Synthesis of Self-timed Circuits by Program Transformation*

Steven M. Burns and Alain J. Martin

Computer Science Department
California Institute of Technology
Pasadena, CA 91125 USA

Self-timed circuits can be synthesized from concurrent programs in two logically separate phases. First, through a series of program transformations, the source program is decomposed into an equivalent program constructed entirely from instances of basic processes. These basic processes correspond to the syntactic constructs of the source language. The remainder of the synthesis procedure consists of compiling each of the basic processes into a self-timed circuit using techniques described in earlier papers. These compilations need to be done only once. This paper describes in detail the program transformations used in an automated synthesis procedure developed at Caltech. The transformations used are applications of *process decomposition*, a simple technique that is easy to verify. The circuits synthesized by these program transformations are correct by construction; thus, this technique provides a simple method for constructing provably correct circuits from a high-level specification.

We propose a method for developing VLSI circuits from an abstract specification. The programmer designs a concurrent program that meets this specification, and then an automatic mechanism transforms the program into a circuit. The programmer's proof obligation is limited to verifying that the concurrent program is an implementation of the specification. The program-to-circuit transformation is verified separately.

Program transformations within the source language provide a powerful tool for deriving implementations of programs. By working at an abstract level in a language with a well-defined semantics, transformations, which are complex if performed at the circuit level, are reduced to trivial syntactic manipulations. Such transformations are both easy to perform and easy to verify. They form the core of an automatic compiler for synthesizing self-timed circuits.

We have constructed a set of program transformation rules that, when applied to any program in the source language, transform it into an equivalent program of a very simple form. This form is composed only of simple basic processes that have already been compiled into circuits. In this paper, we describe in detail this set of program transformations. In addition, we show the compiled circuits for each of the basic processes and the resulting syntax-directed translation rules. We also introduce and compare various schemes for guard evaluation and then apply these schemes to a simple example.

*appears in *The Fusion of Hardware Design and Verification*, G.J. Milne, ed., North-Holland (1988)

```
1.  ⟨process⟩     ::=  ( ⟨process⟩ { || ⟨process⟩ } ) { ⟨channel⟩ }
2.                     | { ⟨port⟩ } { ⟨var⟩ } ⟨sequence⟩
3.  ⟨channel⟩     ::=  channel ( ⟨NAME⟩ , ⟨NAME⟩ )
4.  ⟨port⟩        ::=  ( passive | active ) ⟨NAME⟩ ( ⟨INT⟩ , ⟨INT⟩ )
5.  ⟨var⟩         ::=  boolean ⟨NAME⟩ = ( true | false )
6.  ⟨sequence⟩    ::=  ⟨statement⟩ [ ; ⟨sequence⟩ ]
7.  ⟨statement⟩   ::=  skip
8.                     | ⟨NAME⟩ ( up | down )
9.                     | ⟨NAME⟩ ( ⟨INT⟩ ) : [ ⟨responses⟩ ]
10.                    | ( [ | *[ ) ⟨gcs⟩ ]
11. ⟨responses⟩   ::=  ⟨response⟩ { | ⟨response⟩ }
12. ⟨response⟩    ::=  ⟨INT⟩ --> ⟨sequence⟩
13. ⟨gcs⟩         ::=  ⟨gc⟩ { | ⟨gc⟩ }
14. ⟨gc⟩          ::=  ⟨expr⟩ --> ⟨sequence⟩
15. ⟨expr⟩        ::=  ⟨conjunct⟩ [ or ⟨expr⟩ ]
16. ⟨conjunct⟩    ::=  ⟨primary⟩ [ and ⟨conjunct⟩ ]
17. ⟨primary⟩     ::=  not ⟨primary⟩
18.                    | ( ⟨expr⟩ )
19.                    | ⟨NAME⟩
20.                    | probe ⟨NAME⟩
21.                    | (true | false)
```

Figure 1: Backus-Naur Form (BNF) for Source Language

1 Source Language

The source language is based on CSP[3], with the addition of the *probe*[6] and a new communication construct. A complete description of the language syntax is given in Figure 1. We shall refer to this figure when deriving the individual transformation rules.

A program in this language consists of a set of sequential processes with interconnecting channels. Associated with each sequential process is a set of ports, a set of private variables, and a list of statements to be executed sequentially. Ports that do not connect to another process connect to the environment.

Only boolean variables are allowed. Variables are changed by assignment to true (`x up`) or to false (`x down`). The selection (`[`⟨gcs⟩`]`) and repetition (`*[`⟨gcs⟩`]`) constructs are based on guarded commands. We use `*[`⟨sequence⟩`]` as an abbreviation for `*[true-->`⟨sequence⟩`]`.

Synchronization between two processes is accomplished by zero-slack communication actions across channels denoted by pairs of ports. Of the two ports that make up a channel, one is declared *active* and the other is declared *passive*. The process that owns the *passive* port can determine whether the other process is waiting for a communication on this channel by evaluating a boolean condition called a probe. Probes may be used in arbitrary boolean expressions.

Though concurrently operating processes may not share variables, processes may communicate data by exchanging values from small sets during a synchronization action.

When declaring a port, we specify both the send and receive sets of values, each set being represented by a single integer. For example,

```
passive L(3,2)
```

declares a *passive* port `L` with send set $\{0,1,2\}$ and receive set $\{0,1\}$. The syntactic construct for a communication action allows different sequences of commands to be executed based on the value received during a communication. An execution of the communication action (on the same port, `L`)

```
L(1):[ 0 --> x down | 1 --> x up ] ,
```

sends the value `1` and simultaneously receives either a `1` or a `0`. If a `0` is received, x is set to false; if a `1` is received, x is set to true. We allow two abbreviations in the specification of a communication action: The output value may be omitted if the port has only one send value, and the receive value selection may be omitted if the port has only one receive value.

2 Target Language — Self-timed Circuits

The target of the compilation is a self-timed circuit—a set of circuit variables (nodes) interconnected by a set of operators (gates). These circuits are designed to function correctly regardless of the internal delays of the operators. The required operator types include the combinational elements, *WIRE*, *AND*, and *OR*; and the state-holding elements shown in Figure 2. Each operator is defined in terms of a set of production rules[4, 5]. A production rule is a simple transition rule of the form $G \longmapsto S$, where G is a boolean expression and S is an assignment to true or false. All references to a circuit variable are assumed to have the same value (isochronic forks)[1, 4]. A synchronizer, which cannot be represented in terms of production rules, is included to allow the implementation of programs with negated probes. The synchronizer, as well as the other operators, have been implemented as CMOS standard cells.

Self-timed circuit implementations of concurrent programs are generated by implementing each sequential process as a separate sub-circuit. The sub-circuits are connected (by wire operators) only to implement communication actions. The simultaneity required in the zero-slack communications is implemented using a four-phase handshaking protocol. In order to implement general communication actions (those in which data is transmitted) the usual request/acknowledge pair of wires is replaced by one wire for each send value and one wire for each receive value.

