Synchronous Extensions to Operation-Centric Hardware Description Languages

Grace Nordin and James C. Hoe

Carnegie Mellon University
Pittsburgh, USA
{yhn, jhoe}@ece.cmu.edu

Abstract

The Abstract Transition System (ATS) is a high-level hardware description framework. ATS's operation-centric abstraction permits perspicuous descriptions of complex concurrent hardware behavior as a sequence of atomic state transitions. However, non-determinism in the ATS semantics prevents it from capturing the behavior of systems whose correctness depends upon both function and exact synchronous timing. To address this shortcoming, we present two extensions to ATS—committing transitions and synchronously delayed expressions—to support the specification of synchronous behaviors and interfaces. The new synchronous extensions compose naturally with the original ATS. We describe a compilation strategy for the synchronous extensions that leverages existing ATS synthesis capabilities. We also evaluate the new extensions' ease of description and synthesis quality in several design examples.

1. Introduction

The Abstract Transition System (ATS), which shares similar semantics as Term Rewriting Systems (TRS) [2] and guarded commands [7], was previously proposed for high-level hardware description and synthesis [9, 10]. ATS hardware description is *operation-centric* because the description of behavior is organized into atomic operations that each affect multiple state elements (rather then next-state equations that each control one state element). An ATS consists of a set of state elements and a set of state transitions. An ATS transition comprises a predicate and a set of state-update actions. In a state where the predicates of several transitions are simultaneously true, any one of those transitions can be selected non-deterministically. The state-update actions of an ATS transition are applied atomically; that is, in each round of state transitions, all transitions read all state values in one instantaneous step before the one selected transition writes all state elements in another instantaneous step. Thus, the execution of an ATS can be abstractly interpreted as a sequence of atomic applications of transitions, where each transition produces a state that satisfies the predicate of the next transition.

The atomic and sequential semantics of ATS do not prevent a correct implementation from executing several transitions concurrently. References [9] and [10] describe a method for synthesizing ATS into a highly-concurrent clock-synchronous implementation that efficiently executes multiple transitions per clock cycle but still maintain the appearance of a sequential and atomic execution. Additional details of ATS and its compilation are described in Section 2.

Despite its various advantages, the non-deterministic semantics of ATS presents an obstacle to describing hardware designs that require synchronous and deterministic operation. The non-deterministic semantics of ATS says that any enabled transition is permitted, but not committed, to execute. Hence, an ATS designer could not guarantee that a transition will execute in a particular clock cycle, or even at all. To address this shortcoming, we propose two synchronous extensions to ATS that permit the specification of synchronous behavior and compose naturally with the original ATS.

The two extensions are *committing transitions* and *synchronously delayed expressions*. The resulting ATS+ is intended to simplify the description of a hardware system where only a small portion of the overall activity must be synchronous. For example, in a layered communication stack, the activities in the upper layers interact with asynchronous handshakes (e.g., request/grant), and only the bottom-most physical layer must match the timing of the physical medium. When describing a controller for such a protocol, we can simplify the specification of the upper-layers using the original ATS's simplifying atomic and sequential semantics. At the same time, where necessary, the

[0]0-7803-8509-804$20.00 ©2004 IEEE

$$
\begin{array}{lcl}
\mathcal{ATS} & = & \langle \mathcal{S}, \mathcal{S}^o, \mathcal{X} \rangle \\
\mathcal{S} & = & \langle R_1,.., A_1,.., F_1,.. \rangle \\
\mathcal{S}^o & = & \langle v^{R_1}, .., v^{A_1}, .., v^{F_1}, .. \rangle \\
\mathcal{X} & = & \langle T_1,..,T_M \rangle \\
\mathcal{T} & = & \langle \pi,\alpha \rangle \\
\pi & = & exp \\
\alpha & = & \langle a^{R_1}, .., a^{A_1}, .., a^{F_1}, .. \rangle \\
a^R & = & \epsilon \parallel set(exp) \\
a^A & = & \epsilon \parallel aset(exp_{idx}, exp_{data}) \\
a^F & = & \epsilon \parallel enq(exp) \parallel deq() \\
 & & \parallel endeq(exp) \parallel clear() \\
exp & = & constant \parallel R.get() \\
 & & \parallel PrimitiveOp(exp_1, .., exp_n) \\
 & & \parallel A.aget(exp_{idx}) \parallel F.first() \\
 & & \parallel F.notfull() \parallel F.notempty()
\end{array}
$$

Figure 1. ATS summary

physical layer can be described synchronously using the new ATS+ extensions. The two portions of the design can nevertheless be interpreted together under a simple sequential and atomic interpretation as in the original ATS.

