ISSN 1346-5597

National Institute of Informatics

NII Technical Report Feb 2003

Synthesizing Timed Circuits from High Level Specification Languages

Tomohiro Yoneda and Chris Myers

NII-2003-003E
Feb 2003

URL: http://www.nii.ac.jp/ Tokyo 101-8430, JAPAN +81-3-4212-2000 FAX: +81-3-3556-1916

Synthesizing Timed Circuits from High Level Specification Languages

Tomohiro Yoneda*
National Institute of Informatics
yoneda@nii.ac.jp

Chris Myers†
University of Utah
myers@ee.utah.edu

Abstract

This work proposes an efficient methodology to synthesize timed circuits from high level specification languages. In particular, this paper presents a systematic procedure for translating channel-level models to time Petri net descriptions. Care is taken in this translation to guarantee that there are no state coding violations in the resulting nets greatly simplifying the synthesis process. This paper also presents a modular decomposition method to break up the circuit to be synthesized such that an efficient partial order based synthesis approach can be applied to rapidly produce a circuit implementation. This new synthesis technique is demonstrated by its application to the line fetch module from the TITAC2 instruction cache system.

Key Words: *High level synthesis, timed circuits, partial order reduction, time Petri nets, Balsa*

1 Introduction

When synthesizing large and practical designs, the use of high level specification languages is essential. For synchronous circuit design, circuits are often derived from behavioral level specifications expressed in VHDL or Verilog. For asynchronous circuit design, it is particularly important to hide handshake level descriptions in order that non-experts in asynchronous circuit design can easily design practical asynchronous circuits. The goal of this paper is to develop an efficient methodology to synthesize asynchronous circuits from high level specification languages.

There are two main approaches to the synthesis of asynchronous circuits: logic synthesis [1, 2, 3] and syntax directed translation [4, 5, 6, 7]. In the logic synthesis method, a logic function for every output signal is derived from a low level specification language such as an STG or burst-mode state machine. These methods often require an enumeration of the state space which can be quite expensive for large systems. In the syntax directed method, instead of enumerating the state space, each construct of the specification language is directly modeled by a circuit. While the logic synthesis method can potentially synthesize higher performance and more optimized circuits than the syntax-directed method, the state explosion problem makes it extremely difficult to apply this method to large specifications expressed in a high level language. Another problem with the logic synthesis method is that it is typically quite expensive to perform state assignment for large specifications (the so called *complete state coding* (CSC) problem for speed-independent circuits). Automatic approaches to address state assignment often take a lot of time and can generate redundant and slow circuits. The assistance of designers is often needed to obtain better circuits, but it is tedious and requires deep knowledge of asynchronous circuit design. Therefore, most of synthesis methods that can derive non-trivial circuits from high level specification languages are based on syntax directed translation. Tangram [5, 6] and Balsa [7] are two well-known design methods for this class. Although it is known that the circuits derived by those methods can be inefficient, recent papers [8, 9] report that their methods can synthesize faster circuits than `Petrify`, a logic synthesis based tool. This is mainly due to the inappropriate insertion of internal variables in order to avoid CSC problems.

The goal of this paper is to extend the applicability of logic synthesis methods to high level specifications by applying ideas to avoid the state explosion and CSC problems. It is also very important to aggressively use timing information whenever possible to produce efficient circuit implementations [10, 3]. For example, the method described in [9] uses relative timing constraints to optimize their designs. The process to find the paths to be compared for the relative timing method is, however, expensive to automate. This work aims at using bounded delay information to obtain optimal circuits with respect to this timing information automatically.

This paper proposes the following approach.

1. Our method translates a high level specification language to a (time) Petri net. This Petri net is guaranteed

*This research is supported by JSPS Joint Research Projects.

†This research is supported by NSF Japan Program award INT-0087281 and SRC grant 2002-TJ-1024.

to be CSC conflict free by inserting internal variables based on the semantics of the language. This insertion of CSC variables may not be optimal, but is supposed to be as good as or better than syntax directed methods.

2. An improved modular synthesis method using partial order reduction is applied to this Petri net. In the original modular synthesis method [11, 12], the whole module given by a user is the target for reduced state space enumeration. Thus, only interleavings on the variables internal to the environment of this module are avoided. The method proposed in this paper automatically decomposes the given module to obtain many subcircuits that each contain only one output signal, and applies the modular synthesis method to each subcircuit. Since the number of interface signals in such subcircuits is much smaller than that of the whole module, significant improvement is obtained.

3. The proposed method synthesizes timed circuits using the given bounded delay assumption. Since it explores the timed state space, optimal circuits with respect to the given timing assumptions are automatically derived without additional verification needed as in the case of relative timing assumptions.

In order to demonstrate the proposed method, the line fetch module of TITAC2's [13] instruction cache system is described using the Balsa language, and a whole circuit including the data path is synthesized by the proposed method. The obtained circuit is compared with that obtained by the Balsa system using a Verilog simulation. `Petrify` failed to synthesize this circuit from the untimed version of the same Petri net due to BDD nodetable overflow after 35 hours of CPU time by a workstation with a 2.8GHz Pentium 4 and 4 gigabytes of memory. The translation from the Balsa language to our intermediate language is currently done by hand. The rest of the synthesis to obtain a Verilog gate level net-list is, however, automated (but naively). The proposed method is easily applied to other high level specification languages by modifying the translation algorithm to the intermediate language.

