LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Scheduling and Synthesis of Operation-Centric Hardware Descriptions

Computation Structures Group Memo 426
November 9, 1999

James C. Hoe and Arvind
MIT Laboratory for Computer Science
Cambridge, MA 02139
{jhoe,arvind}@lcs.mit.edu

This paper describes research done at the MIT Laboratory for Computer Science. Funding for this work is provided in part by the Defense Advanced Research Projects Agency of the Department of Defense under the Ft. Huachuca contract DABT63-95-C-0150 and by the Intel Corporation. James C. Hoe is supported by an Intel Foundation Graduate Fellowship.

Scheduling and Synthesis of Operation-Centric Hardware Descriptions

James C. Hoe and Arvind
MIT Laboratory for Computer Science
Cambridge, MA 02139
{jhoe,arvind}@lcs.mit.edu

November 9, 1999

Abstract

High-level specifications, such as processor manuals, are often operation-centric where the behavior of a system is specified on an operation-by-operation basis. Whereas, most synthesizable descriptions such as schematic or RTL are state-centric. This paper presents a synthesis algorithm that compiles an operation-centric description into a state-centric synthesizable RTL description by finding a consistent and efficient scheduling of the independently prescribed operations.

1 Introduction

Traditionally, hardware implementation is captured in a state-centric manner that specifies for each state element in a system, its new state after every time step. This style of hardware specification includes both schematic capture and text-based Register Transfer Language descriptions (RTL's). In a state-centric description, because the events in different parts of the system are not semantically coupled, it is easy to make a mistake in the coordination and synchronization of related events. It is also difficult to modify the timing in one part of the system without considering its interaction with the rest of the system.

On the other hand, high-level specifications of hardware designs are often operation-centric, that is, the collective behavior of the whole system

is described on an operation-by-operation basis. The specification of each operation is self-contained, with minimal dependence between specifications of different operations. Specifying an operation involves stating when an operation should take place and what its effect is. Semantically, these operations are atomic and are interleaved sequentially during an execution. For example, a microprocessor manual specifies the behavior of a processor on an instruction-by-instruction basis. For each instruction, the manual specifies how the processor state registers are affected according to that instruction alone. In contrast, one can imagine the difficulty in reading a state-centric processor manual that describes a processor's behavior by giving the next-state logic of each register.

1.1 Term Rewriting Systems

Term Rewriting Systems (TRS's)[2] can be used as a source language for operation-centric hardware descriptions. In the past, TRS's have been used extensively to give operational semantics of programming languages. Recently, TRS has found applications in computer architecture design and verification. A processor with out-of-order and speculative execution has been modeled as a TRS and verified against a specification TRS[1].

A TRS models a hardware design by a set of terms and a set of rewrite rules. The set of terms corresponds to the different states of the system while the rules specify the allowed state transitions. Abstractly, a rule consists of two components, the π and δ functions. The π function determines when a rule is applicable to a term, and the δ function determines the new term when a rule is applied. The effect of a rule on a term can be captured by the function,

$$\lambda\ t.\ \mathit{if}\ \pi(t)\ \mathit{then}\ \delta(t)\ \mathit{else}\ t$$

In the context of an operation-centric behavioral description, π specifies when an operation could take place, and δ captures the effect of that operation on the hardware state.

During the execution of a TRS, the effect of a rewrite rule is atomic, that is, the whole term is "read" in one step and if the rule is applicable then a new term is returned in the same step. If several rules are applicable, then any one of them is chosen nondeterministically and applied. Afterwards, all rules are re-evaluated for applicability on the new term. Starting from an initial term, successive rewriting produces a sequence of terms. The

execution stops at a term that cannot be rewritten using any rule.

1.2 Synthesis of Operation-Centric Hardware Descriptions

This paper investigates the problem of synthesizing operation-centric hardware description into state-centric implementation descriptions, i.e. synthesizable synchronous RTL. Detailed synthesis algorithms are presented in the context of an abstract transition system (ATS), which is a suitable abstract representation of operation-centric descriptions in general. The crux of the synthesis algorithm involves finding a valid composition of the independently-prescribed operations into a coherent state transition system. For performance reasons, the synthesized implementation should carry out as many operations concurrently as possible, and yet still produce a behavior that is consistent with the atomic and sequential execution semantics of the original operation-centric specification. The algorithms presented in this paper have been implemented in a TRS-to-RTL compiler[7].

1.3 Paper Organization

The next section defines the ATS abstraction for operation-centric hardware description. Section 3 offers a straight-forward reference implementation of an ATS using synchronous circuit elements. Section 3 also establishes the correctness and performance criteria of an implementation. Sections 4 and 5 present two refinements to the synthesis algorithm that uses static analysis to find optimizations that improve the performance of synthesized implementations. Section 6 presents related work in high-level description and synthesis, as well as our conclusions.

2 Abstract Transition Systems

An abstract transition system (ATS) is described by $\mathcal{S}$, $\mathcal{S}^o$ and $\mathcal{T}$. $\mathcal{S}$ is a list of state elements, and $\mathcal{S}^o$ is a list of constants. $\mathcal{T}$ is a list of state transitions. The components of an ATS is summarized in Figure 1.

2.1 State Elements

An element in $\mathcal{S}$ can be a register (R), an array (A), or a FIFO (F). A register R can store an integer value up to a maximum word size. The value stored in a register can be referenced in an expression using the side-effect-free R.***get***()

$$
\begin{array}{rcl}
\mathcal{ATS} & = & <\mathcal{S},\mathcal{S}^o,\mathcal{T}> \\
\mathcal{S} & = & < \mathsf{R}_1,...,\mathsf{R}_{NR},\ \mathsf{A}_1,...,\mathsf{A}_{NA},\ \mathsf{F}_1,...,\mathsf{F}_{NF} > \\
\mathcal{S}^o & = & < V^{R_1},...,V^{R_{NR}} > \\
\mathcal{T} & = & \{\ T_1,...,T_M\ \} \\
T & = & <\pi,\alpha> \\
\pi & = & \mathit{exp} \\
\alpha & = & <\mathbf{a}^{R_1},...,\mathbf{a}^{R_{NR}},\ \mathbf{a}^{A_1},...,\mathbf{a}^{A_{NA}},\ \mathbf{a}^{F_1},...,\mathbf{a}^{F_{NF}} > \\
\mathbf{a}^R & = & \epsilon \ [] \ \mathit{set}(\mathit{exp}) \\
\mathbf{a}^A & = & \epsilon \ [] \ \mathit{a\text{-}set}(\mathit{exp}_{idx},\mathit{exp}_{data}) \\
\mathbf{a}^F & = & \epsilon \ [] \ \mathit{enq}(\mathit{exp}) \ [] \ \mathit{deq}() \ [] \ \mathit{en\text{-}deq}(\mathit{exp}) \ [] \ \mathit{clear}() \\
\mathit{exp} & = & \mathit{constant} \ [] \ \mathit{Op}(\mathit{exp}_1,\mathit{exp}_2,...,\mathit{exp}_n) \\
 & [] & \mathsf{R}.\mathit{get}() \ [] \ \mathsf{A}.\mathit{a\text{-}get}(\mathit{idx}) \\
 & [] & \mathsf{F}.\mathit{first}() \ [] \ \mathsf{F}.\mathit{notfull}() \ [] \ \mathsf{F}.\mathit{notempty}()
\end{array}
$$

Figure 1: $\mathcal{ATS}$ Summary

query operator. For conciseness, R.***get***() can be abbreviated simply as R. A register's content can be set to *value* by the R.***set***(*value*) action operator. Any number of queries is allowed in an atomic update step, but each register only allows at most one ***set*** action in each atomic update step. An *atomic update step* is defined in Section 2.3 where the operational semantics of an ATS is defined.