Paper outline. Section 2 provides additional background on ATS. Section 3 introduces ATS+ extensions. Section 4 explains the compilation of ATS+ for hardware synthesis. Section 5 presents an evaluation of ATS+; this evaluation compares ATS+ to Verilog RTL and Esterel. Section 6 discusses prior work in operation-centric and synchronous HDLs. Finally, Section 7 presents a summary and our conclusions.

2. ATS primer

In this section, we present the ATS abstraction. ATS is an intermediate hardware representation used to support the compilation of operation-centric source-level languages such as TRSpec [9] and Bluespec [1]. We use ATS to present the ideas in this paper to avoid the complications of source-level languages.

2.1. Overview

ATS is a state-based abstract hardware representation with operation-centric state transitions. The structure of ATS is summarized in Figure 1. At the top level, an ATS is a triple $\langle \mathcal{S}, \mathcal{S}^o, \mathcal{X} \rangle$. $\mathcal{S}$ is a list of explicitly declared state elements, including registers (R), arrays (A), and FIFOs (F). $\mathcal{S}^o$ is a list of initial values for the elements in $\mathcal{S}$. $\mathcal{X}$ is a list of transitions, $\langle \pi,\alpha \rangle$. In a transition, π is a boolean predicate expression and α is a list of concurrent actions, with exactly one action for each state element in $\mathcal{S}$. (An acceptable action for all statement types is the null action 'ϵ'.) In an ATS, if π of transition T_x is enabled in a state s (abbreviated as $\pi_{T_x}(s)$=*true*, or simply $\pi_{T_x}(s)$), then the concurrent actions prescribed by α_{T_x} can be applied atomically to update s to $\delta_{T_x}(s)$. We use the notation $\delta_{T_x}(s)$ to mean the resulting state s' after applying α_{T_x} to s; in other words, δ_{T_x} is the functional equivalent of α_{T_x}. Recall from Section 1, the operation-centric transition semantics is non-committing, that is an enabled transition is permitted, but not committed to execute.

An R-type register state element can store an integer value up to a specified maximum word size. The value stored in a register *R* can be referenced using the side-effect-free *get()* query and set to *v* using the *set(v)* action. (In our examples, we abbreviate *R.get()* simply as *R* and *R.set(v)* as *R:=v*.) Figure 1 lists the actions supported for arrays and FIFOs; a complete description of ATS, including I/O elements, is given in [9]. Without loss of generality, this paper focuses on systems with only R elements.

2.2. Compilation

Basic strategy. This paper is concerned with mapping ATS to a clock-synchronous implementation. In the assumed mapping strategy, the elements of $\mathcal{S}$—registers in this case—are instantiated from a design library and the transitions in $\mathcal{X}$ are combined to form the next-state logic for the instantiated state elements. In each clock cycle, the π expressions of all transitions are evaluated combinationally and some subset of transitions whose π expression is asserted is selected to update the state elements on the next clock edge.

In a naive implementation, only one transition is selected in each clock cycle—this automatically satisfies the sequential and atomic semantics of operation-centric transitions. This naive implementation is functionally correct but inefficient due to a lack of hardware concurrency. The implementation produced by an ATS compiler in reality employs an arbitration logic that selects multiple enabled transitions to update state elements concurrently in each clock cycle, provided the resulting new state values correspond to some valid sequence of atomic execution of the same constituent transitions. The ATS compiler generates such an arbitration logic based on a static analysis of *conflict-free* ($<>_{CF}$) and *sequentially-composability* ($<_{SC}$) properties among the ATS transitions.

Arbiter synthesis. $<>_{CF}$ is a symmetric relationship between two transitions that ensures two transitions could be executed correctly in the same clock cycle in a clock-synchronous implementation. Given T_a and T_b are both applicable in state s, $T_a <>_{CF} T_b$ implies that

1. applying one transition before the other does not cancel the applicability of the other (i.e., $\pi_{T_a}(\delta_{T_b}(s)) \wedge \pi_{T_b}(\delta_{T_a}(s))$), and

(a)	T_1:	A==0 $\rightarrow$ B:=1
	T_2:	C==0 $\rightarrow$ D:=1
(b)	T_3:	true $\rightarrow$ B:=A
	T_4:	true $\rightarrow$ A:=1
(c)	T_5:	true $\rightarrow$ B:=A
	T_6:	true $\rightarrow$ A:=B

Figure 2. Examples of (a) $<>_{CF}$, (b) $<_{SC}$, and (c) conflicting rules. (assume A, B, C, D are boolean registers.)