3 Syntax-directed Compilation

An arbitrary program in the source concurrent language is compiled into a target self-timed circuit by a syntax-directed translator, similar to that used in standard program-to-machine-code compilers. Such a translator requires a set of BNF rules describing the

$$(x,y)C(z) \equiv \begin{cases} x \wedge y & \longmapsto z\uparrow \\ \neg x \wedge \neg y & \longmapsto z\downarrow \end{cases}$$

$$(\underline{x},\underline{y})FF(z) \equiv \begin{cases} \bigvee x_i & \longmapsto z\uparrow \\ \bigvee \neg y_i & \longmapsto z\downarrow \end{cases}$$

$$(x,b)SYNC(u,v) \equiv \begin{cases} *[[x \wedge b & \longrightarrow u\uparrow;[\neg x];u\downarrow \\ \;\;|x \wedge \neg b & \longrightarrow v\uparrow;[\neg x];v\downarrow \\ ]] \end{cases}$$

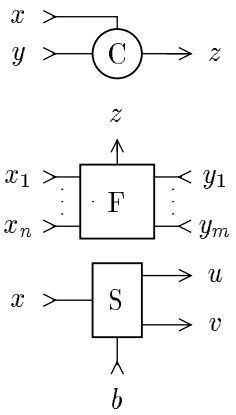

Figure 2: State-holding Operators

syntax of the source language and a set of translation rules describing how to construct objects in the target language. In this application, an object in the target language is a self-timed circuit. The translation rules specify how to generate and connect circuits corresponding to the syntactic constructs. The translation rules are derived in two logically separate phases: program transformation and basic process compilation.

3.1 Process Decomposition

Process decomposition is the most commonly used program transformation. An arbitrary program part β is replaced by a single active communication and a separate process implementing β:

$$\alpha;\beta;\gamma \;\triangleright\; \textbf{active}\; A' \;\; \textbf{passive}\; A\; (\alpha;A';\gamma \parallel *[[\overline{A} \longrightarrow \beta;A]])\, \textbf{channel}\; (A',A)\;.$$

(Read '$\triangleright$' as "is replaced by".) Process decomposition does not introduce concurrency; the active communication A' cannot finish until A and, thus, β complete. The original process and the new process may share variables and ports. These two processes are never active concurrently; thus, exclusive access to each variable and port is ensured. In the following, we do not explicitly declare the ports and channel used in a process decomposition. The two ports of a channel are denoted by the same capital letter. The primed letter represents the active port. We write the above process decomposition as:

$$\alpha;\beta;\gamma \;\triangleright\; \alpha;A';\gamma \parallel (A/\beta)\;.$$

A more general form of process decomposition is used to implement constructs involving guard evaluation. An evaluation construct may be implemented in a separate process, and if this is the case, the multi-valued result of the evaluation is communicated back to the original process by a general communication action. Such a decomposition is of the form:

$$\begin{array}{ll} & [\gamma_0 \longrightarrow \beta_0 | \ldots | \gamma_{n-1} \longrightarrow \beta_{n-1}] \\ \triangleright & \quad \textbf{active}\; G'(1,n) \;\; \textbf{passive}\; G(n,1) \\ & \quad (G' : [0 \longrightarrow \beta_0 | \ldots | n-1 \longrightarrow \beta_{n-1}] \\ & \quad \parallel *[[\overline{G} \longrightarrow [\gamma_0 \longrightarrow G(0) | \ldots | \gamma_{n-1} \longrightarrow G(n-1)]]] \\ & \quad)\, \textbf{channel}\;\; (G',G)\;. \end{array}$$

Notice that each new process is less complicated than the original. The first process performs a general communication action, while the second process evaluates the guard set. Again, no concurrency is introduced by this transformation. The evaluation of the guards γ_i in the second process completes before a statement β_i initiates. This follows from the semantics of the general communication action.

To precisely describe the transformations that follow, we use quantification instead of abbreviated enumeration to denote structures of indefinite size. Using quantification notation, the above decomposition becomes:

$$
\begin{array}{ll}
& [\langle |\ i : 0 \leq i < n :: \gamma_i \longrightarrow \beta_i \rangle] \\
\triangleright & \quad \textbf{active}\ G'(1,n)\ \ \textbf{passive}\ G(n,1) \\
& \quad (G' : [\langle |\ i : 0 \leq i < n :: i \longrightarrow \beta_i \rangle] \\
& \quad \parallel *[[\overline{G} \longrightarrow [\langle |\ i : 0 \leq i < n :: \gamma_i \longrightarrow G(i) \rangle]]] \\
& \quad)\ \textbf{channel}\ \ (G', G)\ .
\end{array}
$$

Again, we write the final form of this decomposition as:

$$G' : [\langle |\ i : 0 \leq i < n :: i \longrightarrow \beta_i \rangle] \parallel (G(n,1)/\langle |\ i : 0 \leq i < n :: \gamma_i \rangle)\ .$$

The $G(n,1)$ denotes the name and size of the passive port used in the decomposition. In this case, the number of output values is n (one per guard) and the number of input values is 1.

3.2 Target Language of the Transformations

The target language of the program transformations is a slight extension of the above source language. Because of process decomposition, a restricted form of shared variables and shared ports is allowed. Processes may share ports and variables if references to these objects are not made concurrently. Also, concurrent execution of multiple statements is allowed and denoted by the comma. For example, $\alpha; \beta_1, \beta_2; \gamma$ denotes the execution of α, followed by the concurrent execution of β_1 and β_2, and, finally, the execution of γ.

Programs translated into this language are written in a different typeface than source language programs. This distinction is not necessary, but serves as an aid in describing which syntactic forms have already been or have yet to be translated. For compatibility with the notation of previous papers[4, 5], we use overlines to denote probes ($\overline{X}$) and up and down arrows to denote assignment ($x\uparrow, x\downarrow$).

3.3 Compilation of the Basic Constructs

Figure 3 displays all of the basic processes produced by the program transformations described in the next chapter. The remaining step is to compile these basic processes in self-timed circuits. These compilations are straight-forward applications of the methods described in [4, 5]. When possible, reshuffling is performed on passive communications introduced by process decomposition. The complete compilations are described in [1].

Both the program transformations and the resulting circuits for each basic process are described together succinctly as translation rules in circuit form. These rules are shown later in the text as Figures 4, 5, and 6.