There are many related works [14, 15, 16, 17, 18, 19]. For example, [16] derives an STG from a handshake circuit description, while our approach uses a high-level description directly. Furthermore, their STG may suffer CSC problems. [17, 18] propose limited clustering and peephole optimization. They can avoid run-time problems by handling only small clusters, and still obtain performance improvement. Our method can handle larger clusters with reasonable run-time by the modular synthesis and automatic decomposition algorithm. None of these works handles timing information to optimize the circuit. [19] shows a protocol search technique to derive an optimal timed circuit from high level specification. It may suffer run-time problem for large systems, but his technique and our decomposition technique could potentially compliment each other.

The rest of this paper is organized as follows. Section 2 presents the basic constructs of a high specification language. Section 3 describes the synthesis method. Section 4 reviews the partial order synthesis approach. Section 5 proposes our modular decomposition technique. Finally, Section 6 presents experimental results, and Section 7 gives our conclusions.

2 High level specification languages

The specification languages considered in this work are supposed to have the following basic sentences for control.

- Handshake Read and Write for ports.
- Sequential sentence $X \equiv Y_1; Y_2; \cdots; Y_n$.
- Parallel sentence $X \equiv Y_1 || Y_2 || \cdots || Y_n$.
- Loop sentence $X \equiv \mathsf{loop}\ Y\ \mathsf{end}$.
- While sentence $X \equiv \mathsf{while}\ cond\ \mathsf{then}\ Y\ \mathsf{end}$.
- If sentence $X \equiv \mathsf{if}\ cond\ \mathsf{then}\ Y\ \mathsf{else}\ Z\ \mathsf{end}$.

Our approach is not restricted to some specific specification language, but it can be applied to any high level specification languages that are based on the above control sentences. This paper, however, uses the Balsa language for demonstration. To this end, the following sentence is also considered.

- Select sentence $X \equiv \mathsf{select}\ a\ \mathsf{then}\ A \mid b\ \mathsf{then}\ B\ \mathsf{end}$.

This operation first waits for the request signal from data port a or b. Thus, it is a kind of a passive read operation. It can, however, execute a sentence or a block of sentences before issuing the acknowledgement for the data port. Since the data is kept on the port before the acknowledgement, the sentences executed by the Select sentence can use that data without latching it. This simplifies the circuit and reduces the delay for latching, if the sender process can stall without loss of performance.

As an example, this paper considers the line fetch module from the TITAC2 instruction cache system [13]. The TITAC2 instruction cache memory contains 256 line (or block) frames, and each line frame contains 8 words. Thus, the size of the cache memory is 8KB (256 line frames $\times$ 8 words $\times$ 32 bits). The lines are direct mapped and fetched in the early restart manner with critical word first [20]. So, when a word with address adr is read and it is not in the cache memory, the line containing the word is fetched in the

```
#define PC_OFFSET ((PC as array 0..31 of bit)[2..4] as 3 bits)
#define PC_LINE_ADR ((PC as array 0..31 of bit)[5..12] as 8 bits)
#define PC_LINE_FRAME_ADR ((PC as array 0..31 of bit)[5..31] as 27 bits)
#define CNT_LINE_ADR ((CNT as array 0..31 of bit)[5..12] as 8 bits)
#define CNT_TAG ((CNT as array 0..31 of bit)[13..31] as 19 bits)
#define CNT_LINE_FRAME_ADR ((CNT as array 0..31 of bit)[5..31] as 27 bits)

procedure lineFetch (
  input PC : 32 bits;
  sync lineFetchComplete;
  output inst_out : 32 bits;
  output TagWrite : tagmCtrlType; -- record addr:8 bits; wr_data:19 bits end
  output existWriteAddr : 3 bits;
  sync existReset;
  output mmem_rd_addr : mmemRdAddrType; -- record off:3 bits; addr:27 bits end
  input mmem_rd_data : 32 bits;
  output cmem_wr_ctrl : cmemCtrlType; -- record off:3 bits; addr:8 bits; wr_data:32 bits end
  output cbuf_wr_ctrl : cbufCtrlType -- record addr:3 bits; wr_data:32 bits  end
) is
local variable addr_counter, add_tmp : 3 bits
      variable count, count_tmp : 3 bits
      variable mbuf : 32 bits
      variable CNT : 32 bits
begin
  loop
    select PC then
      sync existReset ||
      addr_counter := (PC_OFFSET + 1 as 3 bits) ||
      count := (7 as 3 bits) ||
      begin
        mmem_rd_addr <- (mmemRdAddrType {PC_OFFSET, PC_LINE_FRAME_ADR}) ||
        select mmem_rd_data then mbuf := mmem_rd_data end;
        inst_out <- mbuf ||
        cmem_wr_ctrl <- (cmemCtrlType {PC_OFFSET, PC_LINE_ADR, mbuf})||
        cbuf_wr_ctrl <- (cbufCtrlType {PC_OFFSET, mbuf});
        existWriteAddr <- PC_OFFSET
      end ||
      CNT := PC
    end;
    while count > 0 then
      add_tmp := (addr_counter + 1 as 3 bits) ||
      count_tmp := (count - 1 as 3 bits) ||
      begin
        mmem_rd_addr <- (mmemRdAddrType {addr_counter, CNT_LINE_FRAME_ADR}) ||
        select mmem_rd_data then mbuf := mmem_rd_data end;
        cmem_wr_ctrl <- (cmemCtrlType {addr_counter, CNT_LINE_ADR, mbuf}) ||
        cbuf_wr_ctrl <- (cbufCtrlType {addr_counter, mbuf});
        existWriteAddr <- addr_counter
      end;
      addr_counter := add_tmp ||
      count := count_tmp
    end ||
    TagWrite <- (tagmCtrlType {CNT_LINE_ADR, CNT_TAG});
    sync lineFetchComplete
  end
end
```

