LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Hardware Synthesis from Term Rewriting Systems

Computation Structures Group Memo 421A
August 20, 1999

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

In Proceedings of X IFIP International Conference on VLSI (VLSI 99).

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.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Chapter 1

HARDWARE SYNTHESIS FROM TERM REWRITING SYSTEMS

James C. Hoe and Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA
{ jhoe,arvind } @lcs.mit.edu

Abstract Term Rewriting System (TRS) is a good formalism for describing concurrent systems that embody asynchronous and nondeterministic behavior in their specifications. Elsewhere, we have used TRS's to describe speculative micro-architectures and complex cache-coherence protocols, and proven the correctness of these systems. In this paper, we describe the compilation of TRS's into a subset of Verilog that can be simulated and synthesized using commercial tools. TRAC, Term Rewriting Architecture Compiler, enables a new hardware development framework that can match the ease of today's software programming environment. TRAC reduces the time and effort in developing and debugging hardware. For several examples, we compare TRAC-generated RTL's with hand-coded RTL's after they are both compiled for Field Programmable Gate Arrays by Xilinx tools. The circuits generated from TRS are competitive with those described using Verilog RTL, especially for larger designs.

Keywords: Term Rewriting Systems, high level description, high level synthesis, TRAC

1. MOTIVATION

Term Rewriting Systems (TRS's)[Baader and Nipkow, 1998] have been used extensively to give operational semantics of programming languages. More recently, we have used TRS's in computer architecture research and teaching. TRS's have made it possible, for example, to describe a processor with out-of-order and speculative execution succinctly in a page of text[Arvind and Shen, 1999]. Such behavioral descriptions in TRS's are also amenable to formal verification because one can show if two TRS's "simulate" each other. This paper describes hardware synthesis from TRS's.

We describe the Term Rewriting Architecture Compiler (TRAC) that compiles high-level behavioral descriptions in TRS's into a subset of Verilog that can be simulated and synthesized using commercial tools. The TRAC compiler enables a new hardware design framework that can match the ease of today's software programming environment. By supporting a high-level abstraction in design entry, TRAC reduces the level of expertise required for hardware design. By eliminating human involvement in the lower-level implementation tasks, the time and effort for developing and debugging hardware are reduced. These same qualities also make TRAC an attractive tool for experts to prototype large designs.

This paper describes the compilation of TRS into RTL via simple examples. Section 2. presents an introduction to TRS's for hardware descriptions. Section 3. explains how TRAC extracts logic and state from a TRS's type declaration and rewrite rules. Section 4. discusses TRAC's strategy for scheduling rules for concurrent execution to increase hardware performance. Section 5. compares TRAC-generated RTL against hand-coded RTL after each is compiled for Field Programmable Gate Arrays (FPGA) using Xilinx Foundation 1.5i synthesis package. Section 6. surveys related work in high-level hardware description and synthesis. Finally, Section 7. concludes with a few brief remarks.

2. TRS FOR HARDWARE DESCRIPTION

A TRS consists of a set of terms and a set of rewriting rules. The general structure of rewriting rules is:

$$\mathsf{pat}_{lhs}\ \textit{if}\ \mathsf{p} \rightarrow \mathsf{exp}_{rhs}$$

A rule can be used to rewrite a term s if the rule's left-hand-side pattern pat_{lhs} matches s or a subterm in s and the predicate p evaluates to *true*. A successful pattern match binds the free variables of pat_{lhs} to subterms of s. When a rule is applied, the resulting term is determined by evaluating the right-hand-side expression exp_{rhs} in the bindings generated during pattern matching.

In a functional interpretation, a rule is a function which may be expressed as:

$$\begin{array}{l} \lambda\ \mathsf{s}.\ \textit{case}\ \mathsf{s}\ \textit{of} \\ \qquad \mathsf{pat}_{lhs} \Rightarrow \textit{if}\ \mathsf{p}\ \textit{then}\ \mathsf{exp}_{rhs}\ \textit{else}\ \mathsf{s} \\ \qquad\quad \_ \Rightarrow \mathsf{s} \end{array}$$

The function uses a *case* construct with *pattern-matching* semantics in which a list of patterns is checked against s sequentially top-to-bottom until the first successful match is found. A successful match of pat_{lhs} to s creates bindings for the free variables of pat_{lhs}, which are used in the evaluation of the "consequent" expression exp_{rhs}. If pat_{lhs} fails to match to s, the wild-card pattern '_' matches s successfully and the function returns a term identical to s.

In a TRS, the effect of a rewrite 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 a specially-designated *starting* term, successive rewriting progresses until the term cannot be rewritten using any rule.

Example 1 (GCD): Euclid's Algorithm for finding the greatest common divisor (GCD) of two integers may be written as follows in TRS notation:

GCD Mod Rule
$\mathsf{Gcd}(a, b)$ *if* $(a \geq b) \wedge (b \neq 0) \rightarrow \mathsf{Gcd}(a-b, b)$
GCD Flip Rule
$\mathsf{Gcd}(a, b)$ *if* $a < b \rightarrow \mathsf{Gcd}(b, a)$

The terms of this TRS have the form Gcd(a,b), where a and b are positive integers. The answer is the first sub-term of Gcd(a,b) when Gcd(a,b) cannot be reduced any further. For example, the term Gcd(2,4) can be reduced by applying the *Flip* and *Mod* rules to produce the answer 2: Gcd(2,4) → Gcd(4,2) → Gcd(2,2) → Gcd(0,2) → Gcd(2,0) □

TRS's for hardware description are often nondeterministic (*"not confluent"* in the programming language parlance) and restricted so that the terms cannot grow. The latter restriction guarantees that a system described by our TRS's can be synthesized using a finite amount of hardware. The nondeterministic aspect of TRS's has a strong flavor of modeling distributed algorithms as *state-transition systems*. (See for example [Manna and Pnueli, 1991, Lamport, 1994, Lynch, 1996, Chandy and Misra, 1988]). The focus of this paper, however, is on automatic synthesis rather than on formal verification of an implementation against a specification.

In the rest of this section we will describe the TRS notation accepted by TRAC. It includes built-in integers, booleans, common arithmetic and logical operators, non-recursive algebraic types and a few abstract datatypes such as arrays and FIFO's. Other user-defined abstract datatype, with both sequential and combinational functionalities, can be included in synthesis by providing an interface declaration and its implementation.

2.1 SIMPLE TYPES

The language accepted by TRAC is strongly typed, that is, every term has a type specified by the user. The complete list of allowed type declarations are:

TYPE	::	STYPE
	[]	CPRODUCT
	[]	ABSTRACT
CPRODUCT	::	$\text{CN}_k(\text{TYPE}_1, ..., \text{TYPE}_k)$ $where\ k > 0$
ABSTRACT	::	$\text{Array}[\text{STYPE}_{idx}]$ STYPE
	[]	Fifo STYPE
	[]	$\text{Array}_{CAM}[\text{STYPE}_{idx}]$ STYPE_{key}, STYPE
	[]	Fifo_{CAM} STYPE_{key}, STYPE

We begin by describing simple types (STYPE), which include built-in integer, product and algebraic (disjoint) union types. Product types are designated by a *constructor name* followed by one or more elements. An algebraic union is made up of two or more disjuncts. A disjunct is syntactically similar to a product except a disjunct may have zero elements. An algebraic union with only zero-element disjuncts is also known as an enumerable type. Product and algebraic unions can be composed to construct an arbitrary type hierarchy, but *no recursive types are allowed.*

STYPE	::	$\text{Bit}[N]$
	[]	PRODUCT
	[]	ALGEBRAIC
PRODUCT	::	$\text{CN}_k(\text{STYPE}_1, ..., \text{STYPE}_k)$ $where\ k > 0$
ALGEBRAIC	::	DISJUNCT ‖ DISJUNCT
	[]	DISJUNCT ‖ ALGEBRAIC
DISJUNCT	::	$\text{CN}_k(\text{STYPE}_1, ..., \text{STYPE}_k)$ $where\ k \geq 0$

The TRS in Example 1 should be accompanied by the type declaration:

Type	GCD	=	Gcd(NUM, NUM)
Type	NUM	=	Bit[32]

Example 2 (GCD$_2$): We give another implementation of GCD to illustrate some modularity and types issues. Suppose we have the following TRS to implement the *mod* function.