2.	Process	Q'
6.	Sequence	$*[[\overline{Q} \longrightarrow A_1'; A_2'; Q]]$
7.	Skip	$*[Q]$
8.	Assignment	$*[[\overline{Q} \longrightarrow x\uparrow; Q]]$
9.	Com	$*[[\overline{Q} \longrightarrow L(j) : [\langle\mid k : 0 \leq k < n :: k \longrightarrow A_k'\rangle]; Q]]$
10.	Selection	$*[[\overline{Q} \longrightarrow G' : [1 \longrightarrow Q \mid 0 \longrightarrow \textbf{skip}\,]]]$
13.	Control	$*[[\overline{Q} \longrightarrow G' : [0 \longrightarrow Q(0) \mid \langle\mid i : 1 \leq i \leq n :: i \longrightarrow A_i'; Q(1)\rangle]]]$
14.	Seq guards	$*[[\overline{Q} \longrightarrow G' : [1 \longrightarrow Q(1) \mid 0 \longrightarrow P' : [\langle\mid i : 2 \leq i \leq n :: i-1 \longrightarrow Q(i)\rangle \mid 0 \longrightarrow Q(0)\,]]]]$
14.	Con guards	$*[[\overline{Q} \longrightarrow \langle,\ i : 0 \leq i \leq n :: G_i' : [1 \longrightarrow x_i\uparrow \mid 0 \longrightarrow \textbf{skip}]\rangle; [\langle\mid i : 0 \leq i \leq n :: x_i \longrightarrow Q(i); x_i\downarrow\rangle]]]$
14.	Conjunction	$*[[\overline{Q} \wedge x_1 \wedge \ldots \wedge \neg x_k \wedge \ldots \longrightarrow Q(i)]]$
15.	Seq AND	$*[[\overline{Q} \longrightarrow G_1' : [1 \longrightarrow G_2' : [1 \longrightarrow Q(1) \mid 0 \longrightarrow Q(0)] \mid 0 \longrightarrow Q(0)]]]$
15.	Con AND	$*[[\overline{Q} \longrightarrow G_1' : [1 \longrightarrow x_1\uparrow \mid 0 \longrightarrow x_1\downarrow], G_2' : [1 \longrightarrow x_2\uparrow \mid 0 \longrightarrow x_2\downarrow]; [x_1 \wedge x_2 \longrightarrow Q(1) \mid \neg x_1 \vee \neg x_2 \longrightarrow Q(0)]]]$
17.	Negation	$*[[\overline{Q} \longrightarrow G' : [1 \longrightarrow Q(0) \mid 0 \longrightarrow Q(1)]]]$
19.	Variable	$*[[\overline{Q} \wedge x \longrightarrow Q(1) \mid \overline{Q} \wedge \neg x \longrightarrow Q(0)]]$
20.	Probe	$*[[\overline{Q} \wedge \overline{X} \longrightarrow Q(1) \mid \overline{Q} \wedge \neg\overline{X} \longrightarrow Q(0)]]$
21.	True	$*[Q(1)]$

Figure 3: The above basic process types are generated as the result of the program transformations. Each process corresponds to a syntactic construct and is readily compiled into a self-timed circuit.

4 Transformations Rules

We now derive the program transformations corresponding to each syntactic construct. The equation numbers used to identify the transformations correspond to the numbers used in Figures 1 and 3. Several of the BNF rules are only used to define precedence. We do not define program transformations corresponding to these rules.

4.1 Processes, Declarations, and Channels

No transformations are applied to the parallel composition of processes or to the declaration of ports and variables. The only transformation needed in the first five BNF rules involves rule 2. For compatibility with the following transformations, one process decomposition is applied so that all ⟨sequence⟩ forms are guarded by a passive communication:

$$\begin{aligned} &\langle\text{process}\rangle \ \triangleright \{\langle\text{port}\rangle\}\{\langle\text{var}\rangle\}\langle\text{sequence}\rangle \\ &\triangleright\{\langle\text{port}\rangle\}\{\langle\text{var}\rangle\}(Q' \parallel (Q/\langle\text{sequence}\rangle)) \ . \end{aligned} \tag{2}$$

The basic process Q' performs exactly one active communication; thus, ⟨sequence⟩ also is executed exactly once.

4.2 Sequencing

The sequential composition of a ⟨statement⟩ and a ⟨sequence⟩ is transformed by process decomposition into the sequence of two active communications and a process implementing each statement:

$$\begin{array}{l}(Q/\langle\text{sequence}\rangle) \rhd (Q/\langle\text{statement}\rangle_1; \langle\text{sequence}\rangle_2) \\ \rhd *[[\overline{Q} \longrightarrow A_1'; A_2'; Q]] \parallel (A_1/\langle\text{statement}\rangle_1) \parallel (A_2/\langle\text{sequence}\rangle_2)\ .\end{array} \tag{6}$$

4.3 Skip

The **skip** statement is implemented as the infinite repetition of a passive communication:

$$(Q/\langle\text{statement}\rangle) \rhd (Q/\texttt{skip}) \rhd *[[\overline{Q} \longrightarrow \textbf{skip}; Q]] \rhd *[[\overline{Q} \longrightarrow Q]] \rhd *[Q]\ . \tag{7}$$

The probe of a passive communication is always a precondition to performing the action, so we may remove the selection statement with guard $\overline{Q}$.

4.4 Assignment

The assignment statement decomposes into

$$(P/\langle\text{statement}\rangle) \rhd (P/\langle\text{NAME}\rangle\ \texttt{up}) \rhd *[[\overline{P} \longrightarrow x\uparrow; P]]\ , \tag{8}$$

a simple process implementing a register. The name x represents an arbitrary ⟨NAME⟩. A similar decomposition is applied to assignments of `false`.

4.5 Communication

By applying the BNF rules corresponding to communication, we get

$$(Q/\langle\text{statement}\rangle) \rhd (Q/L(j) : [\langle|\ k : 0 \le k < n :: k \longrightarrow \langle\text{sequence}\rangle_k\rangle])\ , \tag{9}$$

where L and j represent an arbitrary ⟨NAME⟩ and ⟨INT⟩, respectively (rules 9, 11 and 12). Process decomposition produces new processes to implement each ⟨sequence⟩, yielding:

$$\begin{array}{l}*[[\overline{Q} \longrightarrow L(j) : [\langle|\ k : 0 \le k < n :: k \longrightarrow A_k'\rangle]; Q]] \\ \parallel \langle\|\ k : 0 \le k < n :: (A_k/\langle\text{sequence}\rangle_k)\rangle\ .\end{array}$$

4.6 Selection and Repetition

To derive the implementation of the control structures, we first review the semantics of these constructs. Operationally, the execution of the selection statement can be

described as: Repetitively evaluate each guard until one or more is true, then pick a true one and execute the corresponding command. The program transformation,

$$*[[\overline{Q} \longrightarrow [\gamma_1 \longrightarrow \beta_1 | \ldots | \gamma_n \longrightarrow \beta_n]; Q]] \\ \triangleright *[[\overline{Q} \longrightarrow [\gamma_1 \longrightarrow \beta_1; Q | \ldots | \gamma_n \longrightarrow \beta_n; Q | \textstyle\bigwedge_i \neg\gamma_i \longrightarrow \mathbf{skip}]]] \ ,$$

does not change the meaning of the selection statement, but makes it easier to implement because at least one guard will evaluate to true. Similarly, the repetition statement may be transformed into:

$$*[[\overline{Q} \longrightarrow *[\gamma_1 \longrightarrow \beta_1 | \ldots | \gamma_n \longrightarrow \beta_n]; Q]] \\ \triangleright *[[\overline{Q} \longrightarrow [\gamma_1 \longrightarrow \beta_1 | \ldots | \gamma_n \longrightarrow \beta_n | \textstyle\bigwedge_i \neg\gamma_i \longrightarrow Q]]] \ .$$