Figure 1. Balsa description for the line fetch module.

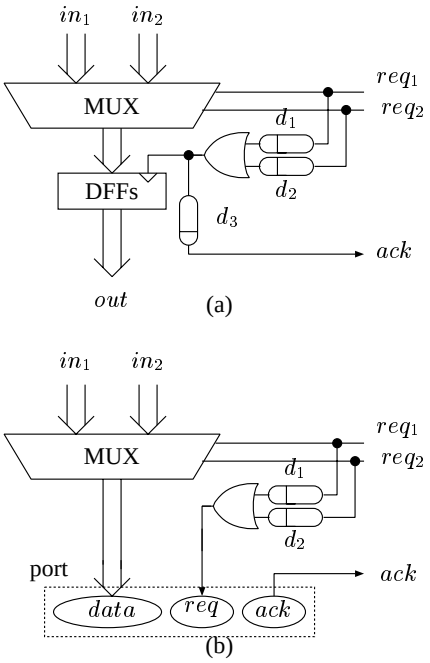

Figure 2. Variable and port implementation.

order $[adr], [adr+1], \cdots, [adr+n], [adr-m], \cdots, [adr-1]$, where $m = adr \bmod 8$ and $n = 7 - m$. Furthermore, the access for the other words within the same line are possible as soon as the words are fetched, while all the accesses for the words not in the line are suspended until the line fetch is completed even if the access is on a hit.

The Balsa description for the line fetch module is shown in Figure 1. The line fetch module starts the line fetch when the instruction address, which is on a miss, is given in port "PC". When the first word specified by port "mmem_rd_addr" is read from Main Memory through "mmem_rd_data" port, the word is sent to an instruction decoder unit through port "inst_out" as well as it is written into Cache Memory through "cmem_wr_ctrl" port and Cache buffer register through "cbuf_wr_ctrl". The line fetch module also sets the corresponding bit in the Exist Register though port "existWriteAddr", which indicates the available word in the line currently being fetched. It is used such that when the other words within this line are read, they are selected from Cache buffer register or the read operation is suspended according to the corresponding bits in the Exist Register. The line fetch module continues to fetch the remaining 7 words in the same line and updates the Cache Memory and Exist Register as well. Also, the tag information to update Tag Memory is sent to the tag memory module through port "TagWrite". Finally, the "lineFetchCompute" signal is issued.

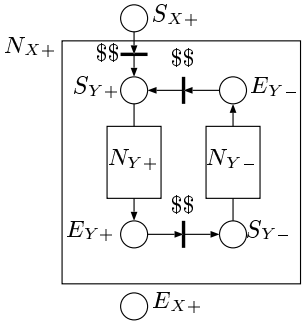

Figure 3. Petri net for a loop sentence.

3 Synthesis of circuits

This work uses the bundled data method for data path circuits and four-phase handshaking protocol for control signals. This section describes the synthesis procedure for the data path and control circuits.

3.1 Data path circuits

Variables such as "add_counter" have multiple sources. Thus, for each variable, our method generates the circuit structure shown in Figure 2(a). Each input of the multiplexer is connected to the corresponding source latch or input port either directly or through some combinational circuit, such as an incrementer, for data manipulation. The delays d_1 and d_2 are supposed to include both the delay of such a combinational circuit and the delay of the multiplexer. If either there is no combinational circuit, or the source latch or the input port becomes stable much earlier than the arrival of req_1 (req_2), then d_1 (d_2) does not need to include the delay of the combinational circuit. Furthermore, if only one source writes this variable, the multiplexer can be removed, and its delay can be omitted from d_1. Similar structures are generated for active output ports as shown in Figure 2(b). In addition, a comparator is generated for each condition in a conditional statement. These data path circuits are generated by statically analyzing the Balsa description.