2. the two transitions can be applied in either order, in two successive steps, to produce the same final state (i.e., s'=$\delta_{T_b}(\delta_{T_a}(s))$=$\delta_{T_a}(\delta_{T_b}(s))$).

On the other hand, $<_{SC}$ is an asymmetric relationship between two transitions that also ensures two transitions can be correctly executed in the same clock cycle. The $<_{SC}$ relationship is less strict than $<>_{CF}$ in that it only requires the concurrent execution to agree with one order of execution. Figure 2 gives examples of $<>_{CF}$, $<_{SC}$, and conflicting transitions that cannot be executed in the same clock cycle. In these examples, we write a transition $\langle\pi,\alpha\rangle$ concretely as

$\pi \rightarrow R_1$:=exp_1; ... R_{NR}:=exp_{NR};

In this notation, we omit registers whose action is 'ϵ' from the register-action list. In the $<>_{CF}$ example (a), T_1 and T_2 read and write two disjoint sets of registers. In the $<_{SC}$ example (b), T_3 and T_4 have a read-write dependence on register A, but concurrent execution of T_3 and T_4 gives the same result as if T_3 is applied before T_4 in sequence. In the conflicting example (c), a circular dependence between T_5 and T_6 prevents the two transitions from producing a valid result if executed concurrently.

Formal definitions of $<_{SC}$ and $<>_{CF}$ are given in [9]. The same reference also gives theorems that states it is correct for an ATS compiler to devise an arbitration logic that, on each clock cycle, selects an arbitrary subset of enabled transitions provided

1. each pair of transitions is related either by $<>_{CF}$ or $<_{SC}$, and
2. a partial order with respect to $<_{SC}$ exists for the selected transitions.

Bluespec compiler. The synthesis results in this paper are produced by the Bluespec Compiler (BSC) [1]. The theorems mentioned above are applied in BSC to produce highly concurrent clock-synchronous implementations from a sequentially conceived and interpreted operation-centric hardware description. The BSC synthesized implementations have two important characteristics. First, in each clock cycle, although the implementation appears to execute a number of transitions in sequence, the execution of those transitions all observe the same state values as latched on the previous clock edge. The second characteristic—a direct consequence of the first—is that the critical delay path of a multi-transition-per-cycle implementation is still determined only by the combinational delay of the single worst-case transition.

2.3. Limitations of operation-centric semantics

An enabled ATS transition is permitted, but not committed, to execute. When multiple ATS transitions are enabled together, ATS's abstract semantics allows any one transition to be selected nondeterministically. Hence, an ATS accepts multiple "correct" versions of deterministic clock-synchronous implementations. In fact, this freedom is taken by the ATS compiler to select a subset of non-conflicting transitions to execute in each clock cycle. The downside of this freedom is that although a transition is guaranteed to execute only when enabled, a transition is never guaranteed to execute on a particular clock cycle, or even at all.

Consequently, ATS lacks the ability to describe a system whose correctness depends both on functionality and the exact timing of events. In BSC, this limitation is addressed by an *assert* pragma which, when applied to a transition, informs the compiler that the transition must execute if its predicate is enabled. If the compiler cannot correctly guarantee the assertion (e.g., the user labels two conflicting transitions), then the compilation should fail with an error. This assert pragma, in essence, imparts synchronous semantics to the labeled transitions. The assert pragma has been successfully used to specify designs with synchronous behavioral constraints. In this paper, we formalize the assert pragma as committing transitions. We also develop the notation and compilation procedure to enable intuitive integration of synchronous committing transitions with the original non-committing transitions in the same representation framework.

3. Synchronous extensions in ATS+

We propose two extensions to ATS to support the specification of synchronous behavior. The current formulation of these two extensions assumes the particular synthesis strategy given in [9] and briefly described in Section 2.2. The resulting extended system is called ATS+. The first extension is a new class of transitions with committing execution semantics. The second extension adds support for synchronously delayed expressions that enable a transition's predicate and actions to read past values of state elements. Figure 3 summarizes the structure of ATS+ where it differs from ATS. As an intermediate representation, the extensions

$$
\begin{array}{lcl}
\mathcal{ATS}+ & = & \langle \mathcal{S}, \mathcal{S}^o, \mathcal{X}, \mathcal{X}_c \rangle \\
\mathcal{X}_c & = & \langle T_{c_1},...,T_{c_M} \rangle \\
T_c & = & \langle \pi, \alpha \rangle \\
exp & = & \text{... "ATS expressions" ...} \\
 & & \| \; exp[exp_t] \\
 & & \| \; exp[\oplus : exp_t..exp_t]
\end{array}
$$

Figure 3. ATS+ summary

in ATS+ enable analogous extensions to be introduced in the source-level languages.