An array A can store a fixed number of values. The content of individual entries in an array can be indexed for reading and writing. The value stored in the *idx*'th entry can be referenced in an expression using the side-effect-free A.***a-get***(*idx*) query operator. A.***a-get***(*idx*) can be abbreviated as A[*idx*]. The content of the *idx*'th entry can be set to *value* using the A.***a-set***(*idx*,*value*) action operator. Out-of-bound queries or actions on an array are not allowed. Each array only allows at most one ***a-set*** action in each atomic update step, but any number of queries on an array is allowed.

A FIFO F stores an unspecified but finite number of values in a first-in-first-out manner. The oldest value in a FIFO can be referenced in an expression using the side-effect-free F.***first***() query operator, and can be removed by the F.***deq***() action operator. A new *value* can be added to a FIFO using the F.***enq***(*value*) action operator. F.***en-deq***(*value*) is a compound action that enqueues a new value after dequeuing the oldest value. In addition, the entire contents of a FIFO can be cleared using the F.***clear***() action operator. Underflowing or overflowing a FIFO by an ***enq*** or a ***deq*** action is not allowed. The status of a FIFO can be queried using the side-effect-free

F.***notfull***() and F.***notempty***() query operators that return a boolean value. These boolean status flags may be false-negative, that is, F.***notfull***() may return *false* even if F is not full. Each FIFO only allows at most one action in each atomic update step, but any number of queries are allowed.

Due to the lack of space, this paper does not discuss input/output. Input and output are accomplished using register like state elements. An input state element (I) is like a register but without the ***set*** operator. I.***get***() returns the value of an external input. An output state element (O) supports both ***set*** and ***get***, and its content is visible outside of the ATS.

2.2 State Transitions

An element in $\mathcal{T}$ is a transition which is a pair, $<\pi,\alpha>$. π is a value expression. A value expression can contain arithmetic and logical operations on values. A value in an expression can be a constant or a value queried from a state element. Given an ATS whose $\mathcal{S}$ has N elements, α is a list of N actions, one for each state element. An action is specified as an action operator plus its arguments in terms of value expressions. A null action is represented by the symbol ϵ. The argument expressions to an action need to be evaluated before the action can be applied to the state element. For all actions in α, the i'th action of α must be valid for the i'th state element in $\mathcal{S}$.

2.3 Operational Semantics

At the start of an execution, all entries in all A's contain 0's, and all F's are empty. The initial value of R_i is taken from the i'th element of $\mathcal{S}^o$. From this initial state, the execution of an ATS takes place as a sequence of state transitions in atomic update steps.

At the start of an atomic update step, all π expressions in $\mathcal{T}$ are evaluated using the current contents of the state elements. In a given step, an applicable transition is a transition whose π expression evaluates to true. All argument expressions to the actions in α's are also evaluated using the current state of $\mathcal{S}$.

At the end of an atomic update step, one transition is selected nondeterministically from all applicable transitions. $\mathcal{S}$ is then modified according to the actions in the selected transition. For all actions in the selected α, the i'th action is applied to the i'th state element in $\mathcal{S}$, using the argument values evaluated at the beginning of the step. If an illegal action

or combination of actions is performed on a state element then the entire state transition system halts with an error. A valid ATS specification never produces an error.

The atomic update step repeats until $\mathcal{S}$ is in a state where none of the transitions is applicable. In this case, the system halts without an error. Alternatively, a sequence of transitions can lead $\mathcal{S}$ to a state where selecting any applicable transition leaves $\mathcal{S}$ unchanged. The system also halts without error in this case. Some systems may never halt. Due to non-determinism, some systems could halt in different states from one execution to the next.

2.4 Functional Interpretation

Thus far, ATS has been defined in terms of state elements and actions with side-effects. In certain situations, it is convenient to assign a functional interpretation to ATS. There is a natural one-to-one correspondence between the two interpretations. In a functional interpretation, the state of $\mathcal{S}$ is represented by a collection of values. R is represented by a value, A is an array of values, and F is an ordered list of values. Using this value representation of state, δ, the state-to-state function corresponding to α=<$\mathbf{a}^{R_1}$..., $\mathbf{a}^{A_1}$..., $\mathbf{a}^{F_1}$...>, can be defined as,

$$
\begin{array}{l}
\delta(s) = \mathit{let} \\
\qquad <\mathsf{R}_1..., \mathsf{A}_1..., \mathsf{F}_1...> = s \\
\quad \mathit{in} \\
\qquad < \mathbf{apply}(\mathbf{a}^{R_1}, \mathsf{R}_1)..., \ \mathbf{apply}(\mathbf{a}^{A_1}, \mathsf{A}_1)..., \\
\qquad\qquad\qquad\qquad \mathbf{apply}(\mathbf{a}^{F_1}, \mathsf{F}_1)...>
\end{array}
$$

Applying an action entails computing the new value of a state element based on the old values of state elements.

$$
\begin{array}{l}
\mathbf{apply}(\mathbf{a}, v) = \mathit{case}\ \mathbf{a}\ \mathit{of} \\
\qquad\qquad \mathbf{set}(v') \Rightarrow v' \\
\qquad\qquad \mathbf{a\text{-}set}(\mathit{idx}, \mathit{data}) \Rightarrow v[\mathit{idx} := \mathit{data}] \\
\qquad\qquad \mathbf{enq}(v') \Rightarrow v; v' \\
\qquad\qquad \mathbf{deq}() \Rightarrow \mathit{let}\ \mathit{first}; \mathit{rest} = v\ \mathit{in}\ \mathit{rest} \\
\qquad\qquad \mathbf{en\text{-}deq}(v') \Rightarrow \mathit{let}\ \mathit{first}; \mathit{rest} = v\ \mathit{in}\ \mathit{rest}; v' \\
\qquad\qquad \mathbf{clear}() \Rightarrow \mathit{empty\ list} \\
\qquad\qquad \epsilon \Rightarrow v
\end{array}
$$

Using a functional interpretation, an execution of ATS produces a sequence of values that corresponds to the sequence of state transitions in the normal interpretation using side-effects on state elements.

2.5 Example: A Simple Processor

$\mathcal{S}$ of a simple processor is $<\mathsf{R}_{PC}, \mathsf{A}_{RF}, \mathsf{A}_{IMEM}, \mathsf{A}_{DMEM}>$ where R_{PC} is a 16-bit program counter, A_{RF} is a general purpose register file of four 16-bit values, A_{IMEM} is a 2^{16}-entry array of 24-bit instruction words, and A_{DMEM} is a 2^{16}-entry array of 16-bit values.

