

High Level Synthesis of Timed Asynchronous Circuits

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

Atsushi Matsumoto Manabu Kato
Tokyo Institute of Technology
{atsushim, kato}@yt.cs.titech.ac.jp

Chris Myers*
University of Utah
myers@ece.utah.edu

Abstract

This paper proposes applying a logic synthesis approach to high level synthesis from SpecC specifications to timed asynchronous gate-level circuits. The state-based logic synthesis is used to allow for global and timing optimization. In order to reduce the overhead in resetting phases, a protocol called early acknowledgment protocol and its STG generation technique are proposed. In this protocol, the state variables inserted to guarantee that STGs have CSC usually cause no overhead. The experiments to synthesize a portion of a DCT circuit show that the proposed method can handle a nontrivial example and produce a smaller and faster circuit than a previous approach.

Key Words: *High level synthesis, SpecC, resource allocation/scheduling, logic synthesis, timed STGs, Balsa*

1. Introduction

Recently, high level synthesis from C-like languages such as SpecC and SystemC is attracting attention of synchronous circuit designers, and several commercial tools (from Mentor Graphics, Bluespec, Forte, etc.) and public tools (e.g., Spark[1]) are now available that support it. We have been developing a tool that synthesizes asynchronous circuits from a subset of SpecC descriptions. In this approach, users can express specifications at a level in which asynchronous and synchronous designs are indistinguishable, and so, asynchronous circuits can be synthesized without special knowledge for asynchronous circuit design. Furthermore, the same specification can be used to synthesize a synchronous circuit, which makes it easier to compare both asynchronous and synchronous implementations.

The high level synthesis of asynchronous circuits have been studied mainly in the context of syntax directed translation or local control [2, 3, 4, 5, 6, 7]. These methods are extremely efficient because each construct of the specifica-

tion language is directly modeled by a circuit without enumerating the state space. These methods, however, make it difficult to perform global optimization such as logic minimization and timed circuit optimization. For this reason, we have been tackling the challenge of using logic synthesis for high level synthesis. In this approach, the high level description is translated to a signal transition graph (STG), to which the low level logic synthesis method is applied. The major reasons that this approach has not been used to date for high level synthesis are the state explosion problem and the *complete state coding* (CSC) problem. Since STGs generated from high level descriptions are usually large, it is extremely difficult to generate its state space and solve the CSC violations. In order to overcome these problems, we have proposed a method for decomposition based logic synthesis [8], in which the given STG is projected for each output signal such that it has sufficient information to synthesize the sub-circuit for this output signal. This approach scales very well, especially for STGs that have CSC and a small relevant input set (CSC support set) for each output signal. For example, in the experiments done in this paper, an STG with 533 signals and 1907 transitions is successfully handled by this approach. As for the CSC problem, this paper shows a solution in which STGs are constructed that already include all the state variables necessary to achieve CSC. The remarkable feature of this approach is that those inserted state variables cause almost no performance degradation in many cases due to the use of a special protocol. Our high level synthesis approach based on logic synthesis, of course, has limitation in the size of STGs, because state space exploration is necessary unlike the syntax directed method. This problem can be avoided by expressing the SpecC specification as a set of procedure declarations, and synthesizing each procedure separately. The performance overhead in using procedures certainly exists, but it can be reduced by defining as large procedures as possible that can be synthesized in a reasonable amount of time and memory.

The goal of this paper is to present the logic synthesis approach to the high level synthesis, to show how the global and timed optimization can be done, and to demonstrate that it can handle nontrivial examples.

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

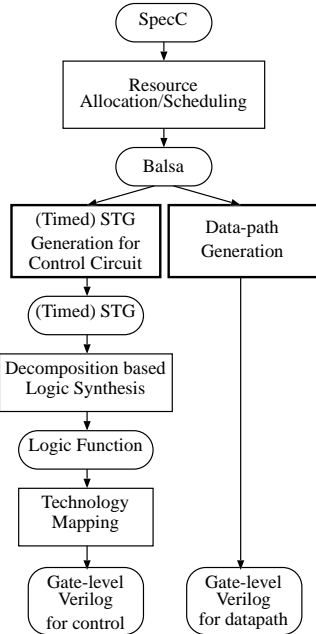


Figure 1. Tool Flow.

The rest of this paper is organized as follows. The next section describes the overall approach of our method. Section 3 discusses the special protocol used in our method. The STG generation algorithm based on that protocol is shown in Section 4. Section 5 explains how timed circuits are synthesized in our approach. Section 6 shows several experimental results, and the conclusion is given in Section 7.