3.2 Control circuits

Our method generates a control circuit using the following three steps:

Step I. Petri net generation

Step II. State graph construction

Step III. Logic minimization

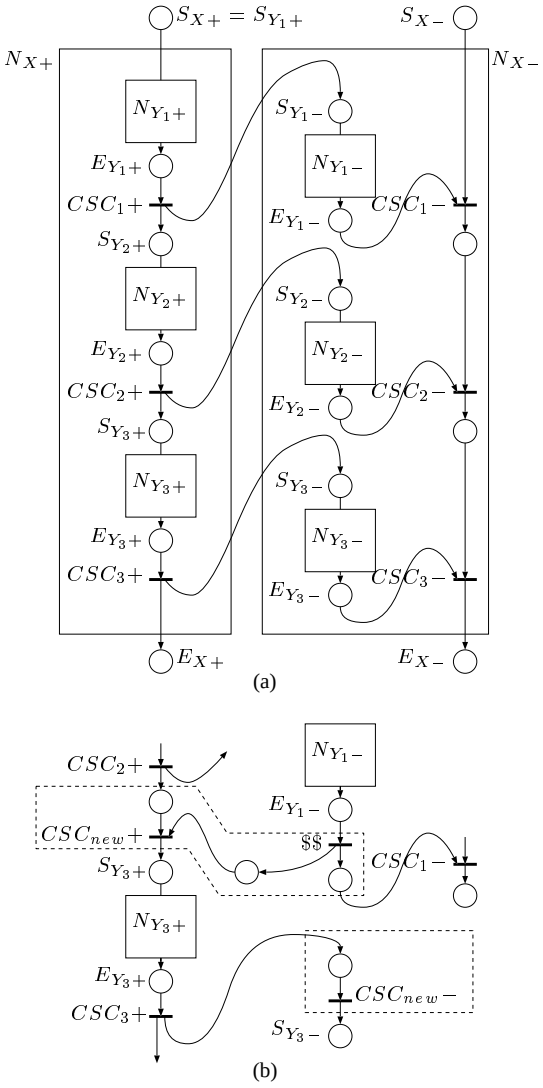

Figure 4. Petri net for a sequential sentence.

This subsection discusses the first step of the algorithm. The efficient approach for the second step is described in the next section, and the third step is achieved using ATACS [3].

The actual execution of each sentence of the specification language consists of a working phase and a resetting phase. Thus, for a sentence X, two subnets N_{X+} and N_{X-} are usually generated to represent each phase. Each subnet starts its execution when a token is put in its unique start place, and when its execution completes, a token is placed into its unique end place. Let S_{X+} and E_{X+} denote the start place and the end place of N_{X+}. Similarly, S_{X-} and E_{X-} are those for N_{X-}. For each sentence, the subnets are generated as follows.

(a) Loop sentence $X \equiv \mathsf{loop}\ Y\ \mathsf{end}$:
Since a Loop sentence never terminates, N_{X-} is empty, and E_{X+} is never reached. The subnet N_{X+} is shown in Figure 3. $\$\$$ represents a dummy (or sequencing) transition.

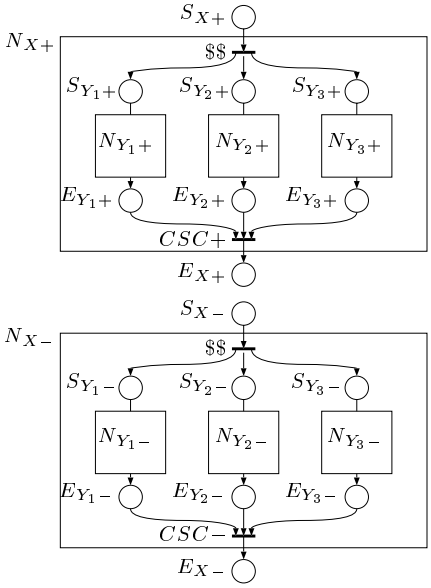

Figure 5. Petri net for a parallel sentence.

(b) Sequential sentence $X \equiv Y_1; Y_2; \cdots; Y_n$:
The subnets N_{X+} and N_{X-} for $n = 3$ are shown in Figure 4 (a). As shown in the figure, internal state variables (CSC_1, CSC_2, $\cdots$) are necessary to solve the CSC conflict. These subnets aim at maximal concurrency, i.e., the working phases and the resetting phases are maximally overlapped. The idea of this approach is the same as that of [16], which introduces, however, no CSC variables. This approach has higher concurrency than that of the concurrent sequencer in [18]. In Balsa circuits, those two phases are completely serialized. Note that N_{X-} is executed only after the execution of N_{X+}, which is guaranteed by the outer sentence containing X. Thus, CSC_1-, CSC_2-, and CSC_3- occur after CSC_3+, which avoids the CSC conflict, although the resetting phases (N_{Y_1-} and so on) may be executed earlier. On the other hand, if the same variables or ports are accessed in, for example, Y_1 and Y_3, then the resetting phase of Y_1 must complete before the working phase of Y_3, but this is not guaranteed by this subnets. Thus, it is necessary to identify such cases by static analysis of the specification, and to add a causality from N_{Y_1-} to N_{Y_3+} as shown in Figure 4(b). A new CSC variable (CSC_{new}) is necessary here. In some cases, such causality can be satisfied by timing. In those cases, these additional places and transitions are redundant. In particular, a redundant CSC variable degrades the performance of the circuit. Thus,