$\mathcal{T}$ consists of transitions that correspond to the execution of different instructions in the processor. Let 0, 2 and 3 be the numerical values assigned to the instruction opcode Loadi (Load Immediate), Add (Triadic Register Add) and Bz (Branch if Zero). Also let the instruction word stored in A_{IMEM} be decoded as follows,

$opcode$ = bits 23 down to 22 of $\mathsf{A}_{IMEM}[\mathsf{R}_{PC}]$
rd = bits 21 down to 20 of $\mathsf{A}_{IMEM}[\mathsf{R}_{PC}]$
$r1$ = bits 19 down to 18 of $\mathsf{A}_{IMEM}[\mathsf{R}_{PC}]$
$r2$ = bits 17 down to 16 of $\mathsf{A}_{IMEM}[\mathsf{R}_{PC}]$
$const$ = bits 15 down to 0 of $\mathsf{A}_{IMEM}[\mathsf{R}_{PC}]$

π's and α's of the transitions corresponding to the execution of Loadi, Add and Bz (when taken and not taken) are:

$$\pi_{Loadi} = (opcode{=}{=}0)$$
$$\alpha_{Loadi} = < \mathit{set}(\mathsf{R}_{PC}{+}1),\ \mathit{a\text{-}set}(rd,\ const),\ \epsilon,\ \epsilon >$$

$$\pi_{Add} = (opcode{=}{=}2)$$
$$\alpha_{Add} = < \mathit{set}(\mathsf{R}_{PC}{+}1),\ \mathit{a\text{-}set}(rd,\ \mathsf{A}_{RF}[r1]{+}\mathsf{A}_{RF}[r2]),\ \epsilon,\ \epsilon >$$

$$\pi_{BzTaken} = (opcode{=}{=}3) \wedge (\mathsf{A}_{RF}[r1]{=}{=}0)$$
$$\alpha_{BzTaken} = < \mathit{set}(\mathsf{A}_{RF}[r2]),\ \epsilon,\ \epsilon,\ \epsilon >$$

$$\pi_{BzNotTaken} = (opcode{=}{=}3) \wedge (\mathsf{A}_{RF}[r1]{\neq}0)$$
$$\alpha_{BzNotTaken} = < \mathit{set}(\mathsf{R}_{PC}{+}1),\ \epsilon,\ \epsilon,\ \epsilon >$$

Lastly, to zero R_{PC} during initialization, $\mathcal{S}^o{=}< 0 >$. □

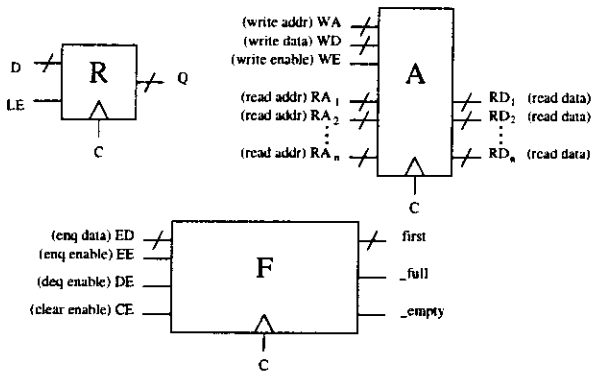


Figure 2: Synchronous State Elements

3 Synchronous Implementation of an ATS

One possible synchronous implementation of an ATS performs one atomic update step per clock period. The elements of S are implemented using clock synchronous registers, arrays, and FIFO's. These library elements are shown with their interfaces in Figure 2. (This discussion assumes that an array always has a sufficient number of read ports to support all ***a-get***() queries. After an RTL is generated, the actual number of combinational read ports can be reduced by common subexpression elimination.) During a clock period, the π expressions and the argument expressions in the α's are evaluated combinationally using the current state of the state elements. In every clock cycle, a scheduler selects one of the transitions whose π expression is asserted. At the rising clock edge at the end of a clock period, all state elements are updated synchronously according to the actions of the selected transition. Details of the circuit implementation are given below.

3.1 Reference Implementation

Scheduler: Based on $\pi_{T_1}(s),...,\pi_{T_M}(s)$ where $\pi_{T_i}(s)$ is the value of T_i's π expression in state s, a scheduler generates *arbitrated* transition enable signals $\phi_{T_1} \vee ... \vee \phi_{T_M}$ where ϕ_{T_i} is used to select the actions of T_i. Any valid scheduler must, at least, ensure that in any s,

1. $\phi_{T_i} \Rightarrow \pi_{T_i}(s)$
2. $\pi_{T_1}(s) \vee ... \vee \pi_{T_M}(s) \Rightarrow \phi_{T_1} \vee ... \vee \phi_{T_M}$

A priority encoder is a valid scheduler that selects one applicable transition per clock cycle. Since ATS allows non-determinism in selection, the priority encoder could use static, round-robin or randomized priority.

Register Update Logic: Each R in $\mathcal{S}$ can be implemented using a synchronous register with clock enable. For each R in $\mathcal{S}$, we first find the set of transitions that update R,

$$\{T_{x_i} \mid \mathbf{a}^R_{T_{x_i}} = \textit{set}(exp_{x_i})\}$$

where $\mathbf{a}^R_{T_{x_i}}$ is α_{T_i}'s action on R. The register's latch enable signal is

$$\mathsf{LE} = \phi_{T_{x_1}} \vee ... \vee \phi_{T_{x_n}}$$

The register's data input signal is

$$\mathsf{D} = \phi_{T_{x_1}} \cdot exp_{x_1} + ... + \phi_{T_{x_n}} \cdot exp_{x_n}$$

This expression corresponds to a pass-gate multiplexer where exp_{x_i} is enabled by $\phi_{T_{x_i}}$.

Array Update Logic: Each A in $\mathcal{S}$ can be implemented using a memory array with a synchronous write port. (Given an array implementation with sufficient independent write ports, this scheme can also be generalized to support ATS that allows multiple array writes in an atomic step.) For each A in $\mathcal{S}$, we first find the set of transitions that write A,

$$\{T_{x_i} \mid \mathbf{a}^A_{T_{x_i}} = \textit{a-set}(idx_{x_i}, data_{x_i})\}.$$

The array's write address and data are

$$\mathsf{WA} = \phi_{T_{x_1}} \cdot idx_{x_1} + ... + \phi_{T_{x_n}} \cdot idx_{x_n}$$

$$\mathsf{WD} = \phi_{T_{x_1}} \cdot data_{x_1} + ... + \phi_{T_{x_n}} \cdot data_{x_n}$$

and the array's write enable signal is

$$\mathsf{WE} = \phi_{T_{x_1}} \vee ... \vee \phi_{T_{x_n}}$$

FIFO Update Logic: Each F in S can be implemented using a first-in-first-out queue with synchronous enqueue, dequeue and clear interfaces. For each F, the inputs to the interfaces can be constructed as follows:

Enqueue Interface: We first find the set of transitions that enqueues a new value into F

$$\{T_{x_i} \mid (\mathbf{a}^F_{T_{x_i}} = \textbf{\textit{enq}}(exp_{x_i})) \vee (\mathbf{a}^F_{T_{x_i}} = \textbf{\textit{en-deq}}(exp_{x_i}))\}$$

Every transition T_{x_i} that enqueues into F is required to test F.***notfull***() in its π expression. Hence, no $\phi_{T_{x_i}}$ will be asserted if the FIFO is already full. The FIFO's enqueue data and enable signals are

$$\mathsf{ED} = \phi_{T_{x_1}} \cdot exp_{x_1} + ... + \phi_{T_{x_n}} \cdot exp_{x_n}$$

$$\mathsf{EE} = \phi_{T_{x_1}} \vee ... \vee \phi_{T_{x_n}}$$

Dequeue Interface: We find the set of transitions that dequeues from F

$$\{T_{x_i} \mid (\mathbf{a}^F_{T_{x_i}} = \textbf{\textit{deq}}()) \vee (\mathbf{a}^F_{T_{x_i}} = \textbf{\textit{en-deq}}(exp_{x_i}))\}$$

Every transition T_{x_i} which dequeues from F is required to test F.***notempty***() in its π expression. Similarly, no $\phi_{T_{x_i}}$ will be asserted if the FIFO is empty. The FIFO's dequeue enable signal is

$$\mathsf{DE} = \phi_{T_{x_1}} \vee ... \vee \phi_{T_{x_n}}$$

Clear Interface: We find the set of transitions that clears the contents of F

$$\{T_{x_i} \mid \mathbf{a}^F_{T_{x_i}} = \textbf{\textit{clear}}()\}$$

The FIFO's clear enable is

$$\mathsf{CE} = \phi_{T_{x_1}} \vee ... \vee \phi_{T_{x_n}}$$

3.2 Correctness of a Synchronous Implementation

An implementation is said to implement an ATS correctly if

1. The implementation's sequence of state transitions corresponds to some execution of the ATS.
2. The implementation maintains the liveness of state transitions.