3.1. Committing transitions

The definition of ATS+ includes a new class of committing transitions $\mathcal{X}_c$=$\langle T_{c_1},....T_{c_N} \rangle$. A committing transition has the same structure $\langle \pi,\alpha \rangle$ as the original ATS transitions, which we now refer to as non-committing transitions. In our examples, we write committing transitions with the symbol '$\rightarrow_c$' to differentiate from non-committing transitions. Like non-committing transitions, synchronous transitions obey the invariant that they only execute when their predicate is true. Furthermore, the execution semantics of committing transitions remains atomic. Like ATS, the execution of an ATS+ implementation must still correspond to an interleaving of atomic transitions (both committing and non-committing), where each transition leads to a state that enables the predicate of the next transition.

The sequential interleaving of transitions in ATS+ is, however, sectioned into clock periods that each contains one or more transitions. The clock periods correspond to the real clock in a synthesized clock-synchronous implementation. If a committing transition's predicate is satisfied at the start of a clock period, then it must be executed in that clock period. Consequently, for a valid ATS+, the set of committing transitions must be

1. pairwise $<>_{ME}$[1], $<>_{CF}$ or $<_{SC}$, and
2. have a partial-order with respect to $<_{SC}$.

These two conditions ensure all committing transitions that could potentially be enabled in the same clock cycle in an implementation can be executed concurrently as required by their committing semantics. On the other hand, if a non-committing transition is enabled in a clock cycle but conflicts with another enabled committing transition, the non-committing transition can always be correctly deferred to the next clock cycle.

[1]$T_a <>_{ME} T_b$ implies T_a and T_b have mutually exclusive predicate conditions, i.e., $\forall s \; \neg(\pi_{T_a}(s) \wedge \pi_{T_b}(s))$.

3.2. Synchronously delayed expressions

ATS+ with committing transitions has the ability to specify all synchronous hardware behaviors. Unfortunately, describing synchronous behaviors that span multiple clock periods can be tedious because a committing transition can only relate states that are separated by one clock edge. Synchronous behaviors that span multiple clock periods must be constructed using a sequence of committing transitions as basic building blocks. Below we introduce two syntactic shorthands that simplify the specification of multiple-cycle synchronous behavior in ATS+.

"was" expressions. According to the atomic execution semantics, a transition reads the state values before instantaneously updating all state elements. In the synthesized clock-synchronous implementation described in Section 2.2, this atomicity is preserved for concurrently executing transitions in the same clock cycle, even though all transitions read the same state value latched on the previous clock edge. This allows us to define a synchronously delayed expression *exp[t]* where *(t$\geq$ 0)*. *exp[t]* refers to the value of *exp* evaluated *t* clock cycles ago (in the degenerate case, *exp[0]* is effectively *exp*).

"interval" expressions. Another shorthand is *exp[$\oplus$:t..t']* where *t' $\geq$ t $\geq$ 0* and *exp[$\oplus$:t..t']* means *((...((exp[t']$\oplus$exp[t'-1])$\oplus$exp[t'-2])$\oplus$...)$\oplus$exp[t])*.

The user can specify any reduction operator $\oplus$ appropriate for the datatype of exp. For example, if X is a Boolean expression, X[&:1..5] is true if X is true for all five previous cycles; X[|:1..5] is true if X is true for at least one of the last five cycles.

Example. We show a trivial example to further illustrate the syntax and semantics of ATS+ extensions. Additional examples are given in Section 5. Consider T_1 and T_2,

T_1: X[10] & !(Y[&:0..10]) $\rightarrow_c$ A:=B
T_2: !X $\rightarrow$ B:=A

X, Y, A and B are Boolean registers. T_1 is a committing transition that enforces the synchronous behavior which says "A gets B's value if X was true 10 clock cycles ago and Y has not been true within the last 10 clock cycles." T_2 is a non-committing transition that says "if X is *false* then B *should* get A's value." However, T_2 only executes in a clock cycle when the predicate of T_1 is false, since T_1 and T_2 conflicts in their actions. If T_2 were written with '$\rightarrow_c$' as a committing transition, then T_1 and T_2 together would not be valid.