The new forms for selection and repetition are similar. Only the position of the communication Q is different. We can perform a general process decomposition on both the selection and repetition forms and use the same implementation of a guarded command set:

$$(Q/\langle \text{statement} \rangle) \ \triangleright \ (Q/[\langle \text{gcs} \rangle]) \\ \triangleright *[[\overline{Q} \longrightarrow G' : [1 \longrightarrow Q | 0 \longrightarrow \mathbf{skip}\,]]] \parallel (G(2,1)/\langle \text{gcs} \rangle) \ , \tag{10}$$

where

$$(G(2,1)/\langle \text{gcs} \rangle) \ \triangleright \ (G(2,1)/\langle | \ i : 1 \leq i \leq n :: \langle \text{expr} \rangle_i \texttt{-->} \langle \text{sequence} \rangle_i \rangle) \\ \triangleright *[[\overline{G} \longrightarrow [\langle | \ i : 1 \leq i \leq n :: \langle \text{expr} \rangle_i \longrightarrow \langle \text{sequence} \rangle_i ; G(1) \rangle \\ | \langle \wedge \ i : 1 \leq i \leq n :: \neg \langle \text{expr} \rangle_i \rangle \longrightarrow G(0) \\ ]]] \ .$$

The value returned by the communication G denotes the result of evaluating the disjunction of the guards within the guarded command set. Notice that in the selection statement, the guarded command set is reevaluated if a false value is returned. Repetition is the opposite of selection. Reevaluation occurs if the guarded command set returns true as described by the basic process:

$$*[[\overline{Q} \longrightarrow G' : [0 \longrightarrow Q | 1 \longrightarrow \mathbf{skip}\,]]] \ .$$

4.7 Guarded Command Sets

The guarded command set process is decomposed into a control process that sequences guard evaluation and the associated command execution, a set of processes that implement the commands and a process that evaluates the guard set:

$$(Q(2,1)/\langle \text{gcs} \rangle) \ \triangleright \ (Q(2,1)/\langle | \ i : 1 \leq i \leq n :: \langle \text{expr} \rangle_i \texttt{-->} \langle \text{sequence} \rangle_i \rangle) \\ \triangleright *[[\overline{Q} \longrightarrow G' : [0 \longrightarrow Q(0) | \langle | \ i : 1 \leq i \leq n :: i \longrightarrow A'_i ; Q(1) \rangle]]] \\ \parallel \langle \parallel \ i : 1 \leq i \leq n :: (A_i/\langle \text{sequence} \rangle_i) \rangle \\ \parallel (G(n+1,1)/\langle | \ i : 0 \leq i \leq n :: \langle \text{expr} \rangle_i \rangle) \ . \tag{13}$$

(Rules 13 and 14 are applied here.) The control process provides a separation between the issues of guard evaluation and statement execution by storing the guard that evaluated to true. This process distinguishes between the program state prior to the guarded command set and the program states following the arrows in each guarded command.

Guard evaluation completes before subsequent statements change variable values. The guard set process includes $\langle \text{expr} \rangle_0$, the negation of the disjunction of all the other expressions.

The non-trivial translation rules corresponding to the BNF rules 1–12 are shown in Figure 4. The remainder of the paper is concerned with the compilation of the guard set process.

4.8 Guard Set Evaluation

The semantics of the language does not specify the order in which to evaluate the guards. Whereas the other constructs require a strict ordering between command executions, concurrency may be exploited in guard evaluation. Because of the potential gains of concurrency, there is no single best scheme for guard evaluation. Instead, depending on the syntactic structure of the guard set and on invariant properties of the original program, different evaluation schemes will produce the most efficient implementation. Of the four schemes we describe, one is entirely sequential, while the other three represent different methods for using and controlling concurrency.

All four decomposition schemes require that the guard sets be in special forms. The special forms consists of both syntactic and invariant properties. For each property, we define a program transformation that, from an arbitrary guard set, produces an equivalent set in which the property holds. We choose to define both the properties and transformations because often a programmer can establish the properties (in particular, the invariants) by more subtle transformations. An automatic compiler can bypass these transformations if the programmer specifies in the source program that the desired properties are satisfied. We now define some properties and transformations on the guard set process,

$$(Q(n+1,1)/\langle |i : 0 \leq i \leq n :: \langle \text{expr} \rangle_i \rangle) \ .$$

4.8.1 Syntactic and Invariant Properties

Mutual Exclusion A guard set is **exclusive** if, when it is evaluated, at most one guard is true. This property is expressed by the invariant,

$$\neg \overline{Q} \vee \langle \wedge \ i,j : 0 \leq i,j \leq n \wedge i \neq j :: (\neg \langle \text{expr} \rangle_i \vee \neg \langle \text{expr} \rangle_j) \rangle \ .$$

The invariant can always be achieved by successive strengthenings of the guards, which will produce an entirely deterministic implementation of the original non-deterministic guard set. Precisely, for $1 \leq i \leq n$,

$$\langle \text{expr} \rangle'_i \equiv \langle \wedge \ j : 1 \leq j < i :: \neg \langle \text{expr} \rangle_j \rangle \wedge \langle \text{expr} \rangle_i$$

and

$$\langle \text{expr} \rangle'_0 \equiv \langle \text{expr} \rangle_0 \ .$$

(Read '$\equiv$' as "is defined as".)