Type	VAL	=	Mod(NUM, NUM) ‖ Val(NUM)
Type	NUM	=	Bit[32]

Mod Iterate Rule
    Mod(a, b) *if* $a \geq b \rightarrow$ Mod($a-b$, b)
Mod Done Rule
    Mod(a, b) *if* $a < b \rightarrow$ Val(a)

Using this definition of *mod*, GCD can be written as follows:

Type GCD_2 = Gcd_2(VAL, VAL)
GCD_2 Flip&Mod Rule
    Gcd_2(Val(a), Val(b)) *if* $b \neq 0 \rightarrow$ Gcd_2(Val(b), Mod(a, b)) □

2.2 ABSTRACT TYPES

Abstract datatypes are defined by their interfaces only and are included to facilitate hardware description and synthesis. An interface can be classified as either *combinational* or *state-transforming*. We discuss array, FIFO and content addressable memory abstract datatypes next.

Array is used to model register files and memories, and has only two operations defined in its interface. Syntactically, if a is an Array then $a[idx]$ represents a combinational "read" operation which gives the value stored in the idx'th location, and $a[idx{:=}v]$, a state-transforming "write" operation gives a new Array identical to a except location idx has been updated to value v. We only support Array of STYPE with an enumerable index type.

Fifo buffers provide the primary means of communication between different modules and pipeline stages. The two main state-transforming operations on Fifo's are *enqueuing* and *dequeuing*. Enqueuing element e to q appears as enq(q,e) while dequeuing the first element from q appears as deq(q). An additional state-transforming interface clr(q) clears the contents of the Fifo. The combinational operation first(q) gives the value of the first element in q. In the description phase, Fifo is abstracted to have a bounded but unspecified size. A rule that makes use of Fifo interfaces has an implied predicate condition that tests whether the Fifo is not empty or not full, as appropriate. We also support access to other Fifo entries with appropriate projection functions. Fifo entries are also restricted to be of STYPE.

Array_{CAM} is similar to Array except its data fields are subdivided into a key field and a normal-data field. The same is true for Fifo_{CAM} and Fifo. The content-associative lookup interface cam(a,key) returns *true* if an entry with a matching key field is found. The content-associative lookup interface camidx(a,key) returns the index of an entry with a matching key field whereas camdata(a,key) returns the data field. The value of camidx(a,key) and camdata(a,key) are undefined when cam(a,key) is *false*.

As can be seen from the definition of TYPE, abstract datatypes are not allowed in algebraic disjuncts. Thus, only a complex product type can have elements of abstract types.

2.3 RULE SYNTAX

Syntactically, a rule is composed of a left-hand-side pattern and a right-hand-side expression. The predicate and *where* bindings are optional. The *where* bindings on the left-hand-side can require pattern matching. Any failure in matching PAT_i to EXP_i in the *where* bindings also deems the rule inapplicable. The expression on the right-hand-side, exp_{rhs}, can also have *where* bindings, but RHS *where* bindings can be made only to simple variables and do not involve pattern matching. In the following '_' represents the "don't care" symbol.

RULE	::	LHS $\rightarrow$ RHS
LHS	::	PAT_{lhs} [*if* EXP_p] [*where* PAT_1=EXP_1, ..., PAT_n=EXP_n]
PAT	::	_ [] *variable* [] *constant* [] CN_0( ) [] CN_k(PAT_1, ..., PAT_k)
RHS	::	EXP_{rhs} [*where* *variable*$_1$=EXP_1, ..., *variable*$_n$=EXP_n]
EXP	::	_ [] *variable* [] *constant* [] CN_0( ) [] CN_k(EXP_1, ..., EXP_k)
	[]	Prim-Op (EXP_1, ..., EXP_k)
Prim-Op	::	*Arithmetic* [] *Logical* [] *Array-Access* [] *FIFO-Access*

The type of PAT_{lhs} must be either CPRODUCT or ALGEBRAIC. *In addition, each rule must have* PAT_{lhs} *and* EXP_{rhs} *of the same type.* This restriction, together with non-recursive type declaration, guarantees that the size of every term is finite and the size does not change by applying the rewriting rules. In Example 2, VAL is an ALGEBRAIC type with two disjuncts, Val and Mod. It is because of this type declaration that the *Mod Done Rule* does not violate the type discipline - both sides of the rule have the type, VAL.

Example 3 (Single-Cycle RISC Processor): The state of an unpipelined, simple RISC processor is described by its program counter (PC), register file (RF) and memory (MEM). This information is captured in the following type declaration:

$$
\begin{array}{lrl}
\textit{Type} & \mathsf{PROC} = & \mathsf{Proc}_s(\mathsf{PC}, \mathsf{RF}, \mathsf{MEM}) \\
\textit{Type} & \mathsf{PC} = & \mathsf{Bit}[N] \\
\textit{Type} & \mathsf{VAL} = & \mathsf{Bit}[N] \\
\textit{Type} & \mathsf{RF} = & \mathsf{Array}\ \mathsf{VAL}[\mathsf{RNAME}] \\
\textit{Type} & \mathsf{RNAME} = & \mathsf{Reg0}(\,) \parallel \mathsf{Reg1}(\,) \parallel \mathsf{Reg2}(\,) \parallel \ldots.\ \mathsf{Reg}m(\,) \\
\textit{Type} & \mathsf{MEM} = & \mathsf{Array}\ \mathsf{INST}[\mathsf{PC}] \\
\textit{Type} & \mathsf{INST} = & \mathsf{Loadc}(\mathsf{RNAME},\mathsf{VAL}) \\
 & & \parallel \mathsf{Loadpc}(\mathsf{RNAME}) \\
 & & \parallel \mathsf{Add}(\mathsf{RNAME},\mathsf{RNAME},\mathsf{RNAME}) \\
 & & \parallel \mathsf{Sub}(\mathsf{RNAME},\mathsf{RNAME},\mathsf{RNAME}) \\
 & & \parallel \mathsf{Bz}(\mathsf{RNAME},\mathsf{RNAME}) \\
 & & \parallel \mathsf{Load}(\mathsf{RNAME},\mathsf{RNAME}) \\
 & & \parallel \mathsf{Store}(\mathsf{RNAME},\mathsf{RNAME})
\end{array}
$$

The processor we synthesized in Section 5. has four 32-bit general purpose registers, i.e. N=32, m=4. The behavior of the 7 instructions — move PC to register, load immediate, register-to-register addition and subtraction, branch if zero, memory load and store — can be specified as a TRS by giving a rewrite rule for each instruction. The following rule conveys the execution of the *Add* instruction.

$$
\begin{array}{ll}
 & \mathsf{Proc}_s(\mathit{pc}, \mathit{rf}, \mathit{mem}) \\
 & \quad \textit{where } \mathsf{Add}(\mathit{rd},\mathit{r1},\mathit{r2}) = \mathit{mem}[\mathit{pc}] \\
\rightarrow & \mathsf{Proc}_s(\mathit{pc}+1,\ \mathit{rf}[\mathit{rd}:=(\mathit{rf}[\mathit{r1}]+\mathit{rf}[\mathit{r2}])],\ \mathit{mem})
\end{array}
$$ □

Example 4 (Pipelined RISC Processor): The processor in Example 3 can be pipelined by introducing FIFO's as pipeline-stage buffers and by systematically splitting each rule into local rules for various pipeline stages. For example, in a two-stage pipeline design, the processing of an instruction can be broken down into separate fetch and execute steps. We model buffers between pipeline stages as a Fifo of an unspecified but finite size. In a behavioral description, it is convenient if the operation of each stage can be described without reference to other stages. FIFO buffers provide this isolation; most pipelined design rules dequeue an input from one FIFO and enqueue the result into another FIFO. In the synthesis phase these FIFO buffers are replaced by a fixed-depth FIFO or simply registers, and flow control logic ensures that a rule does not fire if the destination FIFO is full.