4. ATS+ compilation

Our goal is to map an ATS+ description to a clock-synchronous Verilog RTL description. Our ATS+ compiler is a meta-compiler layered on top of the Bluespec Compiler (BSC). As shown in Figure 4, the ATS+ compiler compiles

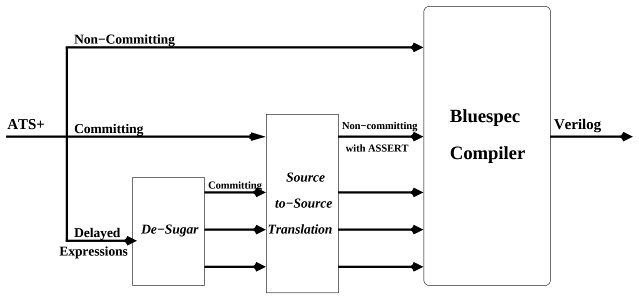

Figure 4. Compilation overview

an ATS+ description into the equivalent ATS description by

1. *de-sugaring* synchronously delayed expressions, and
2. translating committing transitions to Bluespec's non-committing transitions with the assert pragma.

The resulting ATS description is sent to BSC for conflict analysis and then synthesis to Verilog RTL. We use commercial tools to complete the synthesis flow below the RTL level.

4.1. Delayed expressions

The first pass of ATS+ compilation expands delayed expressions into a set of committing transitions without delayed expressions.

"was" expressions. A *was* expression, $exp[t]$, refers to the value of exp at t cycles ago. First consider the transition,

T_0: *pred*$[\![exp[t]]\!] \rightarrow_c$ *action*

where "*pred*$[\![exp[t]]\!]$" means *exp[t]* appears in the boolean *pred* expression. Assuming (t>0), the ATS+ compiler expands this transition to

T_1: *true* $\rightarrow_c R_t$:=R_{t-1}, ..., R_2:=R_1,R_1:=*exp*
T_2: *pred*$[\![R_t\ /\ exp[t]]\!] \rightarrow_c$ *action*

where *pred*$[\![R_t\ /\ exp[t]]\!]$ means *pred* with all occurrences of *exp[t]* replaced by R_t. (If *t==0*, *exp[0]* simply becomes *exp*.)

T_1 creates the equivalent of a shift register chain of length *t* to hold the delayed values of *exp* from the last *t* cycles. In T_2, all references to *exp[t]* in *pred* have been replaced by explicit references to R_t. The expansion is analogous for a *was* expression in the actions of a transition.

"interval" expressions. Next consider a transition that uses an *interval* expression *exp[*$\oplus$*: t..t']* in its predicate, that is

T_0: *pred*$[\![exp[\oplus : t..t']]\!] \rightarrow_c$ *action*

Assuming *(t>0)*, the expansion of this transition is

T_1: *true*$\rightarrow_c R'_t$:=$R_{t'-1}$,...,R_t:=R_{t-1},...,R_2:=R_1,R_1:=*exp*
T_2: *pred*$[((...((R_{t'} \oplus R_{t'-1}) \oplus R_{t'-2}) \oplus ...) \oplus R_t)\ /$
$(exp[\oplus : t..t'])] \rightarrow_c$ *action*

Again T_1 creates the equivalent of a shift register chain with length *t'* to hold the delayed values of *exp*. T_2 replaces the occurrences of *exp[*$\oplus$*:t..t']* in the *pred* by a direct reduction of the delayed values between *t* and *t'*. The *(t==0)* case requires a trivially different special case.

4.2. Committing transitions

The delayed expressions in an ATS+ are expanded one at a time until only committing and non-committing transitions without delayed expressions remain. In the second pass of ATS+ compilation, every committing transition is recast into a non-committing transition annotated by BSC's *assert* pragma. The committing transitions need to be checked for validity according to the requirement posed in Section 3.1 (i.e., the committing transitions should be pairwise $<>_{ME}$, $<>_{CF}$, or $<_{SC}$). By design, this validity check coincides with BSC's existing check for the scheduling of *asserted* transitions. If BSC cannot schedule all of the asserted transitions, it will generate an error. The committing transitions generated by the ATS+ compiler during the delayed expression expansion cannot change the validity of an ATS+. However, because BSC's conflict analysis is a conservative approximation, it is possible for a valid set of committing transitions to fail compilation. This issue is discussed as the latter of the two optimizations described next.