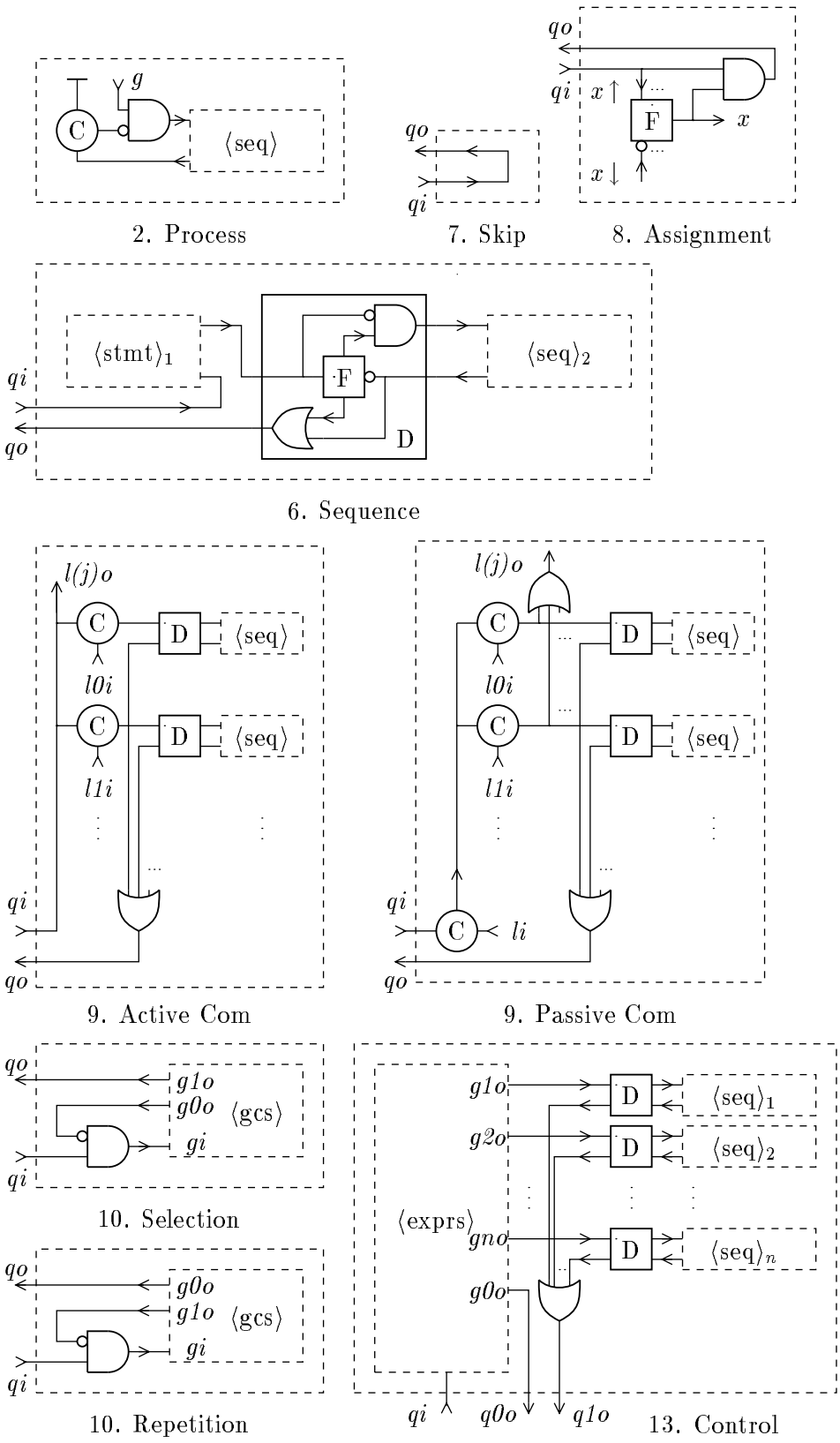

Figure 4: Translation Rules Excluding Those for Guard Set Evaluation

Disjoint Disjunctive Form Two of the implementation schemes require that each guard be in so-called "disjoint disjunctive form". Each guard must be expressed in AND-OR form, and when the guard set is evaluated, at most one AND term is true. That is, a guard set is **ddf**, if for each expression $\langle \text{expr} \rangle_i$, $0 \leq i \leq n$,

$$\neg \overline{Q} \vee \langle \wedge\ j, k : 0 \leq j, k < m_i \wedge j \neq k :: (\neg \langle \text{conj} \rangle_i^j \vee \neg \langle \text{conj} \rangle_i^k) \rangle\ ,$$

where $\langle \text{expr} \rangle_i \equiv \langle \vee\ j : 0 \leq j < m_i :: \langle \text{conj} \rangle_i^j \rangle$ and each $\langle \text{conj} \rangle_i^j$ is a simple conjunction of possibly negated variables and probes.

The program transformation to achieve the **ddf** property is similar to the transformation of an arbitrary expression into disjunctive normal form, except that the conjuncts must be successively strengthened to achieve disjointness. See [1] for details. Notice that this transformation potentially increases the size of a guard set to an exponential in the number of its variables.

Negated Probes An expression is *stable* if once it becomes true, it remains true. The underlying compilation method allows only restricted implementations of non-stable expressions. Since process decomposition does not introduce concurrency, the guard set process is not active concurrently with any processes modifying variables; thus, all variables are stable in the guard set process. All positive probes are, as well, but negated probes are not. A negated probe may change asynchronously from true to false. We call a guard set **noneg** if it contains no negated probes.

Any expression containing a negated probe is potentially non-stable. We define a transformation that stablizes all negated probes. Each probe in the guard set is evaluated and assigned to a local variable before the boolean expressions are evaluated. References of a probe's value within the guard set refer to the corresponding variable's value.

Let $\mathcal{X}$ be the set of all negated probes named in the guard set. To transform the guard set into **noneg** form,

$$\begin{array}{l} (Q(n+1,1)/\langle \| \ i : 0 \leq i \leq n :: \langle \text{expr} \rangle_i \rangle) \\ \rhd * [[\overline{Q} \longrightarrow \langle,\ X : X \in \mathcal{X} :: [\overline{X} \longrightarrow x\uparrow\ |\neg \overline{X} \longrightarrow x\downarrow]\rangle; \\ \qquad\qquad G' : [\langle |\ i : 0 \leq i \leq n :: i \longrightarrow Q(i)\rangle]]] \\ \|\ (G(n+1,1)/\langle |\ i : 0 \leq i \leq n :: \langle \text{expr} \rangle_i' \rangle)\ , \end{array}$$

where, for $0 \leq i \leq n$,

$$\langle \text{expr} \rangle_i' \equiv \langle X : X \in \mathcal{X} :: \text{replace } \overline{X} \text{ by } x \rangle \langle \text{expr} \rangle_i\ .$$

Non-atomic Evaluation If a guard set is not evaluated atomically, expressions that change value during the evaluation cause special problems. For example, the expression $\overline{X} \vee \neg \overline{X}$ may evaluate to false if different values for $\overline{X}$ are used in the two subexpressions. A guard set is **nonatomic** if the subexpressions within it can be evaluated in any order. This property is achieved if each probe in the guard set is named only once. The same transformation used to achieve the **noneg** property will put an arbitrary guard set in this form if we let $\mathcal{X}$ be the set of all probes named more than once.

4.8.2 Evaluation Schemes

Sequential Guard Evaluation This scheme requires that the guard set fulfill the **nonatomic** property. The guards are evaluated one by one until one evaluates to true. If none evaluate to true, the communication $Q(0)$ is performed. Process decomposition for this scheme may be defined recursively. If $n > 1$,

$$\begin{array}{l}(Q(n+1,1)/\langle|\ i : 0 \le i \le n :: \langle \text{expr} \rangle_i\rangle) \\ \triangleright *[[\overline{Q} \longrightarrow G' : [1 \longrightarrow Q(1) \\ \qquad\qquad |0 \longrightarrow P' : [\langle|\ i : 2 \le i \le n :: i-1 \longrightarrow Q(i)\rangle \\ \qquad\qquad\qquad\qquad |0 \longrightarrow Q(0) \\ \qquad\qquad\qquad\qquad]]]] \\ \|\ (G(2,1)/\langle \text{expr} \rangle_1) \\ \|\ (P(n,1)/\langle \text{expr} \rangle_0 \mid \langle|\ i : 2 \le i \le n :: \langle \text{expr} \rangle_i\rangle)\ ;\end{array}$$

and, if $n = 1$,

$$(Q(2,1)/\langle \text{expr} \rangle_0 \mid \langle \text{expr} \rangle_1) \triangleright (Q(2,1)/\langle \text{expr} \rangle_1)\ .$$

We note that the **exclusive** property is not required for this decomposition. Conditional evaluation ensures mutual exclusion among the guards. For the same reason, $\langle \text{expr} \rangle_0$ is not used.