Here, we introduce the pipeline buffer BS in the declaration of the PROC_p term.

Type $\mathsf{PROC}_p = \mathsf{Proc}_p(\mathsf{PC}, \mathsf{RF}, \mathsf{BS}, \mathsf{MEM})$
Type $\mathsf{BS} = \mathsf{Fifo\ ITEMP}$
Type $\mathsf{ITEMP} = \mathsf{Loadc}(\mathsf{RNAME}, \mathsf{VAL})$
|| Loadpc(RNAME)
|| Add(RNAME,VAL,VAL)
|| Sub(RNAME,VAL,VAL)
|| Bz(VAL,VAL)
|| Load(RNAME,ADDR)
|| Store(ADDR,VAL)

The Add and Bz instruction rules are split into *Fetch* and *Execute* stage rules:

Fetch Rule
$\mathsf{Proc}_p(pc, rf, bs, mem)$
$\rightarrow$ $\mathsf{Proc}_p(pc+1, rf, \mathrm{enq}(bs, mem[pc]), mem)$

Add Rule
$\mathsf{Proc}_p(pc, rf, bs, mem)$
where $\mathsf{Add}(rd, r1, r2) = \mathrm{first}(bs)$
$\rightarrow$ $\mathsf{Proc}_p(pc, rf[rd{:=}(rf[r1]+rf[r2])], \mathrm{deq}(bs), mem)$

Branch-Taken Rule
$\mathsf{Proc}_p(pc, rf, bs, mem)$
if $rf[rc]{=}0$ *where* $\mathsf{Bz}(rc, ra) = \mathrm{first}(bs)$
$\rightarrow$ $\mathsf{Proc}_p(rf[ra], rf, \mathrm{clr}(bs), mem)$

Branch-Not-Taken Rule
$\mathsf{Proc}_p(pc, rf, bs, mem)$
if $rf[rc]{\neq}0$ *where* $\mathsf{Bz}(rc, ra) = \mathrm{first}(bs)$
$\rightarrow$ $\mathsf{Proc}_p(pc, rf, \mathrm{deq}(bs), mem)$

Notice the *Fetch* rule is always ready to fire. At the same time one of the execute stage rules may be ready to fire as well. This is the first example we have seen where more than one rule can be enabled on a given state. Even though according to TRS semantics, only one rule should be fired in each step, we will see that our compiler tries to fire as many rules in parallel as possible while maintaining correct TRS execution semantics. Without parallel firing of rules we won't get the pipelining effect we want.

Since there is a race to update the *pc* between the *Fetch* and the *Branch Taken* rules, the above rules can exhibit nondeterministic behavior. Specification of microprocessors and cache-coherence protocols often entails nondeterminism, even though a given realization is usually completely deterministic. Our compiler can handle such nondeterministic TRS's. □

In addition to the TRS-to-RTL compilation to be described in Sections 3. and 4., we are developing source-to-source TRS transformations that can achieve the kind of pipelining described in Example 4. The dependence between the rules has to be analyzed carefully to ensure the correctness of all such transformations. Presently, human intervention is required to guide the transformation process at the high level. It is also possible to automatically derive the rules for a superscalar version of the pipelined processor in Example 4 [Arvind and Shen, 1999].

2.4 INPUT AND OUTPUT

Traditionally a TRS describes a closed system, but we are experimenting with new notations and semantics to support description of a system with input and output (I/O) ports. In an approach that only requires minimal deviation from a standard TRS, the designer assigns I/O specific semantics to terms using source code annotations. For example, a wrapper to start and terminate a GCD computation can be given as:

Type	TOP	=	Top(MODE, NUMI, NUMI, NUMO, GCD)
Type	MODE	=	$\_\_iport\_\_$ Load( ) || Run( )
Type	NUMI	=	$\_\_iport\_\_$ NUM
Type	NUMO	=	$\_\_oport\_\_$ NUM

GCD Start
    Top(Load( ), x, y, _, _)
$\rightarrow$ Top(_, _, _, 0, Gcd(Val(x), Val(y)))

GCD Done
    Top(Run( ), _, _, _, Gcd(Val(ans), Val(0)))
$\rightarrow$ Top(_, _, _, ans, _)

Ignoring the I/O annotations ($\_\_iport\_\_$ and $\_\_oport\_\_$), the type declaration and rules can be interpreted exactly as before. In fact, the combinational logic generated by TRAC is the same irrespective of I/O annotations. The first rule states as long as the first subterm of TOP is Load( ), the GCD term can be rewritten using the second and third subterms of TOP. The second rule states if the first subterm of TOP is Run and the GCD computation is done (when the second subterm of GCD is 0), then copy the first subterm of GCD to the fourth subterm of TOP.

The only effect of annotating the fourth subterm of TOP as an $\_\_oport\_\_$ is that TRAC will attach wires to the output of the registers in that subterm and make their content externally visible through an output port. Conversely, the effect of annotating a term as an $\_\_iport\_\_$ is that the wires normally connected to the output of the registers in that term are redirected to an input port instead. A rule cannot rewrite a term labeled as an $\_\_iport\_\_$ since the value of the term does not correspond to any internal register. From the TRS perspective, an $\_\_iport\_\_$ term may change unexpectedly, but atomically, without any rule application.

By driving the appropriate values on the input ports corresponding to the first three subterms of TOP, a new GCD computation is started. Asserting signals corresponding to Run( ) at the input port enables GCD to execute to completion, and at which point, the answer appears on the output port as a consequence of the *GCD Done* rule.

3. BASIC SYNTHESIS STRATEGY

Although TRS's provide great flexibility in specifying hierarchically organized state and state transitions, a TRS, where recursive types are not allowed and rules are required to have the same type on both sides of $\rightarrow$, can only describe a finite state machine (FSM). TRAC maps a TRS to a synchronous FSM by

- Mapping TRS terms to storage elements (e.g., registers, register files and other abstract datatypes)
- Mapping TRS rules to combinational logic that generates next state values and enable signals for storage elements.

In this section we first describe a functional interpretation of each rule and then derive an "action on state" view of the same rule. The latter view is the starting point for hardware synthesis.