4.3. Optimizations

Interval optimization. For delayed intervals with large upper bounds, the shift register chain introduces a high area overhead. A Boolean interval expression using the reduction operator '&' can be optimized by computing the reduction using a saturation counter. This optimization replaces a (*t'-t+1*)-bit section of the shift register chain with a $\lceil$*log*$_2$ *(t'-t+2)*$\rceil$-bit counter. This optimization is expressed in the expansion below. Given

T_0: *pred*$[\![exp[\&: t..t']]\!] \rightarrow_c$ *action*

the optimized expansion assuming *(t>0)* is

T_1: *!exp* $\rightarrow_c R_{ctr}$:= *(t'-t+1)*
T_2: *exp* & R_{ctr}*!=0* $\rightarrow_c R_{ctr}$:= R_{ctr} *- 1*
T_3: *pred*$[\![(R_{ctr} == 0)[t-1]/(exp[\& : t..t'])]\!] \rightarrow_c$action

R_{ctr} is 0 only if *exp* has been true for the previous *(t'-t+1)* consecutive cycles. Thus, *exp[&:t..t']* equals *(*R_{ctr}*==0)[t-1]*. Again, the *(t==0)* requires a trivially different special case.

T_1: X & Y $\rightarrow_c$ A := B
T_2: X & !Y $\rightarrow_c$ B := A

T_3: (X & Y)[3] $\rightarrow_c$ A := B
T_4: (X & !Y)[3] $\rightarrow_c$ B := A

T_5: X[3] & Y[3] $\rightarrow_c$ A := B
T_6: X[3] & !Y[3] $\rightarrow_c$ B := A

Figure 5. Examples where delayed expressions obscure dependence

Conflict analysis. First, consider T_1 and T_2 in Figure 5. BSC's conflict analysis can easily recognize that the two transitions have mutually exclusive predicates on the value of Y. Therefore, T_1 and T_2 are valid and synthesizable together even though their actions are neither $<>_{CF}$ nor $<_{SC}$. Next consider T_3 and T_4 with predicates that are mutually exclusive delayed expressions. In this case, after de-sugaring, the actions of T_3 and T_4 are each predicated by values of delay registers that corresponds to the values of (X&Y) and (X&!Y) from 3 cycles ago. In this case, BSC would fail to recognize that the delayed predicate values are mutually exclusive since they appear to be coming from two uncorrelated registers. T_3 and T_4 would not be synthesizable by BSC although they are valid together.

To fully expose the data dependencies between transitions to BSC, the ATS+ compiler *pushes* the delay of an expression down into its variables, as shown in transitions T_5 and T_6. After common sub-expression elimination, the expression Y[3] in the de-sugared version of both T_5 and T_6 would be replaced by references to the same delay register. Thus, BSC would be able to deduce $T_5 <>_{ME} T_6$, and successfully synthesize these two transitions together. However, the push transformation increases the number of shift-register chains in a design since each delayed variable uses its own chain. Therefore, BSC is actually invoked twice by our ATS+ compiler. First, *pushed* transitions are compiled by ATS+ compiler and then BSC to enable the most precise conflict analysis possible. To produce the more efficient implementation, the ATS+ compiler and BSC are invoked again with the unmodified transitions, annotated with pragmas to override BSC's analysis.

5. Results

In this section, we compare ATS+ to hand-coded Verilog RTL, and the synchronous language Esterel [5] in terms of ease of description. We compare ATS+ synthesis quality to Verilog RTL synthesis. The examples used include standalone primitive statements as well as small examples.

Table 1. Area in um^2 after synthesis and layout

Examples		Verilog	ATS+
1	true $\rightarrow_c$ B:= A[5]	344	344
2	true $\rightarrow_c$ B:= A[&:5..25]	1688	1688
3	Example 2 (optimized)	979	995
4	Shared Token 1	1020	1036
5	Shared Token 2	7897	8106

Table 2. Cycle time in nanoseconds after synthesis and layout

Examples		Verilog	ATS+
1	true $\rightarrow_c$ B:= A[5]	0.14	0.14
2	true $\rightarrow_c$ B:= A[&:5..25]	0.15	0.15
3	Example 2 (optimized)	0.15	0.14
4	Shared Token 1	0.15	0.16
5	Shared Token 2	0.17	0.18

In Tables 1 and 2, we report the synthesized area and cycle time of ATS+ and RTL Verilog examples. The results are generated using Synopsis Design Compiler for a commercial 0.18um standard cell library. In these examples, the quality of the circuits generated from ATS+ description through BSC and from hand-coded Verilog are very similar. In the first two examples based on standalone primitives, the circuits synthesized from ATS+ through BSC are essentially identical to those synthesized directly from hand-coded RTL and therefore have the same area and cycle time. In addition, we synthesized the interval expression example both with and without the optimization discussed in Section 4.3. We compare the result to a Verilog counterpart that has been manually converted to use counters. The counter optimization achieves the expected area reduction. The degree of impact from this optimization is a function of the interval length.