Evaluation of each individual guard is implemented conditionally. Sequential evaluation of the `and` connective starts by evaluating the first sub-expression. If the first sub-expression is true, the value of the second sub-expression determines the value of conjunction. Otherwise, the value of the conjunction is false:

$$\begin{array}{l}(Q(2,1)/\langle \text{conjunct} \rangle) \triangleright (Q(2,1)/\langle \text{primary} \rangle_1 \texttt{ and } \langle \text{conjunct} \rangle_2) \\ \triangleright *[[\overline{Q} \longrightarrow G_1' : [1 \longrightarrow G_2' : [1 \longrightarrow Q(1)|0 \longrightarrow Q(0)]|0 \longrightarrow Q(0)]]] \\ \|\ (G_1(2,1)/\langle \text{primary} \rangle_1) \|\ (G_2(2,1)/\langle \text{conjunct} \rangle_2)\ .\end{array}$$

The sequential scheme and the next scheme (concurrent-all) share the same decompositions for the remaining expression constructs. De Morgan's Law allows the `or` connective to be defined in terms of `and` and `not`:

$$\begin{array}{l}(Q(1,2)/\langle \text{expr} \rangle) \triangleright (Q(1,2)/\langle \text{conjunct} \rangle_1 \texttt{ or } \langle \text{expr} \rangle_2) \\ \triangleright (Q(1,2)/\texttt{not (not } \langle \text{conjunct} \rangle_1 \texttt{ and not } \langle \text{expr} \rangle_2))\ .\end{array} \tag{15}$$

Similarly, `false` is defined in terms of `true` and `not`. Negation exchanges the results of the evaluation:

$$\begin{array}{l}(Q(1,2)/\langle \text{primary} \rangle) \triangleright (Q(1,2)/\texttt{not } \langle \text{primary} \rangle_1) \\ \triangleright *[[\overline{Q} \longrightarrow G' : [0 \longrightarrow Q(1)|1 \longrightarrow Q(0)]]] \|\ (G(1,2)/\langle \text{primary} \rangle_1)\ .\end{array} \tag{17}$$

The evaluation of a simple variable is implemented by the process:

$$\begin{array}{l}(Q(1,2)/\langle \text{primary} \rangle) \triangleright (Q(1,2)/\langle \text{NAME} \rangle) \\ \triangleright *[[\overline{Q} \longrightarrow [x \longrightarrow Q(1)|\neg x \longrightarrow Q(0)]]]\ .\end{array} \tag{19}$$

If a probe is named only once in a guard set, the evaluation of a probe is implemented by the process:

$$\begin{array}{l}(Q(2,1)/\langle \text{primary} \rangle) \triangleright (Q(2,1)/\texttt{probe } \langle \text{NAME} \rangle) \\ \triangleright *[[\overline{Q} \longrightarrow [\overline{X} \longrightarrow Q(1)|\neg \overline{X} \longrightarrow Q(0)]]]\ .\end{array} \tag{20}$$

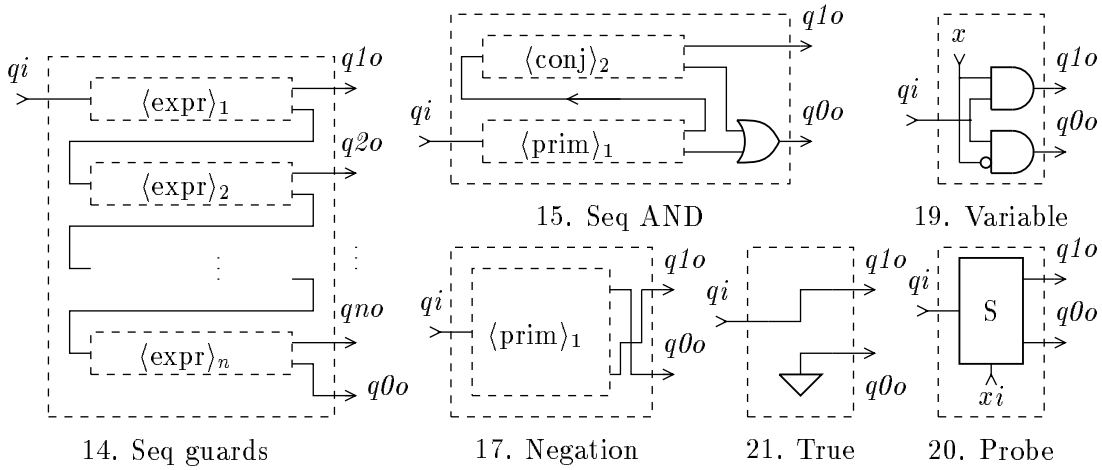

Figure 5: Translation Rules for Sequential Guard Evaluation

The evaluation of **true** has an implementation similar to that of **skip**:

$$\begin{array}{l}(Q(2,1)/\langle \text{primary}\rangle) \triangleright (Q(2,1)/\texttt{true}) \\ \triangleright *[[\overline{Q} \longrightarrow [\textbf{true} \longrightarrow Q(1) | \textbf{false} \longrightarrow Q(0)]]] \triangleright *[Q(1)] \ .\end{array} \tag{21}$$

Figure 5 shows all the translation rules used in the sequential guard evaluation scheme.

Concurrent-all Guard Evaluation This scheme requires that the guard set fulfill both the **exclusive** and the **nonatomic** properties. Each guard is evaluated separately and concurrently. The variable corresponding to the true guard (exactly one because the **exclusive** property holds) is raised. When all guards have been evaluated, the communication corresponding to the set variable is performed, and then the variable is reset:

$$\begin{array}{l}(Q(n+1)/\langle |\ i : 0 \le i \le n :: \langle \text{expr}\rangle_i\rangle) \\ \triangleright *[[\overline{Q} \longrightarrow \langle,\ i : 0 \le i \le n :: G_i : [1 \longrightarrow x_i\uparrow\ |0 \longrightarrow \textbf{skip}]\rangle; \\ \qquad\qquad [\langle |\ i : 0 \le i \le n :: x_i \longrightarrow Q(i); x_i\downarrow\rangle]]] \\ \|\ \langle \|\ i : 0 \le i \le n :: (G_i(2,1)/\langle \text{expr}\rangle_i)\rangle \ .\end{array}$$

To evaluate the `and` connective, both sub-expressions are evaluated concurrently and the results stored in variables. The conjunction of these values is returned by the communication Q:

$$\begin{array}{l}(Q(2,1)/\langle \text{conjunct}\rangle) \triangleright (Q(2,1)/\langle \text{primary}\rangle_1 \texttt{and} \langle \text{conjunct}\rangle_2) \\ \triangleright *[[\overline{Q} \longrightarrow G_1' : [1 \longrightarrow x_1\uparrow\ |0 \longrightarrow x_1\downarrow], G_2' : [1 \longrightarrow x_2\uparrow\ |0 \longrightarrow x_2\downarrow]; \\ \qquad\qquad [x_1 \wedge x_2 \longrightarrow Q(1) \\ \qquad\qquad |\neg x_1 \vee \neg x_2 \longrightarrow Q(0) \\ \qquad\qquad]]] \\ \|\ (G_1(2,1)/\langle \text{primary}\rangle_1) \|\ (G_2(2,1)/\langle \text{conjunct}\rangle_2) \ .\end{array}$$

Figure 6 shows the translation rules corresponding to concurrent-all guard evaluation.