Unless the priority encoder in the reference implementation has true randomization, the reference implementation is deterministic. In other words, the implementation could only embody one of behaviors allowed by the ATS. The implementation guarantees the liveness of the system in that a transition is taken on every clock cycle if one is available. The implementation could not guarantee strong-fairness in the selection of transitions to prevent life lock. However, a round-robin priority encoder is sufficient to ensure weak-fairness, that is, if a transition stays applicable, it will be selected in a finite number of steps.

3.3 Performance Considerations

In a given atomic update step, if two simultaneously applicable transitions read and write mutually disjoint parts of $\mathcal{S}$, then the two transitions could be executed in any order in two successive steps to produce the same final state. In this scenario, although the semantics of an ATS requires an execution in sequential and atomic update steps, a hardware implementation can exploit the underlying parallelism and execute the two transitions concurrently in one clock cycle. In general, it is not safe to allow two arbitrary applicable transitions to execute in the same clock cycle because of possible data dependence and structural conflicts. The next two sections formalize the conditions for concurrent execution of transitions and suggest more aggressive schedules that execute multiple transitions in the same clock cycle. In a multiple transitions per cycle implementation, each implementation state transition must correspond to a sequential execution of the ATS transitions in some order.

4 The Scheduling of Conflict-Free Transitions

If two transitions T_a and T_b become applicable in the same clock cycle when $\mathcal{S}$ is in state s, for an implementation to correctly select both transitions for

execution, $\pi_{T_a}(\delta_{T_b}(s))$ or $\pi_{T_b}(\delta_{T_a}(s))$ must be true. Otherwise, executing both transitions would be inconsistent with any sequential execution in two atomic update steps.

Given $\pi_{T_a}(\delta_{T_b}(s))$ or $\pi_{T_b}(\delta_{T_a}(s))$, there are two approaches to compose the actions of T_a and T_b in the same clock cycle. The first approach cascades the combinational logic from the two transitions. However, arbitrary cascading does not always improve circuit performance since it may lead to a longer cycle time. In a more practical approach, T_a and T_b are allowed to execute in the same clock cycle only if the correct final state can be constructed from an independent evaluation of their combinational logic on the same starting state.

This section develops a scheduling algorithm based on the *conflict-free* relationship ($<>_{CF}$). $<>_{CF}$ is a symmetrical relationship that imposes a stronger requirement than necessary for executing two transitions concurrently. However, the symmetry of $<>_{CF}$ permits a straight-forward implementation that concurrently executes multiple transitions if they are pairwise $<>_{CF}$. The next section will present an improvement based on a more exact condition.

4.1 Conflict-Free Transitions

The conflict-free relationship and the parallel composition function $\boldsymbol{PC}$ are defined in Definition 1 and Definition 2.

Definition 1 (Conflict-Free Relationship)

Two transitions T_a and T_b are said to be conflict-free ($T_a <>_{CF} T_b$) if

$$\begin{aligned}\forall s.\ \pi_a(s) \wedge \pi_b(s) \Rightarrow \pi_b(\delta_{T_a}(s)) \wedge \pi_a(\delta_{T_b}(s)) \wedge \\ (\delta_{T_b}(\delta_{T_a}(s)) == \delta_{T_a}(\delta_{T_b}(s)) \\ == \delta_{PC}(s))\end{aligned}$$

where δ_{PC} is the functional equivalent of $\boldsymbol{PC}(\alpha_{T_a},\alpha_{T_b})$.

Definition 2 (Parallel Composition)

$$\begin{aligned}\boldsymbol{PC}(\alpha_a,\alpha_b) = <\boldsymbol{pc}_R(\mathbf{a}^{R_1},\mathbf{b}^{R_1})..., \boldsymbol{pc}_A(\mathbf{a}^{A_1},\mathbf{b}^{A_1})..., \\ \boldsymbol{pc}_F(\mathbf{a}^{F_1},\mathbf{b}^{F_1})...>\end{aligned}$$

where α_a=<$\mathbf{a}^{R_1}$..., $\mathbf{a}^{A_1}$..., $\mathbf{a}^{F_1}$...>, α_b=<$\mathbf{b}^{R_1}$..., $\mathbf{b}^{A_1}$..., $\mathbf{b}^{F_1}$...>

pc_R(**a**,**b**)=*case* **a**,**b** *of* **a**,ϵ $\Rightarrow$ **a**
ϵ,**b** $\Rightarrow$ **b**
... $\Rightarrow$ *undefined*
pc_A(**a**,**b**)=*case* **a**,**b** *of* **a**,ϵ $\Rightarrow$ **a**
ϵ,**b** $\Rightarrow$ **b**
... $\Rightarrow$ *undefined*
pc_F(**a**,**b**)=*case* **a**,**b** *of* **a**,ϵ $\Rightarrow$ **a**
ϵ,**b** $\Rightarrow$ **b**
enq(*exp*),***deq***() $\Rightarrow$ ***en-deq***(*exp*)
deq(),***enq***(*exp*) $\Rightarrow$ ***en-deq***(*exp*)
... $\Rightarrow$ *undefined*

The function ***PC*** computes a new α by composing two α's that do not contain conflicting actions on the same state element. It can be shown that ***PC*** is commutative and associative.

Suppose T_a and T_b become applicable in the same state ***s***. $T_a <>_{CF} T_b$ implies that the two transitions can be applied in either order in two successive steps to produce the same final state ***s***'. $T_a <>_{CF} T_b$ further implies that an implementation could produce ***s***' by applying the parallel composition of α_{T_a} and α_{T_b} to the same initial state ***s***. Theorem 1 extends this result to multiple pairwise $<>_{CF}$ transitions.

Theorem 1 (Composition of $<>_{CF}$ Transitions)

Given a collection of n transitions applicable in state ***s***, if all n transitions are pairwise conflict-free, then the following holds for any ordering T_{x_1},...,T_{x_n},

$$\pi_{T_{x_2}}(\delta_{T_{x_1}}(s)) \wedge ... \wedge \pi_{T_{x_n}}(\delta_{T_{x_{n-1}}}(... \ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s))) ...)) \wedge$$
$$(\delta_{T_{x_n}}(\delta_{T_{x_{n-1}}}(... \ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s))) ...)) == \delta_{PC}(s))$$

where δ_{PC} is the functional equivalent of the parallel compositions of $\alpha_{T_{x_1}}$,...,$\alpha_{T_{x_n}}$, in any order.

4.2 Conservative Static Deduction of $<>_{CF}$

Our synthesis algorithm can work with a conservative $<>_{CF}$ test, that is, if the conservative test fails to identity a pair of transitions as $<>_{CF}$, the algorithm would generate a less optimized but correct implementation.

4.2.1 Analysis based on Domain and Range

A static determination of $<>_{CF}$ can be made by comparing domains and ranges of transitions. The domain of a transition is the set of state elements in $\mathcal{S}$ "read" by the expressions in either π or α. The domain of a transition could be further classified as π-domain and α-domain. The range of a transition is the set of state elements in $\mathcal{S}$ that are acted on by α. For the purpose of analysis, the head and the tail of a FIFO are considered to be separate elements. The functions to extract the domain and range of a transition are defined below.

