14.*

The three lemmas taken together show that the punion of two partial orders is reflexive, anti-symmetric and transitive and hence a partial order.