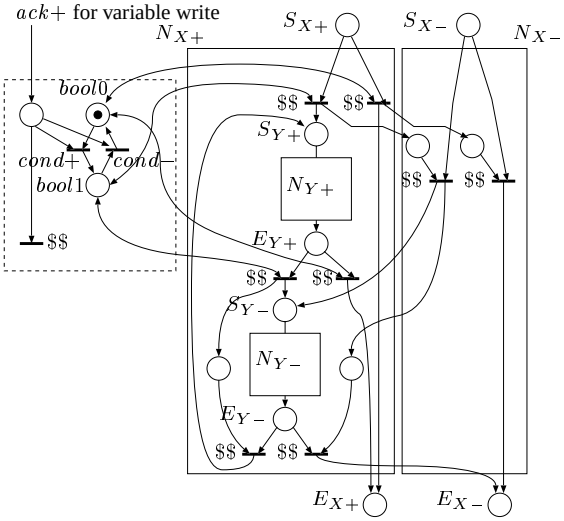

Figure 6. Petri net for a while sentence.

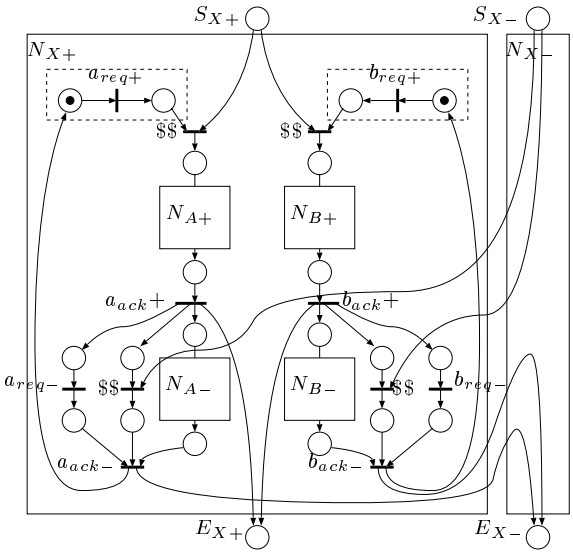

Figure 7. Petri net for a select sentence.

when timing information is given (see Section 6), replacing those CSC transitions by dummy transitions is tried as an optimization.

(c) Parallel sentence $X \equiv Y_1||Y_2||\cdots||Y_n$:
The subnets N_{X+} and N_{X-} for $n = 3$ are shown in Figure 5. If a parallel sentence is used inside a sequential sentence, the CSC transitions of Figure 5 are redundant, and they can be replaced by dummy transitions. Note that from these subnets, a C-element of n inputs is derived. Specifying $(Y_1||Y_2)||Y_3||Y_4$ avoids such the C-element with large fan-ins, because the CSC state variable decomposes the C element.

(d) While sentence $X \equiv$ while $cond$ then Y end :
The subnets N_{X+} and N_{X-} are shown in Figure 6. The part for handling the $cond$ input (inside the dashed box in the figure) is a little tricky. The state space exploration considering actual data values costs too much. Thus, instead, our subnet considers a nondeterministic behavior of the $cond$ input when the variable to be checked is written. That is, since the cond signal comes from some combinational circuit such as a comparator, and the comparator refers to some variable, the cond signal can change whenever this variable is written. Thus, when the ack signal of this variable goes high, the place from $ack+$ in the dashed box of the figure gets a token. Then, if the dummy transition inside the dashed box fires, the $cond$ input does not change, and otherwise, the $cond$ input changes. This allows the synthesized circuit to accept any possible change of the $cond$ input. If the comparator refers to more than one variable, then for each variable, a place from its ack signal and a dummy transition are generated in the dashed box.

(e) Select sentence $X \equiv$ select a then $A \mid b$ then B end:
The subnets N_{X+} and N_{X-} are shown in Figure 7. Since the request and acknowledge signals for a and b work as context signals, no CSC variables are necessary. If more than one select sentence exists for the same input signals a and b, then the places and the transitions in the dashed boxes should be shared.

The rest of the sentences are handled similarly, and the discussion for them is omitted in this paper.

4 Partial order reduction for timed circuit synthesis

In order to perform the above second step (state graph construction) efficiently, [11] proposes an approach that takes advantage of hierarchy in a specification by applying partial order reductions to modular synthesis. The idea is to consider a single interleaving of the signals that are invisible to the target subcircuit. This section reviews this idea. Note that in order to simplify the explanation, the following uses a little different notation than [11].

Let us consider a set $M = \{E, S_1, \cdots, S_i, \cdots, S_n\}$ of modules which is a closed system, where E is the environment of the whole circuit and S_i is a specification of the i-th subcircuit with I_i and O_i as the input signal set and output signal set. Our goal is to synthesize a subcircuit C_i that implements S_i for each i. Here, suppose that O_i includes internal signals for CSC variables that are introduced in the

previous section. Let *visible signals* denote those signals that are in I_i and O_i.