2. Approach

The flow of the high level synthesis tool that we are developing is shown in Figure 1. This tool synthesizes data path circuits using the *bundled data method*, where it is assumed that the maximal data-path delay for each operational unit or memory, called a *matched delay*, is given. The input of the tool is a specification for a circuit to be synthesized expressed by a subset¹ of the SpecC language. The tool also takes the resource constraints, i.e. the maximal number of operational units for each type of arithmetic and relational operations.

The first step of the synthesis is to allocate the operational units and registers according to the resource constraints. The list scheduling algorithm and left-edge algorithm [9] used in synchronous design are adapted to our purpose in this step [10, 3, 11]. The result of this step is expressed in the Balsa language [12]. The Balsa language is chosen because sequential and parallel execution can be ex-

¹ Currently, it is almost C except that it also allows datapath declarations, synchronization, and port communication.

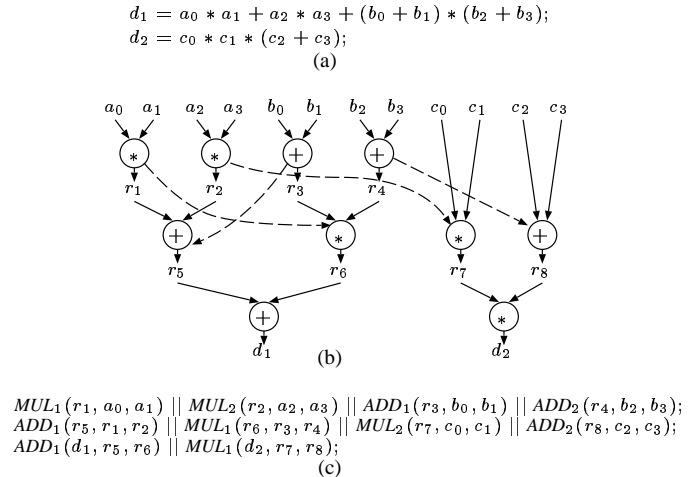


Figure 2. Resource Allocation/Scheduling.

pressed explicitly and clearly as well as port write/read operations are supported. Other languages can be used as long as these properties are satisfied. Another advantage of using Balsa descriptions is that the Balsa synthesis system can be applied to obtain actual circuits, which makes it easier to check the specification and estimate the circuit performance.

Although our Balsa descriptions to express the resource allocation results are legal with respect to the Balsa grammar, they do not use any Balsa built-in arithmetic operations such as addition and subtraction. Instead, every operational unit is connected to the corresponding input and output ports, and each operation is performed by sending and receiving data through those ports to/from the operational unit determined by the resource allocation algorithm. For example, consider a sequence of assignment statements shown in Figure 2(a) that are taken from a SpecC program. Its data dependency is shown by the solid arrows in Figure 2(b). From this data dependency, one can notice that 4 multipliers and 4 adders are necessary if these operations must be done as concurrently as possible. Note that 4 multipliers instead of 3 are necessary, because the multiplication for r_3 and r_4 is ready after “ $b_0 + b_1$ ” and “ $b_2 + b_3$ ”, but there it no guarantee that any of the other three multiplications are finished at that time. On the other hand, the multiplication for r_7 and r_8 can be started only after “ $c_0 \times c_1$ ”. Thus, the multiplier used for “ $c_0 \times c_1$ ” can be used again for “ $r_7 \times r_8$ ”, which implies that 5 multipliers are not needed.

Suppose that at most two multipliers and two adders can be used at the same time. This resource constraint is represented by adding *resource edges* [11], shown by dotted arrows in the figure. A resource edge from operation 1 to operation 2 indicates that operation 2 should wait until the completion of operation 1. Since maximal delays of func-

tional operations are assumed to be known in our method, the list scheduling algorithm for synchronous clock cycle assignment can be used to schedule the operational units, if every operation is considered to take multi-clocks. The obtained scheduling of operational units is actually represented by giving causality relation between operations using resource edges, instead of assigning clock cycle to each operation. Registers that can be shared are determined also from the maximal delays of operational units based on the left-edge algorithm.

Figure 2(c) shows one implementation of this data flow graph with resource constraints which uses sequential (;) and parallel (||) constructs, where, for example, $MUL_1(r_1, a_0, a_1)$ denotes that the multiplication between a_0 and a_1 is performed by using the multiplier No.1, and the result is stored into a register r_1 . This multiplication is actually done through the output port out_MUL_1 and input port in_MUL_1 to/from the multiplier No.1 with the following parallel port write/read operations.