Delayed expressions primitives. We first compare how simple ATS+ delayed expressions would be expressed in Verilog RTL and Esterel. The two primitive statements are

1. *was* expression: true $\rightarrow_c$ B:= A[5]
2. *interval* expression: true $\rightarrow_c$ B:= A[&:5..25]

Equivalent statements in Esterel and Verilog are given in Figure 6.[2] In both examples, because neither Esterel nor

[2]To conserve space, we omit the matching *end* statements in Esterel and Verilog examples.

```
ATS+
true →c B:=A[5];

Esterel
loop
  present pre(A)
    then emit A1 end;
  present pre(A1)
    then emit A2 end;
    :
  present pre(A4)
    then emit B end;
  pause;

Verilog
  always @(clk) begin
    B   <= A4;
    A4  <= A3;
    :
    A1  <= A;
```

```
ATS+
true →c B := A[&: 5..25];

Esterel
loop
   trap T in
     repeat 20 times
       pause;
       present A else exit T end;
     loop
       pause;
       present A then emit X
         else exit T end
 || loop
     present pre(X) then emit X1 end;
     present pre(X1) then emit X2 end;
     :
     present pre(X4) then emit B end;
     pause;

Verilog
   always @(clk) begin
     B   <= A3
     A3  <= A2;
     :
     A1  <= (ctr==0);
     ctr  <= A ? (ctr ? ctr - 1 : 0 ) : 21 ;
```

Figure 6. Code comparison of delayed expressions primitives

```
ATS+
  ReqA[&:0..10], Token !=A →c Token := A;
  ReqB[&:0..10], Token !=B →c Token := B;
  ReqA → Token := A;
  ReqB → Token := B;

Esterel
  loop
    trap T in
      repeat 10 times     pause; present ReqA else exit T end;
      end repeat;
      loop
        pause; present ReqA then emit TenReqAs else exit T end;
  || (Similar Loop for TenReqBs)
  || loop
    present TenReqAs then
      if not(?Token = 0) then pause; emit Token(0)
      else present TenReqBs then
        if not(?Token = 1) then pause; emit Token(1)
        else present ReqA then pause; emit Token(0)
          else present ReqB then pause; emit Token(1)
 Verilog
 always @(posedge CLK) begin
      Token <= Tokennxt;
      if (ReqA) ReqActr <= ReqActr + 1;
      else ReqActr <= 0;
      if (ReqB) ReqBctr <= ReqBctr + 1;
      else ReqBctr <= 0;
 always @(*) begin
    if ((ReqActr == 10) && (Token != 0)) Tokennxt = 0;
    else if ((ReqBctr == 10) && (Token != 1)) Tokennxt = 1;
    else if (ReqA) Tokennxt = 0;
    else if (ReqB) Tokennxt = 1;
    else Tokennxt = Token;
```

Figure 7. Code comparison of arbiter in shared token 1.

T_0: true $\rightarrow_c$ Token := (Token + 1) % 3;
T_1: Token==0, Req0 $\rightarrow_c$ Ack:=001, Free:=False;
T_2: Token==1, Req1 $\rightarrow_c$ Ack:=010, Free:=False;
T_3: Token==2, Req2 $\rightarrow_c$ Ack:=100, Free:=False;
T_4: Free, Req0 $\rightarrow$ Ack:=001, Free:=False;
T_5: Free, Req1 $\rightarrow$ Ack:=010, Free:=False;
T_6: Free, Req2 $\rightarrow$ Ack:=100, Free:=False;

Figure 8. ATS+ description of the arbiter in Shared Token 2.

Verilog support references to values more than one cycle ago, the equivalent expression has to be explicitly constructed from single-cycle primitives.

Shared token example 1. This example is based on an arbiter that manages the sharing of a common resource between two clients with the help of a token. A client is allowed to use the resource if it logically possesses the token. A client must maintain its request signal until it is granted the token. This example has a hard synchronous requirement that a client must be granted a token within 10 cycles or less. Excerpts of descriptions for this arbiter in ATS+, Verilog and Esterel are shown in Figure 7.

The ATS+ description specifies the above requirements with four transitions. The first two are committing transitions that specify the synchronous requirement, i.e., if a client has been waiting for 10 cycles, then the token must be taken from the other client. The last two non-committing transitions say a client may get the token upon request, provided there are no other committing or non-committing transitions that conflict with the token grant.

The equivalent Esterel description consists of three parallel loops. We use the first two loops to keep track of ten successive requests. It is important to point out that since both Esterel and Verilog are deterministic, their descriptions must fix a priority for what happens if both clients request the token but neither has waited without a token for 10 cycles.