Definition 3 (Domain of π and α)

$$
\begin{aligned}
\boldsymbol{D}_e(exp) = \textit{case } exp \textit{ of}\\
&\textit{constant} \Rightarrow \{\}\\
&\mathsf{R}.\boldsymbol{get}() \Rightarrow \{\mathsf{R}\}\\
&\mathsf{A}.\boldsymbol{a\text{-}get}(idx) \Rightarrow \{\mathsf{A}\} \cup \boldsymbol{D}_e(idx)\\
&\mathsf{F}.\boldsymbol{first}() \Rightarrow \{\mathsf{F}_{head}\}\\
&\mathsf{F}.\boldsymbol{notfull}() \Rightarrow \{\mathsf{F}_{tail}\}\\
&\mathsf{F}.\boldsymbol{notempty}() \Rightarrow \{\mathsf{F}_{head}\}\\
&Op(exp_1,...,exp_n) \Rightarrow \boldsymbol{D}_e(exp_1) \cup ... \cup \boldsymbol{D}_e(exp_n)
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{D}_\alpha(\alpha) &= \boldsymbol{D}_R(\mathbf{a}^{R_1}) \cup ... \cup \boldsymbol{D}_A(\mathbf{a}^{A_1}) \cup ... \cup \boldsymbol{D}_F(\mathbf{a}^{F_1}) \cup ...\\
&\quad \textit{where } \alpha = <\mathbf{a}^{R_1}..., \mathbf{a}^{A_1}..., \mathbf{a}^{F_1}...>
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{D}_R(\mathbf{a}) = \textit{case } \mathbf{a} \textit{ of } \epsilon &\Rightarrow \{\}\\
\boldsymbol{set}(exp) &\Rightarrow \boldsymbol{D}_e(exp)
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{D}_A(\mathbf{a}) = \textit{case } \mathbf{a} \textit{ of } \epsilon &\Rightarrow \{\}\\
\boldsymbol{a\text{-}set}(idx,data) &\Rightarrow \boldsymbol{D}_e(idx) \cup \boldsymbol{D}_e(data)
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{D}_F(\mathbf{a}) = \textit{case } \mathbf{a} \textit{ of } \epsilon &\Rightarrow \{\}\\
\boldsymbol{enq}(exp) &\Rightarrow \boldsymbol{D}_e(exp)\\
\boldsymbol{en\text{-}deq}(exp) &\Rightarrow \boldsymbol{D}_e(exp)\\
\boldsymbol{deq}() &\Rightarrow \{\}\\
\boldsymbol{clear}() &\Rightarrow \{\}
\end{aligned}
$$

Definition 4 (Range of α)

$$
\begin{aligned}
\boldsymbol{R}_\alpha(\alpha) &= \boldsymbol{R}_R(\mathbf{a}^{R_1}) \cup ... \cup \boldsymbol{R}_A(\mathbf{a}^{A_1}) \cup ... \cup \boldsymbol{R}_F(\mathbf{a}^{F_1}) \cup ...\\
&\quad \textit{where } \alpha = <\mathbf{a}^{R_1}..., \mathbf{a}^{A_1}..., \mathbf{a}^{F_1}...>
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{R}_R(\mathbf{a}^R) = \mathit{case}\ \mathbf{a}^R\ \mathit{of}\ \epsilon &\Rightarrow \{\} \\
\mathit{set}(\mathit{exp}) &\Rightarrow \{\mathsf{R}\} \\
\boldsymbol{R}_A(\mathbf{a}^A) = \mathit{case}\ \mathbf{a}^A\ \mathit{of}\ \epsilon &\Rightarrow \{\} \\
\mathit{a\text{-}set}(\text{-},\text{-}) &\Rightarrow \{\mathsf{A}\} \\
\boldsymbol{R}_F(\mathbf{a}^F) = \mathit{case}\ \mathbf{a}^F\ \mathit{of}\ \epsilon &\Rightarrow \{\} \\
\mathit{enq}(\text{-}) &\Rightarrow \{\mathsf{F}_{tail}\} \\
\mathit{deq}() &\Rightarrow \{\mathsf{F}_{head}\} \\
\mathit{en\text{-}deq}(\text{-}) &\Rightarrow \{\mathsf{F}_{head},\mathsf{F}_{tail}\} \\
\mathit{clear}() &\Rightarrow \{\mathsf{F}_{head},\mathsf{F}_{tail}\}
\end{aligned}
$$

Using $\boldsymbol{D}()$ and $\boldsymbol{R}()$, a sufficient condition that ensures two transitions are $<>_{CF}$ is given in Theorem 2.

Theorem 2 (Sufficient Condition for $<>_{CF}$)

Given T_a and T_b,
$((\boldsymbol{D}(\pi_{T_a}) \cup \boldsymbol{D}(\alpha_{T_a})) \not\cap \boldsymbol{R}(\alpha_{T_b})) \wedge$
$((\boldsymbol{D}(\pi_{T_b}) \cup \boldsymbol{D}(\alpha_{T_b})) \not\cap \boldsymbol{R}(\alpha_{T_a})) \wedge$
$(\boldsymbol{R}(\alpha_{T_a}) \not\cap \boldsymbol{R}(\alpha_{T_b}))$
$\Rightarrow (T_a <>_{CF} T_b)$
If the domain and range of two transitions do not overlap, then the two transitions have no data dependence. Since their range do not overlap, a valid parallel composition of α_{T_a} and α_{T_b} must exit.

4.2.2 Analysis based on Mutual Exclusion

If two transitions can never become applicable on the same state, then they are said to be mutually exclusive.

Definition 5 (Mutually Exclusive Relationship)

$T_a <>_{ME} T_b$ *if* $\forall\ s.\ \neg(\pi_{T_a}(s) \wedge \pi_{T_b}(s))$

Two transitions that are $<>_{ME}$ satisfy the definition of $<>_{CF}$ by default. Testing $<>_{ME}$ requires determining the satisfiability of the expression $(\pi_{T_a}(s) \wedge \pi_{T_b}(s))$. Fortunately, in practice, the π expression is usually a conjunction of relational constraints on the values of state elements. A conservative test that searches two π expressions for contradicting constraints on any one state element works well in practice.

4.3 Scheduling of $<>_{CF}$ Transitions

Using Theorem 1, instead of selecting a single transition per clock cycle, a scheduler could select a number of applicable transitions that are pairwise conflict-free. In other words, in each clock cycle, the ϕ's should satisfy the condition,

$$\phi_{T_a} \wedge \phi_{T_b} \Rightarrow T_a <>_{CF} T_b$$

where ϕ_T is the arbitrated transition enable signal for transition T. Given a set of applicable transitions in a clock cycle, many different subsets of pairwise conflict-free transitions could exist. Selecting the optimum subset requires evaluating the relative importance of the transitions. Alternatively, an objective metric simply optimizes the number of transitions executed in each clock cycle.