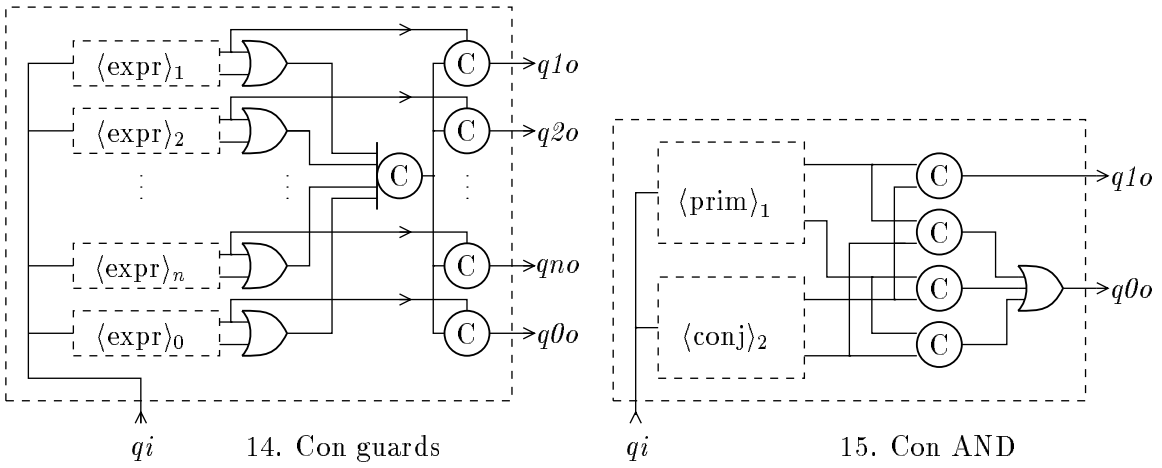

Figure 6: Translation Rules for Concurrent-all Guard Evaluation

Concurrent-one Guard Evaluation In this scheme, all guards are evaluated simultaneously. The evaluation of each guard (in fact, each conjunct of each guard) is implemented by a separate process. For this decomposition to be valid, no two of these processes may operate concurrently. This is ensured by both the **exclusive** and the **ddf** properties. After decomposition, each remaining basic process implements a simple conjunction of variables and probes:

$$(Q(n+1,1)/\langle |\ i : 0 \le i \le n :: \langle \vee\ j : 0 \le j < m_i :: \langle \mathrm{conj} \rangle_i^j \rangle \rangle)$$
$$\triangleright \langle \|\ i : 0 \le i \le n :: \langle \|\ j : 0 \le i < m :: *[[\overline{Q} \wedge \langle \mathrm{conj} \rangle_i^j \longrightarrow Q(i)]] \rangle \rangle \ .$$

The **noneg** property is required for the implementation of each conjunction.

Concurrent-one-wait Guard Evaluation In special cases, guard evaluation may be implemented as above without performing the all-false communication $Q(0)$. This evaluation scheme is possible when: i) the guard set is part of a selection statement (no repetitions); ii) the guard set has the **exclusive** and **ddf** properties; and iii) after replacing $\langle \mathrm{expr} \rangle_0$ by **false**, no negated probes are named in the guard set.

4.8.3 Applying the Guard Evaluation Schemes to an Example

We illustrate the four different schemes for decomposing guard sets by applying these schemes to a small program fragment. Consider the program fragment:

$$\ldots; [\overline{X} \wedge s \longrightarrow \ldots | \overline{Y} \longrightarrow \ldots]; \ldots \ .$$

Syntax-directed application of the program transformations results in an intermediate form containing the process:

$$(Q(3,1)/\overline{X} \wedge s \mid \overline{Y} \mid \neg(\overline{X} \wedge s \vee \overline{Y}\ )) \quad .$$

We construct implementations of this guard set using the four different evaluation schemes. These circuits are shown in Figure 7. The number of two-input operators required for each implementation is used as a general space comparison between the schemes. For operators with more than two inputs, each extra input adds $\frac{1}{2}$ to the operator count.

Sequential Since the guard set has the **nonatomic** property (excluding the all-false expression), the decomposition is straightforward, requiring no initial transformation of the guard set. The resulting circuit requires 2 *AND*, 2 *SYNC* and 1 *OR* operators.

Concurrent-all We first put the guard set into a form that satisfies the **exclusive** property:

$$\overline{X} \wedge s \mid \neg(\overline{X} \wedge s) \wedge \overline{Y} \mid \neg(\overline{X} \wedge s) \wedge \neg\overline{Y} \ .$$

Since $\overline{X}$ and $\overline{Y}$ are named more than once, this transformed guard set does not have the **nonatomic** property. However, in this scheme, the evaluation of common subexpressions can be shared between guards. In particular, probes need only be evaluated once per guard set; thus, the **nonatomic** property is satisfied. The resulting circuit requires 2 *AND*, 2 *SYNC*, $16\frac{1}{2}$ *C*, and $7\frac{1}{2}$ *OR* operators.

Concurrent-one In this case, the guard set must satisfy both the **exclusive** and the **ddf** properties. The transformed guard set is:

$$\overline{X} \wedge s \mid \neg\overline{X} \wedge \overline{Y} \vee \overline{X} \wedge \neg s \wedge \overline{Y} \mid \neg\overline{X} \wedge \neg\overline{Y} \vee \overline{X} \wedge \neg s \wedge \neg\overline{Y} \ .$$

Notice the extra literals needed to ensure disjointness among the conjuncts. Since negated probes are named in this guard set, the value of each probe is assigned to a variable, thus satisfying the **noneg** property. The resulting circuit requires 2 *SYNC*, 2 *FF*, 1 *C*, 4 *OR* and $12\frac{1}{2}$ *AND* operators.

Concurrent-one-wait In order to use this scheme, the **exclusive** property must hold on the original guard set, that is $\neg\overline{Q} \vee \neg\overline{X} \vee \neg s \vee \neg\overline{Y}$ must be an invariant of the original program. If this is the case, the guard set

$$\overline{X} \wedge s \mid \overline{Y} \mid \textbf{false}$$

can be implemented directly without any transformations, resulting in a simple circuit requiring only $2\frac{1}{2}$ *AND* operators.

4.8.4 Comparison of the Guard Evaluation Schemes

In the above example, the sequential and the concurrent-one-wait schemes produce the most efficient circuits. This is not always the case. For other guard sets, each of the schemes can produce the best implementation. We discuss pros and cons of the different schemes.