3.1 FUNCTIONAL INTERPRETATION OF A RULE: π AND δ FUNCTIONS

In a functional interpretation, a rule of the form

$$\begin{array}{ll} & \mathsf{pat}_{lhs} \\ & \quad \textit{if } \mathsf{exp}_p \textit{ where } \mathsf{pat}_1 = \mathsf{exp}_{lhs,1}, ..., \mathsf{pat}_n = \mathsf{exp}_{lhs,n} \\ \rightarrow & \mathsf{exp}_{rhs} \\ & \quad \textit{where } \mathsf{var}_1 = \mathsf{exp}_{rhs,1}, ..., \mathsf{var}_m = \mathsf{exp}_{rhs,m} \end{array}$$

is a function of *typeof*(pat_{lhs})$\rightarrow$*typeof*(pat_{lhs}), and returns a term identical to the input term if the rule is not applicable. If the rule is applicable, the return value is a new term based on the evaluation of exp_{rhs} using the bindings created during pattern matching.

$$
\begin{array}{l}
\mathit{rule} = \lambda\ \mathsf{s}.\ \mathit{case}\ \mathsf{s}\ \mathit{of} \\
\qquad\qquad \mathsf{pat}_{lhs} \Rightarrow \\
\qquad\qquad\quad \mathit{case}\ \mathsf{exp}_{lhs,1}\ \mathit{of} \\
\qquad\qquad\qquad \mathsf{pat}_{1} \Rightarrow \\
\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\qquad \mathit{case}\ \mathsf{exp}_{lhs,n}\ \mathit{of} \\
\qquad\qquad\qquad\qquad\quad \mathsf{pat}_{n} \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\quad if\ \mathsf{exp}_{p}\ then \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{let} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{var}_1 = \mathsf{exp}_{rhs,1}, \ldots, \mathsf{var}_m = \mathsf{exp}_{rhs,m} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{exp}_{rhs} \\
\qquad\qquad\qquad\qquad\qquad\quad else \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{s} \\
\qquad\qquad\qquad\qquad\quad \_ \Rightarrow \mathsf{s} \\
\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad \_ \Rightarrow \mathsf{s} \\
\qquad\qquad \_ \Rightarrow \mathsf{s}
\end{array}
$$

This function can be broken down into its two components: π and δ. The π function determines a rule's applicability to a term and has the type, *typeof*(pat_{lhs})$\rightarrow$*Boolean*. The δ function, on the other hand, determines the new term in case π evaluates to *true*.

$$
\begin{array}{l}
\pi = \lambda\ \mathsf{s}.\ \mathit{case}\ \mathsf{s}\ \mathit{of} \\
\qquad \mathsf{pat}_{lhs} \Rightarrow \\
\qquad\quad \mathit{case}\ \mathsf{exp}_{lhs,1}\ \mathit{of} \\
\qquad\qquad \mathsf{pat}_{1} \Rightarrow \\
\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad \mathit{case}\ \mathsf{exp}_{lhs,n}\ \mathit{of} \\
\qquad\qquad\qquad\quad \mathsf{pat}_{n} \Rightarrow \mathsf{exp}_{p} \\
\qquad\qquad\qquad\quad \_ \Rightarrow \mathit{false} \\
\qquad\qquad\quad \ldots \\
\qquad\qquad \_ \Rightarrow \mathit{false} \\
\qquad \_ \Rightarrow \mathit{false}
\end{array}
$$

$$
\begin{array}{l}
\delta = \lambda\ \mathsf{s}.\ \mathit{let} \\
\qquad\quad \mathsf{pat}_{lhs} = \mathsf{s} \\
\qquad\quad \mathsf{pat}_1 = \mathsf{exp}_{lhs,1}, \ldots, \mathsf{pat}_n = \mathsf{exp}_{lhs,n} \\
\qquad\quad \mathsf{var}_1 = \mathsf{exp}_{rhs,1}, \ldots, \mathsf{var}_m = \mathsf{exp}_{rhs,m} \\
\qquad \mathit{in} \\
\qquad\quad \mathsf{exp}_{rhs}
\end{array}
$$

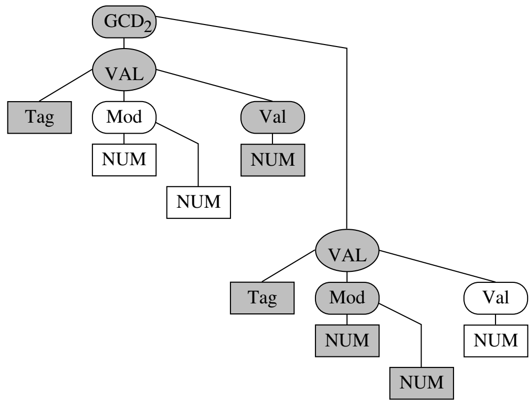

Figure 1.1 A graph representation of the GCD_2 type structure from Example 2. NUM is treated as a type alias for Bit[32].

Using π and δ, an equivalent functional representation of a rule is

$$rule = \lambda\ \mathsf{s}.\ \textit{if}\ \pi(\mathsf{s})\ \textit{then}\ \delta(\mathsf{s})\ \textit{else}\ \mathsf{s}$$

3.2 A RULE AS A STATE TRANSFORMER

In the architectural context, terms represent state, and rules define how the state can be transformed. If we restrict ourselves to synchronous circuits then each rule "reads" the state at the beginning of the clock cycle and if it can fire, it modifies the state at the end of the same clock cycle. In this "actions on state" view of a rule, one needs to update only those parts of the state that actually change. If two rules are enabled simultaneously and affect disjoint parts of the state then it is possible to execute both rules in the same clock cycle. After discussing the hardware to execute one rule in this section, we will return to the issue of concurrent firings in the next section.

Mapping Terms to Storage Elements: A term can be represented as a tree based on its type. For example, the tree representation of GCD_2 is shown in Figure 1.1. Algebraic types have an extra branch, Tag, where a register of width $\lceil log_2 d \rceil$ records which of the d disjuncts the term belongs to. An ALGEBRAIC node has a branch for each of the disjuncts, but, at any time, only the branch whose tag matches the content of the tag register holds meaningful data. As an example we have shaded the active portions of the tree corresponding to Gcd(Val(2), Mod(4, 2)) in Figure 1.1.

We can assign an unique name to each storage element based on its path (also known as projection) from the root. For example, the name for the second (from the left) NUM register in Figure 1.1 would be "$Proj_1$.Mod.$Proj_2$". The storage implied by a term can be represented as a set of $<$*proj*, $REG[N]>$ pairs. For example the storage elements of GCD_2 are represented by the set

$$\{<Proj_1.\mathsf{Tag}, REG[1]>, <Proj_1.\mathsf{Val}.Proj_1, REG[32]>, \\ <Proj_1.\mathsf{Mod}.Proj_1, REG[32]>, <Proj_1.\mathsf{Mod}.Proj_2, REG[32]>, \\ <Proj_2.\mathsf{Tag}, REG[1]>, <Proj_2.\mathsf{Val}.Proj_1, REG[32]>, \\ <Proj_2.\mathsf{Mod}.Proj_1, REG[32]>, <Proj_2.\mathsf{Mod}.Proj_2, REG[32]>\}$$

As an optimization, registers on different disjuncts of an ALGEBRAIC node can share the same physical register. In Figure 1.1, the registers aligned horizontally are mappable to the same register. This idea can be expressed as allowing multiple pathnames to be associated with a single register state element. In a type structure that includes Array and other abstract datatypes, nodes corresponding to the abstract datatypes appear at the leaves of the tree.

The value embedded in the storage elements of a term can be represented in a similar manner using a set of $<$*proj*, *value*$>$ pairs. For example the values of storage elements of Gcd(Val(2), Mod(4, 2)) are represented by the set

$$\{<Proj_1.\mathsf{Tag}, \mathsf{Val}>, <Proj_1.\mathsf{Val}.Proj_1, 2>, \\ <Proj_2.\mathsf{Tag}, \mathsf{Mod}>, <Proj_2.\mathsf{Mod}.Proj_1, 4>, <Proj_2.\mathsf{Mod}.Proj_2, 2>\}$$

The procedure *extract-state*(*s*,*proj*) to extract the values of storage elements from term *s* is defined below. Initially it is called with an empty projection ϵ. Since we propose to use this function only at compile time, we assume the representation of a term includes its type structure.

extract-state(*s*,*proj*)=
  case *s* *of*
    $\mathsf{Bit}[N] \Rightarrow \{<proj, s>\}$
    CPRODUCT: $\mathsf{CN}_k(s_1, ..., s_k) \Rightarrow$
      $extract\text{-}state(s_1, proj.Proj_1) \cup ... \cup extract\text{-}state(s_k, proj.Proj_k)$
    ALGEBRAIC: $\mathsf{CN}_k(s_1, ..., s_k) \Rightarrow$
      $extract\text{-}state(s_1, proj.\mathsf{CN}_k.Proj_1) \cup ...$
        $\cup\ extract\text{-}state(s_k, proj.\mathsf{CN}_k.Proj_k) \cup \{<proj.\mathsf{Tag}, \mathsf{CN}_k>\}$
    Array: $\Rightarrow \{<proj.\mathsf{Array}, s>\}$
    Fifo: $\Rightarrow \{<proj.\mathsf{Fifo}, s>\}$

Rules as Actions on Storage Elements: Because a rule's pat_{lhs} and exp_{rhs} are required to have the same type, the term resulting from a rewrite must have the same storage structure as the initial term. In other words, beginning with a TRS's starting term and its storage elements, successive rewrite operations never add or delete any storage elements. To implement a TRS, TRAC generates a state structure that is extracted from the starting term, and the rules are implemented as combinational logic that updates the content of the storage elements.