Partitioned Scheduler: In a partitioned scheduler, transitions in $\mathcal{T}$ are first partitioned into as many disjoint *scheduling groups*, $\mathcal{P}_1$,...,$\mathcal{P}_k$, as possible such that

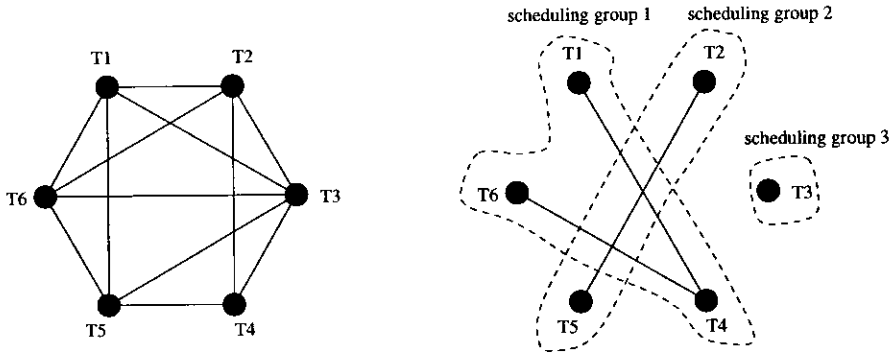
$$(T_a \in \mathcal{P}_a) \wedge (T_b \in \mathcal{P}_b) \Rightarrow T_a <>_{CF} T_b$$

Transitions in different scheduling groups are conflict-free, and hence each scheduling group can be scheduled independently of the other groups. For a given scheduling group containing T_{x_1},...,T_{x_n}, $\phi_{T_{x_1}}$,...,$\phi_{T_{x_n}}$ can be generated from $\pi_{T_{x_1}}(s)$,...,$\pi_{T_{x_n}}(s)$ using a priority encoder. In the best case, a transition T is conflict-free with every other transition in $\mathcal{T}$. T is in a scheduling group by itself, and ϕ_T=π_T without arbitration.

$\mathcal{T}$ can be partitioned into scheduling groups by finding the *connected components* of an undirected graph whose nodes are transitions T_1,...,T_M, and whose edges are $\{(T_i,T_j) \mid \neg(T_i <>_{CF} T_j)\}$. Each connected component is a scheduling group.

Example: Partitioned Scheduler

The undirected graph in Figure 3.a depicts the $<>_{CF}$ relationships in an ATS with six transitions, $\mathcal{T}$=$\{T_1, T_2, T_3, T_4, T_5, T_6\}$. Two nodes that are connected by an edge are known to be conflict-free, i.e.



(a) Conflict-Free Graph (b) Conflict Graph

Figure 3: (a) A conflict-free graph (b) Corresponding conflict graph and its connected components

$(T_1 <>_{CF} T_2)$, $(T_1 <>_{CF} T_3)$, $(T_1 <>_{CF} T_5)$, $(T_1 <>_{CF} T_6)$,
$(T_2 <>_{CF} T_3)$, $(T_2 <>_{CF} T_4)$, $(T_2 <>_{CF} T_6)$,
$(T_3 <>_{CF} T_4)$, $(T_3 <>_{CF} T_5)$, $(T_3 <>_{CF} T_6)$,
$(T_4 <>_{CF} T_5)$,
$(T_5 <>_{CF} T_6)$

Figure 3.b gives the corresponding conflict graph where two nodes are connected if they may not be $<>_{CF}$. The absence of an edge between two nodes T_i and T_j implies $T_i <>_{CF} T_j$. The conflict graph has three connected components, corresponding to the three $<>_{CF}$ scheduling groups. The ϕ signals corresponding to T_1, T_4 and T_6 can be generated using a priority encoding of their corresponding π's. Scheduling group 2 also requires a scheduler to ensure ϕ_2 and ϕ_5 are not asserted in the same clock cycle. However, $\phi_{T_3}=\pi_{T_3}$ without any arbitration. □

Enumerated Scheduler: Scheduling group 1 in the previous example contains three transitions $\{T_1,T_4,T_6\}$ such that $T_1 <>_{CF} T_6$ but both T_1 and T_6 are not $<>_{CF}$ with T_4. Although the three transitions cannot be scheduled independently of each other, T_1 and T_6 could be selected together as long as T_4 is not also selected in the same clock cycle. This selection is valid because T_1 and T_6 are $<>_{CF}$ between themselves and with ev-

ery transition selected by the other groups. In general, the scheduler for each group could independently select multiple transitions that are pairwise $<>_{CF}$ within the scheduling group.

For a scheduling group with transitions $T_{x_1},...,T_{x_n}$, $\phi_{T_{x_1}},...,\phi_{T_{x_n}}$ can be computed by a $2^n \times n$ lookup table indexed by $\pi_{T_{x_1}}(s),...,\pi_{T_{x_n}}(s)$. The data value $d_1,...,d_n$ at the table entry with index $b_1,...,b_n$ can be determined by finding the *maximum clique* of an undirected graph whose nodes $\mathcal{N}$ and edges $\mathcal{E}$ are defined as follows,

$$\begin{aligned} \mathcal{N} &= \{T_{x_i} \mid b_i \text{ is asserted}\} \\ \mathcal{E} &= \{(T_{x_i},T_{x_j}) \mid (T_{x_i} \in \mathcal{N}) \wedge (T_{x_j} \in \mathcal{N}) \wedge \\ &\qquad (T_{x_i} <>_{CF} T_{x_j})\} \end{aligned}$$

For each T_{x_i} that is in the maximum clique, assert d_i.

Example: Enumerated Encoder

Instead of a priority encoder, the scheduling group 1 from the partitioned scheduler example above can use an enumerated encoder that allows T_1 and T_6 to execute concurrently.

π_{T_1}	π_{T_4}	π_{T_6}	ϕ_{T_1}	ϕ_{T_4}	ϕ_{T_6}
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	0	0
1	1	1	1	0	1

□

4.4 Performance Gain

When $\mathcal{T}$ can be partitioned into scheduling groups, the partitioned scheduler is smaller and faster than the monolithic encoder used in the reference implementation in Section 3.1. The partitioned scheduler also reduces wiring cost and delay since π's and ϕ's of unrelated transitions do not need to be brought together for arbitration.

The property of the parallel composition function, $\boldsymbol{PC}$, ensures that transitions are $<>_{CF}$ only if their actions on state elements do not conflict. For example, given a set of transitions that are all pairwise $<>_{CF}$, each R in $\mathcal{S}$ can be updated by at most one of those transitions. Hence, the state update logic described in Section 3.1, which assumes single-transition-per-cycle, can be used without modification. Consequently, parallel composition of actions does not increase the combinational delay of the datapath. All in all, the implementation in this section achieves better performance than the reference implementation by allowing more transitions to execute in a clock cycle without increasing the clock period.

5 Sequential Composition of Transitions

Consider the following example, where $\boldsymbol{PC}(\alpha_{T_a},\alpha_{T_b})$ and its functional equivalent, δ_{PC}, is well-defined for any two transitions T_a and T_b in the ATS,

$$\begin{array}{rcl} \mathcal{S} & = & <\mathsf{R}_1,\mathsf{R}_2,\mathsf{R}_3> \\ \mathcal{T} & = & \{T_1,T_2,T_3\} \\ T_1 & = & <\mathit{true},\ <\boldsymbol{set}(\mathsf{R}_2{+}1),\epsilon,\epsilon>> \\ T_2 & = & <\mathit{true},\ <\epsilon,\ \boldsymbol{set}(\mathsf{R}_3{+}1),\epsilon>> \\ T_3 & = & <\mathit{true},\ <\epsilon,\ \epsilon,\ \boldsymbol{set}(\mathsf{R}_1{+}1)>> \end{array}$$

Although all transitions are always applicable, the previous section's implementation would not permit T_a and T_b to executed in the same clock cycle because, in general, $\delta_{T_a}(\delta_{T_b}(s)) \neq \delta_{T_b}(\delta_{T_a}(s))$. However, it can be shown that for all s, $\delta_{PC}(s)$ is consistent with some sequential order of execution of T_a and T_b. Hence, their concurrent execution is allowed in a correct implementation. One should also note that concurrent execution of all three transitions in a parallel composition does not always produce a consistent new state due to circular data dependencies between the three transitions. To capture these conditions, this section presents a more exact requirement for concurrent execution based on the sequential composibility relationship.