Let $\mathcal{T}$ be a set of all traces of M, and $s \xrightarrow{t} s' \in \mathcal{T}$ denote that the state transition from s to s' by firing t occurs in $\mathcal{T}$. $\mathsf{visible}(\mathcal{T}, S_i)$ represents a set of visible signal transitions of S_i, i.e.,

$$\mathsf{visible}(\mathcal{T}, S_i) = \{(s, s') \mid t \in I_i \cup O_i, \; s \xrightarrow{t} s' \in \mathcal{T}\}.$$

The subcircuit C_i can be synthesized from a state graph that contains only visible transitions for S_i, i.e.,

$$\mathsf{rsg}(\mathcal{T}, S_i) = \{\langle v \rightarrow v' \rangle \mid (s, s') \in \mathsf{visible}(\mathcal{T}, S_i), \; v = \mathsf{proj}(s, S_i), v' = \mathsf{proj}(s', S_i)\},$$

where $\mathsf{proj}(s, S_i)$ is a state projected to the visible signals of S_i. The traditional approach to constructing $\mathsf{rsg}(\mathcal{T}, S_i)$ is very expensive, because the possible interleavings of all transitions are considered. In the partial order reduction approach, instead of $\mathcal{T}$, a smaller set $\mathcal{T}_i$ of traces that are sufficient for synthesizing C_i is considered. Such $\mathcal{T}_i$ can be obtained by modifying the Stubborn set method [21] such that in addition to the conflicting transitions, all the transitions for the visible signals of S_i are also interleaved. Since the other concurrent invisible transitions are not interleaved, the state space searched for $\mathcal{T}_i$ is usually much smaller than that of $\mathcal{T}$. Then, for such $\mathcal{T}_i$, it can be proven that $\mathsf{rsg}(\mathcal{T}, S_i) = \mathsf{rsg}(\mathcal{T}_i, S_i)$ holds. Hence, C_i can be synthesized from $\mathsf{rsg}(\mathcal{T}_i, S_i)$, which sometimes reduces the cost of the synthesis dramatically. The works described in [11, 12] show that a one to two orders of magnitude reduction in synthesis time is possible when compared to the traditional approach.

This approach, however, still needs a lot of computational cost, if S_i has many visible signals. In the case of our example shown in Section 2, the specification has no hierarchy, and so it has more than 15 input control signals and 40 output control signals (including the control signals for latches and multiplexers as well as the CSC variables). The next section proposes a new idea to automatically decompose visible signals and obtain many small circuits, each of which contains only one output signal.

5 Modular decomposition

Since this decomposition is for applying the partial order reduction method described in the previous section, its main goal is to reduce the size of the visible signal set for each subcircuit. Suppose that a specification for the subcircuit with an output o_i is denoted by S_i, then $O_i = \{o_i\}$. As for the input signals of S_i, note that all other visible signals of the original circuit are the candidates. Unlike the usual circuit decomposition, including redundant signals in I_i does not imply that a redundant subcircuit is obtained. It just causes extra cost for the state graph construction, but it is guaranteed that the correct subcircuit is obtained, because the interleavings of invisible signals are just redundant for $\mathsf{rsg}(\mathcal{T}_i, S_i)$, e.g., the extreme case is the traditional total order approach. On the other hand, if some signal is missing in I_i, then two possible cases can occur according to the type of the missing input signal. When each transition on an output signal is implemented using a single cube, each input signal for the subcircuit can be classified as being either a *trigger* or *context* signal [22]. The trigger signals are those that change when the target output signal should be excited. But, the inverse is not always true, i.e., the trigger signal may also change at other times. The context signals, which are stable in the *excitation region* of the target output signal (i.e., the set of states in which this transition is enabled), are used to solve this ambiguity and to determine the exact condition for the output excitation. If some trigger signal is missing in I_i, then some state transitions are not considered for the circuit behavior, and it results in a wrong subcircuit. If some context signal is missing, then either a CSC conflict occurs or a correct but redundant subcircuit is derived. The latter case can occur when, due to the missing context signal, either a single cube implementation is no longer possible or non-optimal context signals are used. Although the behavior of the derived subcircuit is correct, it is desired to prevent this case, but it is now an open problem.

From the above discussion, the algorithm proposed here guarantees that all trigger signals are included in I_i and tries to include enough context signals such that no CSC conflicts occur. For a given Petri net N obtained in Subsection 3.2 and a target output out, our algorithm performs the following steps.

Step 1: From two transitions $t_{out}+$ and $t_{out}-$ of N representing the change of signal out, N is traversed upwards until either non-dummy transitions (which correspond to visible signals of the original circuit) or already traversed nodes (places or transitions) are reached, The traversal is forked when the arcs of N merges at places or at transitions. The reached visible signals include the trigger signals, and they are put into I_i.

Step 2: Construct a state graph using O_i and I_i obtained in the previous step, and check if a CSC conflict occurs or not. If not, derive a subcircuit through the logic minimization step. Otherwise, go to the next step.

Step 3: From each trigger signal that is newly obtained in Step 1 or previous iteration of Step 3, N is traversed upwards in the same way as Step 1. This step intends to find trigger signals of each newly found trigger signal x, because they may be able to be used as context

Table 1. Delay information.