The pattern matching on the left-hand-side of a rule (the π function) essentially tests the values of some of the storage elements. π can also include combinational functions from the interface of abstract datatypes. π for the *Flip&Mod* rule of Example 2 will look like the following:

$$\pi = (\mathit{Proj}_1.\mathsf{Tag}(\mathsf{state}) = \mathsf{Val}) \wedge (\mathit{Proj}_2.\mathsf{Tag}(\mathsf{state}) = \mathsf{Val}) \\ \wedge (\mathit{Proj}_2.\mathsf{Val}.\mathit{Proj}_1(\mathsf{state}) \neq 0)$$

The right-hand-side of a rule (or δ) can be viewed as specifying actions on the storage elements of the input term. The actions can be represented in a set of $<\mathsf{proj}, \mathsf{action}>$ pairs. Possible actions include setting a register to a value ($\mathit{set}(\mathsf{v})$) or invoking an abstract datatype's state-transforming interface. The δ of the *Flip&Mod* rule in Example 2 can be viewed as the following set of actions:

$$\{<\mathit{Proj}_1.\mathsf{Tag}, \mathit{set}(\mathsf{Val})]>, <\mathit{Proj}_1.\mathsf{Val}.\mathit{Proj}_1, \mathit{set}(\mathsf{b})>, \\ <\mathit{Proj}_2.\mathsf{Tag}, \mathit{set}(\mathsf{Mod})>, <\mathit{Proj}_2.\mathsf{Mod}.\mathit{Proj}_1, \mathit{set}(\mathsf{a})>, \\ <\mathit{Proj}_2.\mathsf{Mod}.\mathit{Proj}_2, \mathit{set}(\mathsf{b})>\}$$

Recall, a and b refer to some subterms in the initial term s as established by the pattern matching semantics. In cirucit implementation, a and b refer to the initial values of the corresponding storage elements.

Notice, the left-hand-side of the rule requires the first tag register ($\mathit{Proj}_1.\mathsf{Tag}$) to be Val when this rule is applicable. Thus we can detele the action $<\mathit{Proj}_1.\mathsf{Tag}, \mathit{set}(\mathsf{Val})>$ without affecting the outcome. Thus, if a compiler can detect that a storage element is assigned the same value as its original content, it can delete that particular action. In general, the necessary actions when a rule fires are

$$\mathit{extract\text{-}state}(\delta(\mathsf{s}),\epsilon) - \mathit{extract\text{-}state}(\mathsf{s},\epsilon)$$

where '$-$' represents the set difference. In practice, instead of dynamically testing for equality between the next and current state values of a register to eliminate actions, TRAC statically eliminates actions in which a register is updated by a value coming from itself and when a register is updated by the same value that it must have for π to be satisfied.

In another example, consider the pipelined processor of Example 4, whose storage elements are

$$\{<\mathit{Proj}_1, \mathsf{pc}>, <\mathit{Proj}_2.\mathsf{Array}, \mathsf{rf}>, \\ <\mathit{Proj}_3.\mathsf{Fifo}, \mathsf{bs}>, <\mathit{Proj}_4.\mathsf{Array}, \mathsf{mem}>\}.$$

The Add rule specifies the following actions on this state:

$\{<$*Proj*$_2$.Array, array-update(*rd*,*rf*[*r1*]+*rf*[*r2*])$>$ $<$*Proj*$_3$.Fifo, deq( )$>\}$

In general, a rule can be applied to a subterm of a whole term. In these cases, *extract-state*(*s*,*proj*) is called by a projection, relative to the whole term, that corresponds to the subterm. Furthermore, a rule can be applied to many parts of a term. In these cases, a rule's logic is instantiated multiple times, once for each state sub-structure where the rule is to be applied. In an alternative interpretation, a subterm-applicable rule needs to be lifted to the same type as the TRS's starting term prior to analysis. The effect of applying the lifted rule to the whole term is the same as applying the original rule to the subterm within the whole term. A subterm rule may be applicable to multiple positions in the whole term. A separate lifted version must be created for each possible application. For example, the *Mod Done* rule from GCD_2 in Example 2 could be applicable to both the first and second subterms of a GCD_2 term. The two lifted versions of the *Mod Done* rule are:

$$\begin{array}{ll} & \text{Gcd}_2(\text{Mod}(a, b), t) \textit{ if } a<b \\ \rightarrow & \text{Gcd}_2(\text{Val}(a), t) \end{array}$$

and

$$\begin{array}{ll} & \text{Gcd}_2(t, \text{Mod}(a, b)) \textit{ if } a<b \\ \rightarrow & \text{Gcd}_2(t, \text{Val}(a)) \end{array}$$

3.3 CIRCUIT SYNTHESIS

The π and δ functions for the two GCD rules, *GCD Mod* and *GCD Flip*, in Example 1 are given below. A valid starting term for this TRS has the form Gcd(x, y) where x and y are postive integers. This starting term implies the set of storage elements: $\{<$*Proj*$_1$, *REG*[32]$>$, $<$*Proj*$_2$, *REG*[32]$>\}$. For conciseness, we refer to these registers as *a* and *b* in the following definitions:

$$\begin{array}{lcl} \pi_{Mod} & = & a \geq b \wedge b \neq 0 \\ \pi_{Flip} & = & a<b \\ \delta_{Mod,a} & = & set(a-b) \\ \delta_{Flip,a} & = & set(b) \\ \delta_{Flip,b} & = & set(a) \end{array}$$

For hardware synthesis we break down δ into actions on individual storage elements as specified above. Therefore, for each storage element *e* affected by a rule R, $\delta_{R,e}$ gives its next state value. π_R is the latch-enable signal of all the affected registers. Two state transition circuits corresponding to the two GCD rules, considered indenpendently, is first shown in Figure 1.2.

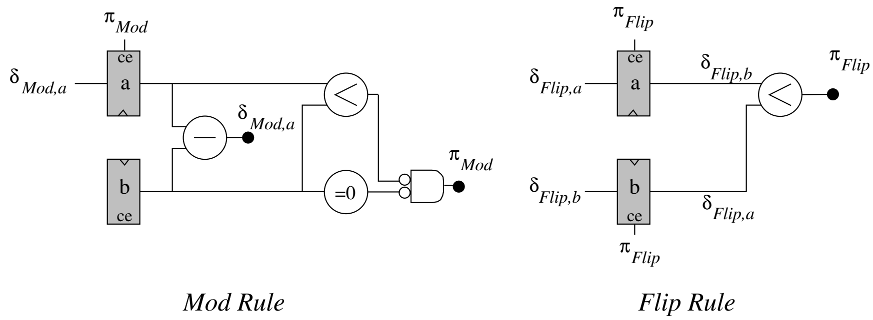

Figure 1.2 FSM for a TRS with only one rule.