5.1 Sequentially-Composible Transitions

Definition 6 (Sequential Composibility)

Two transitions T_a and T_b are said to be sequentially composible ($T_a <_{SC} T_b$), if

$$\forall s.\ \pi_{T_a}(s) \wedge \pi_{T_b}(s) \Rightarrow \pi_{T_b}(\delta_{T_a}(s)) \wedge (\delta_{T_b}(\delta_{T_a}(s)) == \delta_{SC}(s))$$

where δ_{SC} is the functional equivalent of $\boldsymbol{SC}(\alpha_{T_a},\alpha_{T_b})$.

Definition 7 (Sequential Composition)

$$\boldsymbol{SC}(\alpha_a,\alpha_b) = <\boldsymbol{sc}_R(\mathbf{a}^{R_1},\mathbf{b}^{R_1})..., \boldsymbol{sc}_A(\mathbf{a}^{A_1},\mathbf{b}^{A_1})..., \boldsymbol{sc}_F(\mathbf{a}^{F_1},\mathbf{b}^{F_1})...>$$

where $\alpha_a = <\mathbf{a}^{R_1}..., \mathbf{a}^{A_1}..., \mathbf{a}^{F_1}...>$, $\alpha_b = <\mathbf{b}^{R_1}..., \mathbf{b}^{A_1}..., \mathbf{b}^{F_1}...>$

The sequential composition function $\boldsymbol{SC}$ returns a new α by composing actions on the same element from two α's. $\boldsymbol{sc}_R$, $\boldsymbol{sc}_A$ and $\boldsymbol{sc}_F$ are the same as $\boldsymbol{pc}_R$, $\boldsymbol{pc}_A$ and $\boldsymbol{pc}_F$ except in two cases where $\boldsymbol{SC}$ allows two conflicting actions to be sequentialized. First, $\boldsymbol{sc}_R(\boldsymbol{set}(exp_a),\boldsymbol{set}(exp_b))$ is $\boldsymbol{set}(exp_b)$ since the effect of the first action is overwritten by the second in a sequential application. Second, $\boldsymbol{sc}_F(\mathbf{a},\boldsymbol{clear}())$ returns $\boldsymbol{clear}()$ since regardless of $\mathbf{a}$, applying $\boldsymbol{clear}()$ leaves the FIFO emptied.

$<_{SC}$ is a relaxation of $<>_{CF}$. In particular, $<_{SC}$ is not symmetric. Suppose T_a and T_b are applicable in state s, $T_a <_{SC} T_b$ only requires the concurrent execution of T_a and T_b on s to correspond to $\delta_{T_b}(\delta_{T_a}(s))$, but not $\delta_{T_a}(\delta_{T_b}(s))$. Two $<_{SC}$ transitions can also have conflicting actions that can be sequentialized. Theorem 3 extends this result to multiple transitions whose transitive closure on $<_{SC}$ is ordered.

Theorem 3 (Composition of $<_{SC}$ Transitions)

Given a sequence of n transitions, $T_{x_1},...,T_{x_n}$, applicable in state s, if $T_{x_j} <_{SC} T_{x_k}$ for all $j < k$ then,

$$\pi_{T_{x_2}}(\delta_{T_{x_1}}(s)) \wedge ... \wedge \pi_{T_{x_n}}(\delta_{T_{x_{n-1}}}(... \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s))) ...)) \wedge (\delta_{T_{x_n}}(\delta_{T_{x_{n-1}}}(... \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s))) ...)) == \delta_{SC}(s))$$

where δ_{SC} is the functional equivalent of the nested sequential compositions of $\boldsymbol{SC}(...\boldsymbol{SC}(\boldsymbol{SC}(\alpha_{T_{x_1}},\alpha_{T_{x_2}}),\alpha_{T_{x_3}}),...)$

5.2 Conservative Static Deduction of $<_{SC}$

Using $\boldsymbol{D}()$ and $\boldsymbol{R}()$, a sufficient condition for two transitions to be $<_{SC}$ is given in Theorem 4.

Theorem 4 (Sufficient Condition for $<_{SC}$)

Given T_a and T_b,
$((\boldsymbol{D}(\pi_{T_b}) \cup \boldsymbol{D}(\alpha_{T_b})) \not\cap \boldsymbol{R}(\alpha_{T_a})) \wedge (\boldsymbol{SC}(\alpha_{T_a}, \alpha_{T_b})$ *is defined*$)$
$\Rightarrow T_a <_{SC} T_b$

5.3 Scheduling of $<_{SC}$ Transitions

Incorporating the results from Theorem 3 into the partitioned scheduler from Section 4.3 allows additional transitions to execute in the same clock cycle. For each conflict-free scheduling group containing $T_{x_1},...,T_{x_n}$, its scheduler generates arbitrated transition enable signals $\phi_{T_{x_1}},...,\phi_{T_{x_n}}$. In every s, there must be an ordering of all asserted ϕ's, $\phi_{T_{y_1}},...,\phi_{T_{y_m}}$, such that $T_{y_j} <_{SC} T_{y_k}$ if $j < k$. However, in order to simplify the state update logic, our algorithm further requires a static *SC-ordering* $T_{z_1},...,T_{z_n}$ for each scheduling group such that

$$\forall\ s.\ \phi_{T_{z_j}} \wedge \phi_{T_{z_k}} \Rightarrow T_{z_j} <_{SC} T_{y_k} \text{ if } j < k.$$

Scheduler:
To construct the $<_{SC}$ scheduler for a conflict-free scheduling group that contains $T_{x_1},...,T_{x_n}$, we first compute the group's SC-ordering using a topological sort on a directed graph whose nodes are $T_{x_1},...,T_{x_n}$ and whose edges $\mathcal{E}_{SC_{acyclic}}$ is the largest subset of $\mathcal{E}_{SC}$ such that this graph is acyclic. $\mathcal{E}_{SC}$ is defined as,

$$\{(T_{x_i},T_{x_j}) \mid (T_{x_i} <_{SC} T_{x_j}) \wedge \neg(T_{x_i} <>_{CF} T_{x_j})\}$$

Next, we find connected components of an undirected graph whose nodes are $T_{x_1},...,T_{x_n}$ and whose edges are

$$\{(T_{x_i},T_{x_j}) \mid ((T_{x_i},T_{x_j}) \notin \mathcal{E}_{SC_{acyclic}}) \wedge ((T_{x_j},T_{x_i}) \notin \mathcal{E}_{SC_{acyclic}}) \wedge \neg(T_{x_i} <>_{CF} T_{x_j})\}$$

Each connected component forms a scheduling subgroup. Transitions in different scheduling subgroups are either $<>_{CF}$ or $<_{SC}$. ϕ's for the transitions in a subgroup can be generated using a priority encoder. On each clock cycle, $T_{y_1},...,T_{y_m}$ selected by the encoders of a scheduling group satisfy the

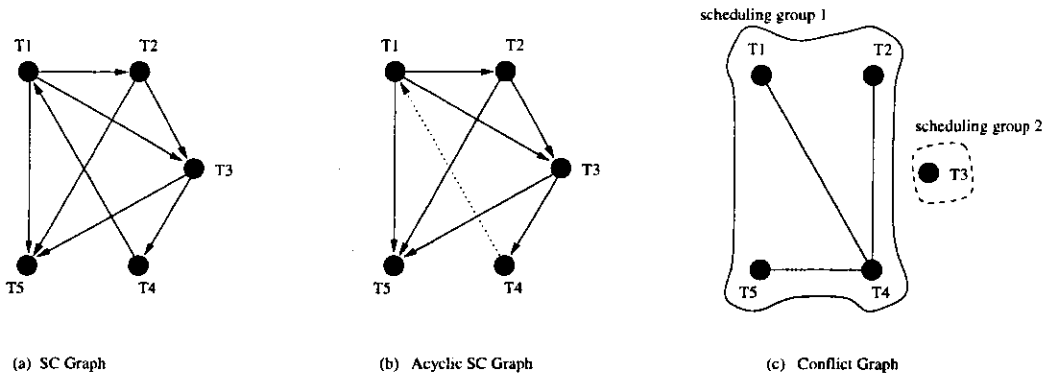


Figure 4: (a) A directed SC graph (b) Corresponding acyclic directed SC graph, and (c) Corresponding conflict graph and its connected components

conditions of Theorem 3 because $T_{y_j} <_{SC} T_{y_k}$ if T_{y_j} comes before T_{y_k} in the *SC-ordering* of the parent scheduling group.