Operations	Min & Max
Output signals	[1,4]
Main memory access	[100,110] ([1,10])
Tag/Cache memory access	[15,20] ([1,10])
Exist/Cache-buffer register access	[2,10] ([1,10])
Other environments	[1,10]
Next Instruction Addr.	[50,350]

(Bounds in parentheses are for reset phase)

signals that partition the states where x is enabled, and may solve the CSC conflict found in Step 2. Thus, those trigger signals are put into I_i. If I_i is modified, then go to Step 2. Otherwise, go to Step 4.

Step 4: All signals except for o_i are put into I_i, and derive a subcircuit through the logic minimization step.

It is clear that Step 1 finds all trigger signals. Step 4 is used to guarantee termination of the algorithm. Step 2 and Step 3 are heuristics to find a smaller set of context signals. In the example of Figure 1, out of 40 output signals, 32 output signals do not run Step 3, and the remaining 8 output signals run Step 3 once. No output signals need to run Step 3 more than once or Step 4. From these results, one may consider that our CSC variables introduced in Section 3 are mostly redundant, because the CSC variables are usually added as context signals for a target output, but the above results imply that the most outputs are built purely of trigger signals. This is, however, not true. It is because our CSC variable added as a context signal for an output a can be used as a trigger signal for another output b. For example, CSC_1 in Figure 4 is a context signal for sentence Y_1, but it can be used as a trigger signal for sentence Y_2.

This decomposition reduces the cost of the state graph construction dramatically as shown in the next section.

6 Experimental results

This section demonstrates the proposed idea with the example shown in Figure 1. The experiments here have been done on a 2.8 GHz Pentium 4 workstation with 4 gigabytes of memory.

From the Balsa description, two intermediate language descriptions for a data path circuit and a control circuit are obtained. This step is currently done by hand. Then, Perl scripts generate a Verilog gate-level description for the former and the Petri net for the latter as described in Section 3.

This Petri net was first given to `Petrify`. It, however, failed to synthesize the control circuit due to BDD nodetable overflow after 35 hours of CPU time. Second, in order to synthesize a timed circuit, the delay information as shown in Table 1 is assumed, and the above Petri net has been modified into the time Petri net that reflects those delays. This is mainly done by adding the delays in Table 1 to transitions that represent the corresponding signals. The delay information for output signals, Tag/Cache memory access, and Exist/Cache-buffer register access is based on the delays of 0.25μm standard cell library, which are shown in Table 3. Note that the values are scaled up by 4 because our tool accepts only integer bounds currently. For example, since an output signal is derived by a gate tree with depth less than or equal to 3, its bound [0.25,1.0] ($= [1,4]/4$) covers both a single gate delay 0.3 ns and a triple gate delay 0.9 ns. The lower bound 12.5 (= 50/4) of "Next Instruction Addr" is determined from the cycle time (17 ns) of NOP operation of TITAC2. The other bounds are set conservatively. The partial order reduction algorithm in [11] was then applied to this time Petri net which also failed due to memory overflow.

Finally, the above time Petri net is applied to the proposed method. In this case, Step II. and III. shown in Subsection 3.2 should be performed for each output signal, while the same time Petri net is used commonly. Table 2 shows the CPU times and the amount of memory used for each step for several output signals. Multiple rows shows the iteration of steps 1 and 3 of Section 5. From these results, it can be seen that significant reduction in both CPU times and amount of memory used is achieved. This is ac-

Table 2. Performance of proposed method.

Output signals	Step II.	Step III.
PC_a	0.5s / 1.5MB	–
	2.4s / 5.3MB	0.02s
$mmem\_rd_data a_a$	0.2s / 0.4MB	0.01s
$mbuf\_load1$	0.2s / 0.4MB	0.01s
$csc4$	0.2s / 0.4MB	–
	0.2s / 0.4MB	0.01s
$csc21$	5.2s / 9.3MB	–
	11.1s / 16.0MB	0.25s

Table 3. Delays for simulation.

Operations	Delays (ns)
Gates (AND, OR, C)	0.3
Latches	0.7
Muxs	0.45
Inc/Dec	1.5
Main memory access	30.0 (1.0)
Tag/Cache memory access	4.5 (1.0)
Other environments	2.0 (1.0)
Delay elements (d_1, d_2)	2.1
Delay elements (d_3)	0.9

(Delays in parentheses are for reset phase)

Table 4. Performance comparison.

Events	Balsa (ns)	Proposed (ns)
inst. addr. (PC) $\rightarrow$ inst. out	44	36
inst. addr. (PC) $\rightarrow$ line fetch complete	598	422
cycle time for main memory read	75	52

complished by using smaller sets of visible signals. For example, for an output PC_a, the number of input signals needed to solve the CSC conflict is 8, while the number of all signals for the whole line fetch circuit is 57. Although the state graph construction must be done for many times, the total CPU time for all output signals is much smaller since the state space increases exponentially in the number of the visible signals. This example takes total 24 seconds for synthesis. Another advantage is that the used memory is released after each execution run. Thus, only the maximum amount of memory used for handling each output is necessary, which is much smaller than that for all output signals.

In order to compare the performance of the circuit obtained by the proposed method with that by the Balsa system, both circuits are simulated in a gate level Verilog simulator. Since the Balsa system derives circuits using standard C-elements, our method also derives standard C-implementations. As for delays of components, the values shown in Table 3 are used.