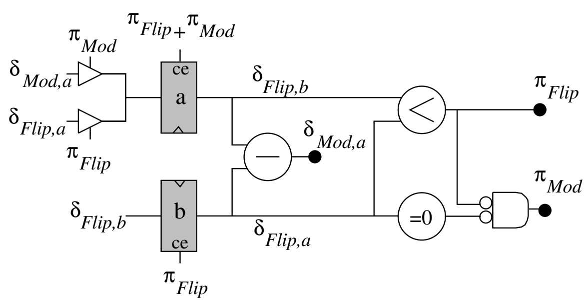

Figure 1.3 Circuit for computing Gcd(*a*, *b*) from Example 1.

The final circuit is arrived by combining the two circuits. In these cases, both rules affect the storage element *a* but only one of them can actually fire in a given state. When merging the actions from rules with mutally-exclusive firing conditions (π), the final latch enable is simply the logical-OR of their firing conditions (e.g., $\pi_{Mod} + \pi_{Flip}$ in this example), and the next state values are chosen from all of the δ's using a multiplexer where a rule's π enables its own δ. A sample update circuit that merges δ's from two mutually-excluisve rules is illustrated as circuit A in Figure 1.4. Figure 1.3 shows the FSM generated by combining the π and δ from both GCD rules.

However, in general, several π's could be asserted, i.e., several rules could be applicable. In the simplest solution, a new set of disjoint triggers $\phi_1, ..., \phi_n$ can be generated using a round-robin priority encoder fed by $\pi_1, ..., \pi_n$. ϕ's, which are mutually exclusive, globally replace π's at all multiplexers and at all latch enable OR-gates. A sample update circuit that merges δ's from two possibly conflicting rules is illustrated as circuit B in Figure 1.4. This arbitration is simple and correct, but the circuit is inefficient and allows only one rewrite per cycle.

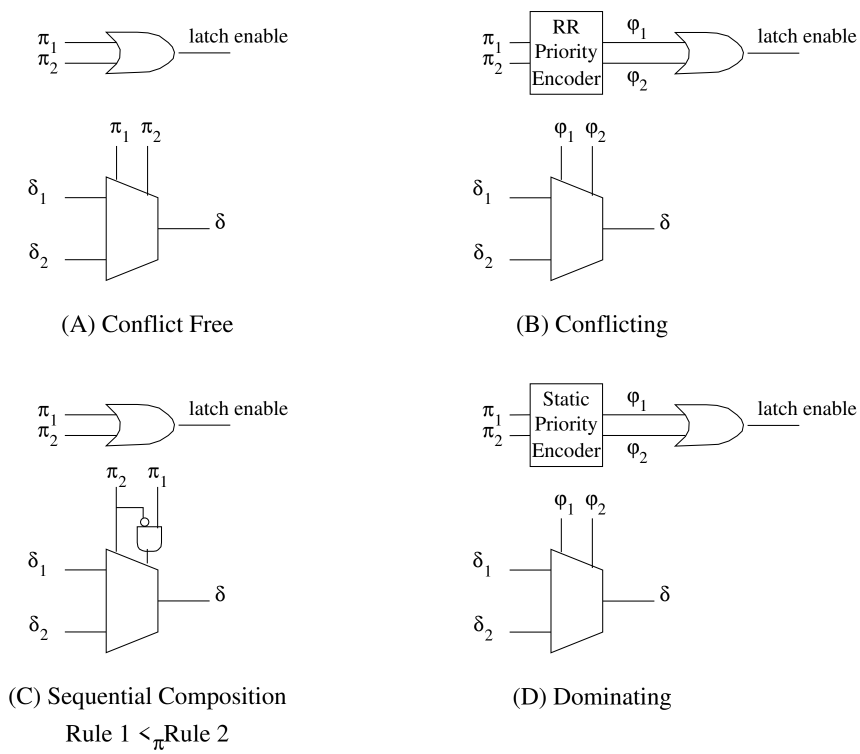

Figure 1.4 Circuits for combining two rules' δ actions on the same state element.

TRAC does not synthesize any state structures for abstract datatypes. When an abstract datatype is used in a TRS, TRAC instantiates the corresponding Verilog module in the RTL and makes appropriate connections to the interfaces. The user or the library is expected to provide a Verilog module in RTL for each abstract datatype. A state transforming interface has an implied signal driving by π (or ϕ) to enable the state changes when the corresponding rule is fired.

4. EXPLOITING PARALLELISM

According to TRS semantics, if multiple rules can simultaneously become applicable on a given term `s`, one of the rules is chosen nondeterministically and applied atomically to rewrite `s` to `s`'. Next, a new round of rewriting is started from scratch on `s`'. When a TRS exhibits such nondeterminism, multiple behaviors are allowed. Using a scheduler based on a round-robin priority encoder as discussed in Section 3.3, TRAC implements one of the allowed behaviors in a deterministic circuit that fires one rule per clock cycle.

If the simultaneously applicable rules involve mutually disjoint parts of the term, then these rules can be executed in any sequence successively to reach the same final term. In this scenario, although the semantics of a TRS specifies a sequential and atomic term rewriting, a hardware implementation can exploit the underlying parallelism and execute the rules concurrently in the same clock cycle. In general it is not safe to allow two arbitrary applicable rules to execute in the same clock cycle because executing one of them can alter the value of the π or the δ function of the other. This section formalizes the conditions for simultaneous rule execution and suggests a scheduling that improves hardware performance by firing multiple rules in the same clock cycle when allowed.

4.1 TRANSPARENCY

The minimum condition for allowing two simultaneously applicable rules to fire in the same clock cycles is captured by the **π-transparent** relationship.

Definition 1 (π-transparent)
*Rule R_1 is **π-transparent** to rule R_2, denoted as $R_1 <_\pi R_2$, if* $\forall s.\pi_1(s) \wedge \pi_2(s) \Rightarrow \pi_2(\delta_1(s))$

This condition states that if two rules ever become applicable on the same term and $R_1 <_\pi R_2$, then firing R_1 first does not prevent R_2 from firing on the resulting term. Firing in the reverse order may not necessarily be allowed, unless a stronger condition of mutual-transparency (or **π-conflict-free**) is satisfied.

Definition 2 (π-conflict-free)
*Rules R_1 and R_2 are **π-conflict-free** if* $(R_1 <_\pi R_2) \wedge (R_2 <_\pi R_1)$

Given two rules where $R_1 <_\pi R_2$, there are two basic approaches to allow both rules to fire in the same clock cycle. The first approach cascades the combinational logic from the two rules such that R_1 is applied first to the physical state elements, and R_2 is applied to the *effective* state after attempting to apply R_1. In effect, we are creating a composite rule where

$$
\begin{array}{l}
\lambda\, \mathtt{s}\,.\ \textit{if}\ \pi_1(\mathtt{s})\ \textit{then} \\
\qquad\qquad \textit{if}\ \pi_2(\mathtt{s})\ \textit{then}\ \delta_2(\delta_1(\mathtt{s}))\ \textit{else}\ \delta_1(\mathtt{s}) \\
\qquad\ \ \textit{else if}\ \pi_2(\mathtt{s})\ \textit{then}\ \delta_2(\mathtt{s})\ \textit{else}\ \mathtt{s}
\end{array}
$$

Arbitrary cascading does not always improve circuit performance since cascading combinational logic may lead to a longer cycle time, especially when several rules are composed. In a synchronous design, if the clock period increases, every rule firing is penalized, even when at most one rule can fire.

In a more practical approach, the input to the combinational logic from all rules are driven directly by state elements. Two transparent rules are allowed to execute in the same clock cycle only if the correct resulting state can be constructed from independent evaluation of the same current state.

4.2 PARALLEL COMPOSIBILITY

Two rules that do not affect the same storage elements are **parallel composible**, provided allowing them to execute concurrently on the same state produces a behavior that corresponds to at least one ordering of rule-execution in TRS.