Example:

The directed graph in Figure 4.a depicts the $<_{SC}$ relationships in an ATS with five transitions, $\mathcal{T}=\{T_1,T_2,T_3,T_4,T_5\}$. A directed edge from T_a to T_b implies $T_a <_{SC} T_b$, i.e.

$(T_1 <_{SC} T_2)$, $(T_1 <_{SC} T_3)$, $(T_1 <_{SC} T_5)$,
$(T_2 <_{SC} T_3)$, $(T_2 <_{SC} T_5)$,
$(T_3 <_{SC} T_4)$, $(T_3 <_{SC} T_5)$,
$(T_4 <_{SC} T_1)$

Figure 4.b shows the acyclic SC graph made by removing the edge from T_4 to T_1. A topological sort of the acyclic SC graph yields the SC-ordering of T_1, T_2, T_3, T_4 and T_5. (The order of T_4 and T_5 can be reversed also.) Figure 4.c gives the corresponding conflict graph. The two connected components of the conflict graph are the two scheduling groups. $\phi_{T_3}=\pi_{T_3}$ without any arbitration. The remaining ϕ signals can be generated using a priority encoding of their corresponding π's. More transitions can be executed concurrently if the following enumerated encoder is used instead.

π_{T_1}	π_{T_2}	π_{T_4}	π_{T_5}	ϕ_{T_1}	ϕ_{T_2}	ϕ_{T_4}	ϕ_{T_5}
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	1
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	1
1	1	0	0	1	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	1	1	0	0
1	1	1	1	1	1	0	1

□

State Update Logic with Sequential Composition:
When sequentially-composible transitions are allowed in the same clock cycle, the register update logic cannot assume only one transition acts on a register in each clock cycle. When multiple actions are enabled for a register, the register update logic should ignore all but the latest action with respect to the SC-ordering of some scheduling group. (All transitions that update the same register are in the same scheduling group, except for a transition that is mutually exclusive with the rest.) For each R in $\mathcal{S}$, we find the set of transitions that update R,

$$\{T_{x_i} \mid \mathbf{a}^{R}_{T_{x_i}} = \textit{set}(exp_{x_i})\}$$

The register's latch enable signal is

$$\mathsf{LE} = \phi_{T_{x_1}} \vee ... \vee \phi_{T_{x_n}}$$

The register's data input signal is

$$\mathsf{D} = \phi_{T_{x_1}} \cdot \psi_{T_{x_1}} \cdot exp_{x_1} + \ldots + \phi_{T_{x_n}} \cdot \psi_{T_{x_n}} \cdot exp_{x_n}$$

where $\psi_{T_{x_i}} = \neg(\phi_{T_{y_1}} \vee \phi_{T_{y_2}} \vee \ldots)$. The expression $\psi_{T_{x_i}}$ contains $\phi_{T_{y_i}}$'s from the set of transitions

$$\{ T_{y_i} \mid \mathsf{R} \in \boldsymbol{R}(\alpha_{T_{y_i}}) \wedge \\ T_{x_i} \text{ comes before } T_{y_i} \text{ in the } \textit{SC-ordering} \wedge \\ \neg(T_{x_i} <>_{ME} T_{y_i})\}$$

In essence, the register's data input (D) is selected through a prioritized multiplexer that gives higher priority to transitions later in the SC-ordering. The update logic for arrays and FIFO's are unchanged from Section 3.1.

6 Related Work and Conclusion

The usage of the term *high-level* and *behavioral* description has been overloaded to mean many different things. In industry, behavioral descriptions sometimes refer to specifying a component by its input/output behavior without implementation or structural details[8]. Synopsys Behavioral Compiler allows a module to be specified as sequences of events in loops[12]. Software programming languages have also been used for high-level representation of hardware. Transmorgafier-C[4] and HardwareC[11] compile hardware from a source language based on C. In these systems, some constructs in C are overloaded to convey hardware related information such as clocking and registered storage. The Programmable Active Memory (PAM) project uses an RTL in C++ syntax[13]. Algorithms described in data-parallel C languages have been used to program an array of FPGA's in Splash 2 [6] and CLAy[5]. Sequential C and Fortran programs have been parallelized to target an array of configurable structures in the RAW project[3]. Other high-level representations have also been developed to verify the correctness of hardware. Windley uses the specification language from the HOL[10] theorem proving system to describe a pipelined processor[14]. Matthews et al. have developed the Hawk language to create executable specifications of processor micro-architectures[9].

Ultimately, the goal of a high-level description is to provide an uncluttered design representation that is easy for a human to comprehend and reason about. Although a concise notation is helpful, the utility of a "high-level" description framework has to come from the elimination of some "lower-level" details. It is in this sense that an operation-centric description

can offer an advantage over a state-centric design capture. Any non-trivial hardware design, like a parallel program, consists of multiple concurrent computation structures. This concurrency must be managed explicitly in RTL-like design representations. In an operation-centric description, the behavior of a hardware design is specified as a collection of independent operations. The atomic and sequential execution semantics gives the collection of operations a simple and unambiguous behavioral interpretation. In a operation-centric description, parallelism and concurrency are implicit in the source-level descriptions, only to be discovered and managed by an optimizing compiler.

References

[1] Arvind and X. Shen. Design and verification of processors using term rewriting systems. *IEEE Micro Special Issue on Modeling and Validation of Microprocessors*, May 1999.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines '99*, Napa Valley, CA, April 1999.

[4] D. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 136–144, Napa Valley, CA, April 1995.

[5] M. Gokhale and E. Gomersall. High level compilation for fine grained FPGAs. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines '97*, Napa Valley, CA, April 1997.

[6] M. Gokhale and R. Minnich. FPGA computing in a data parallel C. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–101, Napa Valley, CA, April 1993.

[7] J. C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *Proceedings of VLSI'99*, Lisbon, Portugal, November 1999.

[8] D. Knapp, T. Ly, D. MacMillen, and R. Miller. Behavioral synthesis methodology for HDL-based specification and validation. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, San Francisco, CA, June 1995.

[9] J. Matthews, J. Launchbury, and B. Cook. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, Chicago, IL, 1998.

[10] SRI International, University of Cambridge. *The HOL System Tutorial, Version 2*, July 1997.

[11] Stanford University. *HardwareC – A Language for Hardware Design*, December 1990.

[12] Synopsys, Inc. *Behavioral Compilere Reference Manual.*

[13] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI*, 4(1):56–69, March 1996.

[14] P. J. Windley. Verifying pipelined microprocessors. In *Proceedings of the 1995 IFIP Conference on Hardware Description Languages and their Applications (CHDL)*, 1995.

Contents