Table 4 shows several response times for both circuits. The performance improvement is mainly obtained from the protocol improvement for the sequential sentences and the optimized circuit by logic minimization. In [18], it is reported that their resynthesis and peephole optimization improved the performance of balsa circuits by 26 % – 55 %. Our improvement ranges from 18 % to 31 %. Note, however, that our example includes several main memory access, which takes pretty long time, and thus, the improvement of the control circuit does not directly affect the total cycle times. As for the size of the control circuits, the gate (AND, OR, C) count of the balsa circuit is 171, while ours is 64. According to [17], the above optimization causes the area-overhead that ranges from 18 % to 27 %. Thus, it implies that their circuit is about three times larger than ours.

In order to see the improvement obtained by using timing information, the same synthesis steps are executed by using a modified delay information where all lower bounds in Table 1 are set to 1, and the upper bound for output signals is set to 10. In this case, 6 gates (16 literals) are added, which are 6 % (gates) and 13 % (literals) of the total. On the other hand, from the gate level simulation, the performance degradation of this redundant circuit is very small (0.5 %), probably because those additional gates are not in critical paths.

7 Conclusion

This paper presents a method for translating high level specifications to Petri net representations that have been optimized for performance and the avoidance of CSC violations. These nets also include timing information that can be utilized to further improve the performance of the circuit implementations. Finally, an automatic technique is described to decompose the circuit into individual modules to improve synthesis runtime. This technique can then make use of an efficient modular synthesis procedure based on partial order analysis. The synthesis method described in this paper is shown to be substantially more efficient than traditional logic synthesis approaches. It is also shown to produce circuits superior to those derived by syntax-directed translation method.

The advantages of the proposed method over the optimization techniques of syntax-directed methods are that timing information can be used to optimize circuits and that a protocol search technique shown in [19] combined with our single output synthesis technique can help to further optimize circuits. On the other hand, a disadvantage is that although the proposed method significantly speeds up the synthesis, its scalability to larger circuits is still unclear. Many case studies are necessary. Another drawback of the single output synthesis technique is that it may produce redundant circuits (e.g., with non-single cube implementation). The quality of the circuits and the speedup of the synthesis are a trade-off.

Future work includes automating the translation steps from some high level specification language to the datapath and control intermediate representations. This will allow us to apply our method to more case studies (including other subsystems of TITAC2) to further evaluate the utility of this approach.

Acknowledgments

We would like to thank Masashi Imai (Univ. of Tokyo) for helping delay setting for Verilog simulation.

References

[1] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for

manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[2] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[3] Chris J. Myers, Wendy Belluomini, Kip Killpack, Eric Mercer, Eric Peskin, and Hao Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, February 2001.

[4] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Advanced Research in VLSI*, pages 35–50. MIT Press, 1988.

[5] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[6] Joep Kessels and Ad Peeters. The Tangram framework: Asynchronous circuits for low power. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 255–260, February 2001.

[7] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[8] Euiseok Kim, Jeong-Gun Lee, and Dong-Ik Lee. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 104–113. IEEE Computer Society Press, April 2000.

[9] A. Bystrov and A. Yakovlev. Asynchronous circuit synthesis by direct mapping: Interfacing to environment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 127–136, April 2002.

[10] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, February 2001.

[11] T. Yoneda, E. G. Mercer, and C. J. Myers. Modular synthesis of timed circuits using partial order reduction. *Proc. of The 10th Workshop on Synthesis And System Integration of Mixed Technologies*, pages 127–134, 2001.

[12] E. Mercer, C. Myers, T. Yoneda, and H. Zheng. Modular synthesis of timed circuits using partial orders on lpns. In *Theory and Practice of Timed Systems (TPTS 2002)*, April 2002.

[13] Akihiro Takamura, Masashi Kuwako, Masashi Imai, Taro Fujii, Motokazu Ozawa, Izumi Fukasaku, Yoichiro Ueno, and Takashi Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model. In *Proc. International Conf. Computer Design (ICCD)*, pages 288–294, October 1997.

[14] P. Kudva, G. Gopalakrishnan, and V. Akella. High level synthesis of asynchronous circuit targeting state machine controllers. In *Asia-Pacific Conference on Hardware Description Languages (APCHDL)*, pages 605–610, 1995.

[15] Prabhakar Kudva, Ganesh Gopalakrishnan, and Hans Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[16] Tilman Kolks, Steven Vercauteren, and Bill Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996.

[17] Tiberiu Chelcea, Andrew Bardsley, Doug Edwards, and Steven M. Nowick. A burst-mode oriented back-end for the Balsa synthesis system. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 330–337, March 2002.

[18] Tiberiu Chelcea and Steven M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, June 2002.

[19] Eric Robert Peskin. *Protocol Selection, Implementation, and Analysis for Asynchronous Circuits*. Ph.D. dissertation, The University of Utah, August 2002.

[20] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan Kaufmann Publishers, 1996.

[21] A. Valmari. A stubborn attack on state explosion. *Proc. of Workshop on Computer Aided Verification*, 1990.

[22] Chris Myers. *Asynchronous Circuit Design*. John Wiley & Sons, 2001.