Definition 3 (Parallel-Composible Transparency)
*Rule R_1 is **PC-transparent** to rule R_2, denoted as $R_1 <_{PC} R_2$, if*

$$(R_1 <_\pi R_2) \wedge \forall s.(\pi_1(s) \wedge \pi_2(s)) \Rightarrow \delta_2(\delta_1(s)) = \mathbf{PC}(s, \delta_1(s), \delta_2(s))$$

Definition 4 (Parallel Composition)

$\mathbf{PC}(\mathsf{s}, \mathsf{s}_1, \mathsf{s}_2) =$
case s *of*
Bit $v \Rightarrow$
if $\mathsf{s}_1 = \mathsf{s}_2$ *then* s_1
else if $\mathsf{s}_1 = v$ *then* s_2
else if $\mathsf{s}_2 = v$ *then* s_1
else $\perp$
$\mathsf{CN}_k(...) \Rightarrow$
if $\mathsf{s}_1 = \mathsf{s}_2$ *then* s_1
else if $\mathsf{s}_1 = n$ *then* s_2
else if $\mathsf{s}_2 = n$ *then* s_1
else if $(\mathsf{Tag}(\mathsf{s}_1) = \mathsf{CN}_k) \wedge (\mathsf{Tag}(\mathsf{s}_2) = \mathsf{CN}_k)$ *then*
$\mathsf{CN}_k(\mathbf{PC}(Proj_1(s), Proj_1(\mathsf{s}_1), Proj_1(\mathsf{s}_2)), ...,$
$\mathbf{PC}(Proj_k(s), Proj_k(\mathsf{s}_1), Proj_k(\mathsf{s}_2))$
else $\perp$
$\mathsf{Array}_n\ a \Rightarrow \forall_{1 \leq i \leq n}.\ a[i{:=}\ \mathbf{PC}(a[i], \mathsf{s}_1[i], \mathsf{s}_2[\mathrm{i}])]$
Fifo $f \Rightarrow$
if $(\mathsf{s}_1$ *is suffix of* $f) \wedge (f$ *is prefix of* $\mathsf{s}_2)$ *then*
chop_prefix(chop_suffix(s_1, f*),*s_2*)*
if $(\mathsf{s}_2$ *is suffix of* $f) \wedge (f$ *is prefix of* $\mathsf{s}_1)$ *then*
chop_prefix(chop_suffix(s_2, f*),*s_1*)*
else $\perp$

Essentially what this definition says is that, if both rules R_1 and R_2 want to update a register, then they must produce the same value. In the case of an array, if the two rules update different elements of the array, then parallel composition will work assuming the array has multiple write ports. In the case of a FIFO, if one rule enqueues and the other dequeus then they can be combined to execute in the same cycle.

Note $R_1 <_{PC} R_2$ does not imply that the outcome is confluent. Consider the following two rules that operate on four registers:

R_1: $\mathsf{F}(1, r_B, r_C, r_D) \rightarrow \mathsf{F}(1, r_B, 1, r_D)$
R_2: $\mathsf{F}(r_A, 1, r_C, r_D) \rightarrow \mathsf{F}(0, 1, r_C, 1)$

Now consider the starting term $\mathsf{F}(1,1,r_C,r_D)$. The effect of executing R_1 after R_2 is $\mathsf{F}(0,1,1,1)$. On the other hand if R_2 is executed first the result would be $\mathsf{F}(0,1,r_C,1)$ and R_1 will no longer fire.

For two rules to be confluent we need the following stronger condition.

Definition 5 (Conflict-free)
*Rules R_1 and R_2 are **conflict-free** if* $(R_1 <_{PC} R_2) \wedge (R_2 <_{PC} R_1)$

If two rules are parallel composible, the δ's do not collide and no special merging circuit is required to arbitrate their actions on the affected storage elements.

4.3 SEQUENTIAL COMPOSIBILITY

Even if two rules do affect some common state, by carefully prioritizing the effect of the two rules such that the effects of R_2 overrides R_1 (in case $R_1 <_\pi R_2$), a legal outcome can still be constructed from simultaneous evaluation of the two rules on the same current state.

Definition 6 (Sequentially-Composible Transparency)
*Rule R_1 is **SC-transparent** to rule R_2, denoted as $R_1 <_{SC} R_2$, if*

$$(R_1 <_\pi R_2) \wedge \forall s.(\pi_1(s) \wedge \pi_2(s)) \Rightarrow \delta_2(\delta_1(s)) = \mathbf{SC}(s, \delta_1(s), \delta_2(s))$$

Sequential composition that implements the priotization is defined as

Definition 7 (Sequential Composition)

$\mathbf{SC}(\mathsf{s}, \mathsf{s}_1, \mathsf{s}_2) =$
  case s *of*
    $\mathsf{Bit}\ v \Rightarrow$ *if* $\mathsf{s}_2 = v$ *then* s_1 *else* s_2
    $\mathsf{CN}_k(...) \Rightarrow$
      if $\mathsf{s}_2 = s$ *then* s_1
      else if $(\mathsf{Tag}(\mathsf{s}_1) = \mathsf{CN}_k \wedge (\mathsf{Tag}(\mathsf{s}_2) = \mathsf{CN}_k)$ *then*
        $\mathsf{CN}_k(\mathbf{SC}(Proj_1(s), Proj_1(\mathsf{s}_1), Proj_1(\mathsf{s}_2)), ...,$
            $\mathbf{SC}(Proj_k(s), Proj_k(\mathsf{s}_1), Proj_k(\mathsf{s}_2))$
      else s_2
    $\mathsf{Array}_n\ a \Rightarrow \forall_{1 \leq i \leq n}.\ a[i := \mathbf{SC}(a[i], \mathsf{s}_1[i], \mathsf{s}_2[i])]$
    $\mathsf{Fifo}\ f \Rightarrow$
      if (s_1 *is suffix of* f) $\wedge$ (f *is prefix of* s_2) *then*
        chop_prefix(*chop_suffix*(s_1, f),s_2)
      if (s_2 *is suffix of* f) $\wedge$ (f *is prefix of* s_1) *then*
        chop_prefix(*chop_suffix*(s_2, f),s_1)
      else $\perp$

If R_1 and R_2 are sequentially composible ($R_1 <_\pi R_2$), then prioritized ϕ_1 and ϕ_2 can be generated. However, instead of applying them globally, they are only used to replace π_1 and π_2 at state elements that are affected by both rules. If a register is only affected by either R_1 or R_2 then π can be used directly. Circuit (C) in Figure 1.4 illustrates the update circuit for this case.

4.4 DOMINANCE

Definition 8 (Dominance)
Rule R_2 ***dominates*** *rule R_1, denoted as $R_1 <_D R_2$, if*
$$(R_1 <_\pi R_2) \wedge \forall s.(\pi_1(s) \wedge \pi_2(s)) \Rightarrow \delta_2(\delta_1(s)) = \delta_2(s)$$

If two rules, R_1 and R_2 are conflicting, but R_2 dominates R_1, we can include this information in the priority encoder when generating ϕ's for global replacement of their π's. If π_1 and π_2 are both asserted on a cycle, instead of using a fair round-robin priority encoder, the encoder would statically give priority to π_2. For a two rule circuit, ϕ_2=π_2 and ϕ_1=$\pi_1 \wedge \neg \pi_2$. Circuit (D) in Figure 1.4 illustrates the update circuit for this case.