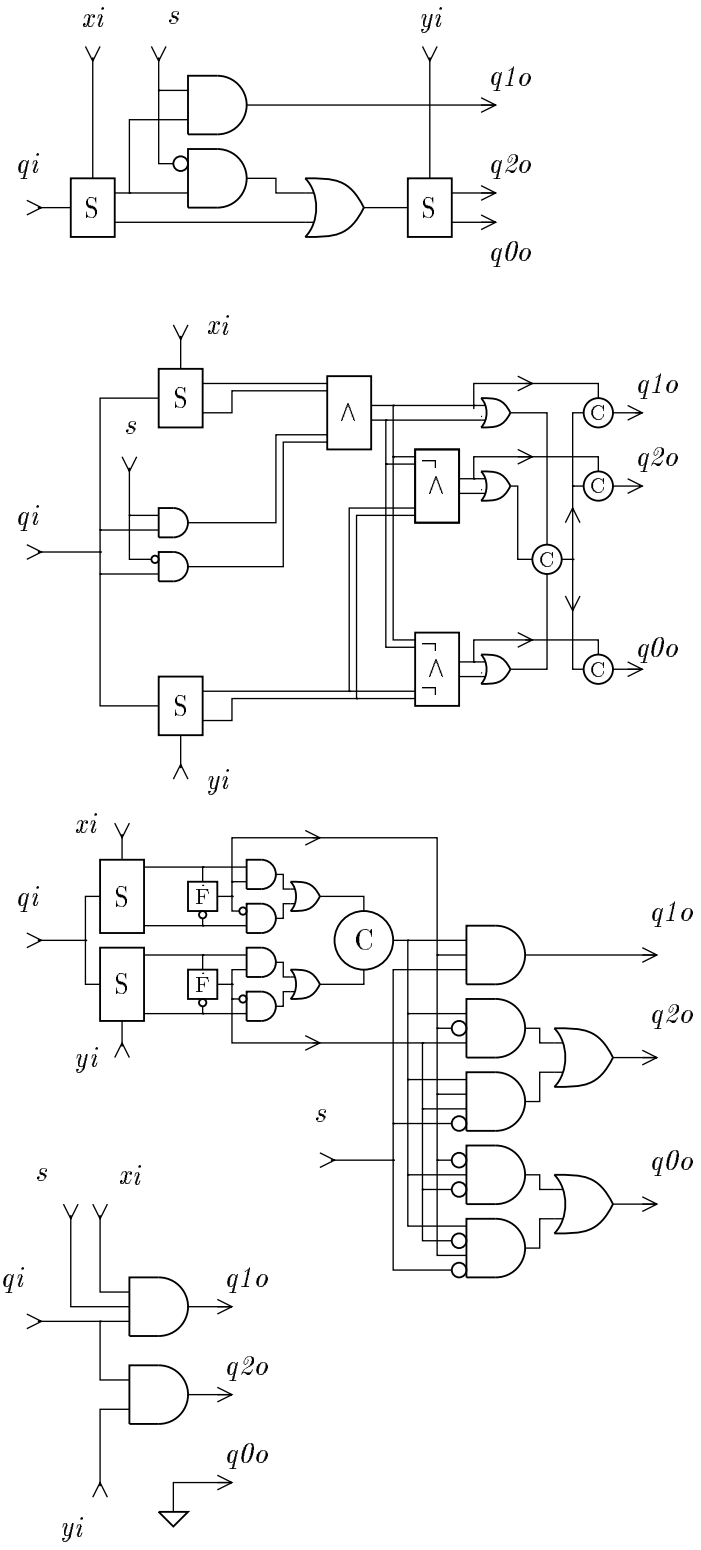

Figure 7: The four circuits show four different guard evaluation schemes applied to the guard set $\overline{X} \wedge s \mid \overline{Y}$. From top to bottom, the circuits were produced by the sequential, concurrent-all, concurrent-one, and concurrent-one-wait schemes. In the circuit produced by the concurrent-all scheme, the boxes enclosing the $\wedge$ symbol represent the circuit implementing the `and` construct, as shown in Figure 6.

Sequential The sequential scheme provides a straightforward implementation of an arbitrary guard set in space that is proportional to the size of the guard set's representation in the source language. Unfortunately, because of its sequential nature, the time needed for evaluation is also linearly related to its size.

Concurrent-all This evaluation scheme offers several potential advantages over the previous one. For guard sets with many guards, the time needed to evaluate the guard set is proportional to the logarithm of the number of guards. The ability to do common sub-expression elimination at no cost is an added benefit; however, the basic processes have larger implementations. While this scheme has the best asymptotic area-time performance, we have yet to find an application large enough to reap its benefits.

Concurrent-one While exponential blow-up may occur when transforming pathological guard sets into disjoint disjunctive form, this scheme produces the smallest and fastest implementations of most expressions that do not contain probes.

Concurrent-one-wait In the cases when the programmer can prove the **exclusive** property without introducing negated probes, this scheme applies and provides a non-polling implementation that does not dissipate any static power. Again, exponential blow-up may occur, but typical implementations are much smaller and much faster than the other schemes.

5 Automatic Compiler

We have constructed an automatic compiler which applies the translation rules derived in this paper. The self-timed circuit description produced by the compiler is then used as input by an automatic place-and-route tool which produces a standard cell implementation of the circuit in VLSI. Using this completely automatic design method, we have fabricated a functionally correct chip implementing a worm-hole message routing system[2].

The translation method produces correct, self-timed implementations of arbitrarily large concurrent programs, and because each translation rule is of fixed size, the size of the implementation is no worse than linearly related to the size of the source program. The translation method and the compiler provide a constructive proof that this design methodology, based on programs, represents a practical approach to the design of VLSI systems.

Acknowledgments and References

We wish to thank Pieter Hazewindus for his comments on early versions of this manuscript and Andy Fyfe for his PostScript expertise. This research is sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

References

[1] S.M. Burns, *Automated Compilation of Concurrent Programs into Self-timed Circuits*, M.S. Thesis, Caltech-CS-TR-88-2.

[2] S.M. Burns and A.J. Martin, "Syntax-directed Translation of Concurrent Programs into Self-timed Circuits", *Advanced Research in VLSI: Proc. Fifth MIT Conference*, ed. J. Allen and F. Leighton, MIT Press, pp 35–50 (1988)

[3] C.A.R. Hoare, "Communicating Sequential Processes", *Comm. ACM* 21, 8, pp 666–677 (August 1978)

[4] A.J. Martin, "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits", *Distributed Computing*, 1, pp 226–234 (1986)

[5] A.J. Martin, "The Design of a Self-Timed Circuit for Distributed Mutual Exclusion", *Proc. Chapel Hill Conf. VLSI*, ed. H. Fuchs, pp 247–260 (1985)

[6] A.J. Martin, "The Probe: an Addition to Communication Primitives", *Information Processing Letters*, 20, pp 125–130 (1985)