Shared token example 2. We present another shared-token arbiter. Only excerpts of the ATS+ description are shown in Figure 8. This arbiter handles requests from three clients for a shared resource and uses a circulating token that cycles among the clients to determine priority. The first transition circulates the token among the three clients. The next three committing transitions indicate that a request from the client who currently holds the token will be granted right away. The final three non-committing transitions handle granting the request to a client without a token if the token holder is not requesting in the same cycle. Non-committing transitions are ideal in these latter scenarios since these events do not have hard synchronous requirements. This exemplifies the use of non-determinism to unburden the designer of “don’t care” decisions.

6. Related work

ATS is an high-level hardware representation based on operation-centric state transitions. ATS descriptions are not behavioral descriptions [4] in that an ATS description is not a procedural description with sequential control flow (as in C or behavioral Verilog.) ATS is developed as an intermediate representation for the compilation of TRSpec [9] and is currently also used to support Bluespec [1]. Our synthesis flow uses the Bluespec operation-centric language and compiler.

The ATS model of computation is inspired by Term Rewriting Systems (TRS) [2], a well-known reduction formalism with lineage from Lambda calculus. ATS is also similar to Dijkstra’s guarded commands [7]. Similar semantics has also served as the basis of parallel programming languages [6], hardware description languages for synchronous and asynchronous design synthesis [13, 15, 16], and languages for hardware design verification [12, 14].

ATS+ supports the natural specification of synchronous and asynchronous behaviors in the same framework. The synchronous subset of ATS+ is a simple synchronous language. “Synchronous languages” [3] refer to a class of formal specification/programming languages that are exemplified by Esterel [5], Lustre [8] and Signal [11]. Synchronous languages typically offer 1. a discrete model of time, 2. *explicit* expressions of concurrency, and 3. a *deterministic* compiled behavior.

7. Conclusions

In this paper, we presented committing transitions and synchronously delayed expressions as two synchronous extensions to ATS. These two new synchronous language elements enable ATS+ to capture both synchronous and asynchronous behaviors in the same hardware description. The intent is to allow the original ATS’s simplifying atomic and sequential semantics to assist in the description of complex concurrent internal behaviors, and in the same description, the synchronous extensions are used to describe synchronous interfaces to external modules or internal synchronous IP blocks.

We described the compilation of the synchronous extensions using existing ATS synthesis capabilities (i.e., BSC). Namely, synchronously delayed expressions are expanded into automatically generated committing transitions, and both the user and the generated committing transitions are translated into non-committing transitions annotated by BSC’s assert pragma. In our evaluation, we compared ATS+ to Verilog RTL and Esterel in terms of ease of description and to Verilog RTL in synthesis quality. We show that ATS+ enables compact descriptions of complex synchronous and asynchronous behavior and permits efficient synthesis to clock-synchronous implementations.

Acknowledgements

We like to thank Bluespec, Inc. for providing access and support to the Bluespec language and compiler.

References

[1] L. Augustsson, et al. Bluespec: Language definition. http://www.bluespec.org, 2001.

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

[3] A. Benveniste, et al. The synchronous languages twelve years later. *Proc of IEEE*, 91(1), Jan 2003.

[4] R. A. Bergamaschi. Behavioral synthesis: An overview. *TR 20944, IBM T.J. Watson Research Center*, Aug 1997.

[5] G. Berry. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, chapter The Foundations of Esterel. MIT Press, 2000.

[6] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. In *Comm of ACM*, volume 18(8), Aug 1975.

[8] N. Halbwachs, et al. The synchronous data flow programming language LUSTRE. *Proc of IEEE*, 79(9), Sept 1991.

[9] J. C. Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD thesis, Massachusetts Institute of Technology, Jun 2000.

[10] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *International Conference on Computer Aided Design*. IEEE, July 2000.

[11] P. Le Guernic, et al. Programming real-time applications with SIGNAL. *Proc of IEEE*, 79(9), Sept 1991.

[12] T. Lee et al. Automatic verification of asynchronous circuits. *IEEE Design and Test of Computers*, 12(1), Spring 1995.

[13] A. P. Ravn and J. Staunstrup. Synchronized transitions. *TR AAR-219, University of Aarhus*, 1987.

[14] V. M. Rodrigues and F. R. Wagner. Synchronous transitions and their temporal logic. In *Proc. Workshop de Métodos Formais*, 1998.

[15] J. Staunstrup and M. R. Greenstreet. From high-level descriptions to VLSI circuits. *BIT*, 28(3), 1988.

[16] J. Staunstrup and M. R. Greenstreet. *Formal Methods for VLSI Design*, chapter 2. Kluwer, 1994.