4.5 SCHEDULING FOR SIMULTANEOUS FIRING

To conclude this section, we describe a scheduler that is currently implemented in TRAC that makes use of conflict-free ($\mathcal{C}F$) relationships. In general, an exact test for $\mathcal{C}F$ relationship between two arbitrary rule instances is expensive (Finding an *s* such that $\pi_i(s) \wedge \pi_j(s)$ is like solving SAT). Instead, TRAC performs several conservative tests to find as many $\mathcal{C}F$ relationships as possible. First, two rule instances that read and write non-overlapping parts of the systems are $\mathcal{C}F$. If two rule instances do not rewrite the same registers, and if none of the registers affected by the δ of one is used by the π and δ of the other, and vice versa, then the two rules are $\mathcal{C}F$ since this condition is stronger than the requirement for $\mathcal{C}F$. Lastly, TRAC symbolically analyzes pairs of π's to conservatively determine when a pair can never be satisfied simultaneously and thus are $\mathcal{C}F$ by default.

TRAC makes use of certain axioms when analyzing the conflict relationships between rules that reference abstract datatype interfaces. For example,

(*a*[*idx*:=v])[*idx*]$=$*v*
((*a*[*idx*:=v])[*idx*':=*v*])[*idx*]$=$*v* if *idx* $\neq$ *idx*'

deq(enq(*q*,*e*)) $=$ enq(deq(*q*),*e*) if *q* is not empty
first(*q*) $=$ first(enq(*q*,*e*)) if *q* is not empty

Based on the analysis above and taking into account the properties of FIFO buffers, it can be shown that the rules of Example 4 are $\mathcal{C}F$ except for the *Fetch* and the *Branch-Taken* rule. However, it can be shown that the *Branch-Taken* rule dominates the *Fetch* rule in the sense that the effect of applying the *Branch-*

Taken rule after the *Fetch* rule is the same as not applying the *Fetch* rule at all i.e., ($\delta_{BzN}(s) = \delta_{BzN}(\delta_{Fetch}(s))$). Thus, instead of arbitrating between these two rules, the compiler gives priority to the *Branch-Taken* rule.

After TRAC has establish $\mathcal{CF}$ relationships between as many rule instances as possible, a graph of rule instances can be constructed by adding an edge between each non-$\mathcal{CF}$ pairs. Scheduling groups is formed by partitioning the graph into connected components. Different groups never interfere and can be scheduled independently. For each group, a round-robin priority encoder can be used to map π to ϕ for arbitration. For a small group, an $n \times n$ look-up table can be computed off-line to encode π to ϕ where more than one ϕ can be asserted if the rules of the asserted π's are $\mathcal{CF}$.

5. PERFORMANCE EVALUATION

TRAC generates RTL Verilog that can be synthesized to a variety of technologies by commercial tools like Synopsys and Xilinx hardware compilers. In this paper, we evaluate the quality of the TRAC-generated RTL's against hand-coded RTL when compiled for Xilinx FPGA's.

Synthesis of the GCD Circuit: Both Example 1 and 2 are compiled to RTL by TRAC. The compile time is less than 2 sec on a 166MHz PowerPC604e. As a reference, our colleague, Daniel L. Rosenband, provided a hand-optimized Verilog RTL for GCD that uses only two 32-bit registers, a single subtracter, and simple boolean logic gates. The three RTL's are compiled for XC4010XL-09 FPGA using Xilinx Foundation 1.5i tools. We report the number of flip-flops and the overall utilization of the FPGA. In addition to the maximum clock frequency, we also report the number of clock cycles needed to compute GCD(53857×10957,91159×10957).

Version	FF (bit)	Util. (%)	Freq. (MHz)	Elapse (cyc)
Example 1	64	20	44.2	54
Example 2	102	38	31.5	104
Hand RTL	64	16	53.1	54

The RTL generated by TRAC from Example 2 is significantly worse than the hand-coded RTL because the input TRS maps to a sub-optimal hardware structure. TRAC does not have the same ingenuity that allowed our colleague to realize the high-level transformations that lead to the smaller and simpler circuit of the hand-optimized RTL. However, the necessary information to achieve the same high-level transformation can be expressed at the TRS level. Given Example 1, TRAC produces an RTL that is structurally similar to the hand-coded version and compiles to within 25% of the hand-written RTL in terms of circuit size and 17% in terms of circuit speed.

Synthesis of the Unpipelined Microprocessor: Hand-optimization can often produce much more efficient implementations than machine compilation on small designs. However, as the problem size increases, the pay-back of hand optimizations diminishes while the effort required increases dramatically. This is evident in the synthesis of the simple microprocessor from Example 3. The TRAC generated RTL and a hand-coded Verilog RTL of the unpipelined processor, when targeting an XC4013XL-08 FPGA, are comparable both in size and speed.

Version	FF (bit)	Util. (%)	Freq. (MHz)
Example 3	161	60 %	40.0
Hand RTL	160	50 %	41.0

6. RELATED WORK

A *behavioral* description refers to specifying a component by its input/output behavior without implementation or structural details. In industry, such descriptions are given typically in a sequential language like the behavioral portion of Verilog. Another approach is to extend or adapt a popular software language. Transmorgafier-C[Galloway, 1995] and HardwareC[HardwareC, 1990] 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. In the Programmable Active Memory (PAM) project, Vuillemin, et al. synthesize from an RTL in C++ syntax[Vuillemin et al., 1996]. Algorithms described in data-parallel C languages have been used to program an array of FPGA's in Splash 2 [Gokhale and Minnich, 1993] and CLAy[Gokhale and Gomersall, 1997]. Sequential C and Fortran programs have been parallelized to target an array of simple configurable hardware structures[Babb et al., 1999]. The TRS-based behavioral descriptions are different from these approaches because on one hand TRS terms convey structural information about the hardware, but on the other hand, TRS rules can embody a set of behaviors, including concurrency and nondeterminism. This is not possible to express in any sequential language. TRS also offers a well-understood formalism which is useful in verification.

More related to TRS are hardware description languages that have been developed in the context of formal specification and verification. TRS is perhaps closest to Lamport's TLA's. Windley uses the specification language from the HOL[HOL, 1997] theorem proving system to describe a pipelined processor[Windley, 1995]. Matthews et al. have developed the Hawk language to create executable specifications of processor micro-architectures[Matthews et al., 1998]. However, none of these systems has been used in synthesis to the best of our knowledge. With a somewhat different motivation, Communicating Sequential Processes have been applied to hardware-software co-design by Gupta et al.[Gupta and de Micheli, 1993] and Thomas et al.[Thomas et al., 1993].

7. CONCLUSION

When applied in conjunction with reconfigurable technologies, TRAC can drastically lower the entry cost of taking on a hardware project by people who are not hardware designers by training. Compilers like TRAC have the potential to close the traditional distinction of hardware and software by creating a continuum of trade-offs between development cost and performance. We anticipate the day when all computers are shipped with a FPGA next to the CPU, and developers are just as ready to program the FPGA for a performance critical application as they would program the processor today.

Acknowledgments

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.

References

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

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

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

[Chandy and Misra, 1988] Chandy, K. M. and Misra, J. (1988). *Parallel Program Design: A Foundation*. Addison-Wesley.

[Galloway, 1995] Galloway, D. (1995). 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.

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

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

[Gupta and de Micheli, 1993] Gupta, R. and de Micheli, G. (1993). Hardware-software cosynthesis for digital systems. *IEEE Design and Test of Computers*, pages 29–41.

[HardwareC, 1990] HardwareC (1990). *HardwareC – A Language for Hardware Design*. Stanford University.

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

[Lamport, 1994] Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3).

[Lynch, 1996] Lynch, N. (1996). *Distributed Algorithms*. Morgan Kaufmann.

[Manna and Pnueli, 1991] Manna, Z. and Pnueli, A. (1991). *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag.

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

[Thomas et al., 1993] Thomas, D. E., Adams, J. K., and Schmit, H. (1993). A model and methodology for hardware-software codesign. *IEEE Design and Test of Computers*, pages 6–15.

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

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