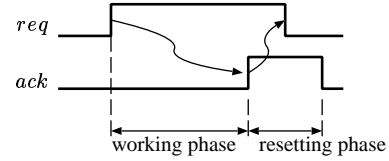
$$out_MUL_1 \leftarrow \{a_0, a_1\} \parallel in_MUL_1 \rightarrow r_1$$

$MUL_1(r_1, a_0, a_1)$ is the macro definition for these operations. The sequential constructs in Figure 2(c) guarantee the data dependency and resource constraints. They, however, include other constraints not required by Figure 2(b), such as $MUL_2(r_2, a_2, a_3)$ to be finished before $ADD_2(r_8, c_2, c_3)$. These constraints are necessary for simplifying our STG generation with CSC², although the optimal resource allocation/scheduling may be missed due to them. Avoiding these redundant constraints is one of our future work.

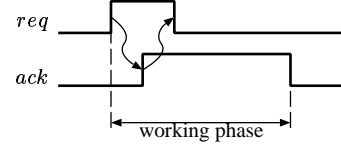
In the second step, STGs for the control circuits are generated from the above Balsa descriptions. The two important issues in this STG generation step are to reduce the overhead in the resetting phase and to guarantee that the STGs have CSC. In our approach, 4-phase signaling protocol is adopted, which has the resetting phase as shown in Figure 3(a), and it is very important to hide it in some appropriate way. To address this problem, our method uses a special protocol, which we call the *early acknowledgment protocol*. While acknowledgement is typically indicated by the rising edge of the acknowledgement signal in the traditional 4-phase signaling protocol, in the early acknowledge protocol, it is indicated by falling edge. Figure 3(b) shows this protocol. As shown in this figure, the resetting phase no longer exists in the early acknowledgment protocol. The details about this protocol are discussed in the next section.

Our algorithm also inserts state variables, which we call CSC variables, whenever they are necessary in order to

² If these constraints are not added, then the constraints cannot be represented only by sequential and parallel constructs. Our STG generation algorithm does not support such a general causality relation, and actually, for such cases, it is not easy to add state variables to guarantee the CSC property.



(a) Original Protocol



(b) Early Acknowledgement Protocol

Figure 3. 4 Phase Protocol.

guarantee that the generated STGs have CSC. Although it is possible to apply automatic approaches to solve CSC, it is often very computationally expensive and can generate redundant and slow circuits especially for large STGs such as those generated from high level specifications. Hence, our approach inserts CSC variables during the STG construction phase.

The third step of synthesis is to generate the data path circuits from the above Balsa descriptions. In this step, registers for local variables and input/output ports as well as multiplexers placed in the input sides of registers and output ports are generated, and the signals from the control circuits are connected. The information about the data path such as widths and bit sections is included in the Balsa description, and the data path synthesis is performed by the static analysis of the Balsa description.

In the fourth step, the control circuits are synthesized from the STGs, obtained in the second step, by the decomposition based synthesis tool *nutas*³. In the decomposition based synthesis, for each output signal w , its possible trigger signals are chosen to form the initial input signal set. The transitions related to the other signals are considered to be dummy, and are eliminated, when possible, by a net contraction algorithm. The resultant reduced STG is analyzed and checked if it has CSC. If so, the sub-circuit for w can be synthesized from the reduced STG. Otherwise, the CSC violation paths are examined and the signals to avoid the CSC violations are added into the input set. The net is contracted with respect to the new input set, and the above process is repeated until the sub-circuit is synthesized. This synthesis process is completely independent of sub-circuits for other output signals. Thus, it is very easy to obtain linear improvement by partitioning the output signal set to n subsets and sending them to n processors. Furthermore, it is sufficient

³ <http://research.nii.ac.jp/~yoneda/>

for each processor to have memory for handling one output signal. On the other hand, no multi-output optimizations are done in our tool. It can be extended to handle multiple outputs at the same time, which allows several multi-output optimizations. It is, however, difficult to find a suitable partition of the output signals automatically.

A similar approach is used in [13, 14]. They show that their methods can handle very large STGs. Compared with these works, our approach has an advantage that the state exploration and CSC violation analysis are done only on the reduced STGs, which makes it possible to handle larger STGs. From the experimental results shown in Section 6, our method seems to work better especially for the large STGs generated by our Balsa compiler. Another advantage of our approach is that state space exploration and CSC violation analysis can easily be extended to incorporate timing assumptions. This work actually uses a timed version of the decomposition based synthesis method [15].

In the final step, the actual gate level control circuits are generated by applying technology mapping to the logic functions obtained in the previous step. The timing information can be used to efficiently perform this step [16, 17]. Also, the reduced STG obtained in the previous step can be used as the input of such the timed circuit technology mapping, which further reduces the cost of this step.

Although the flow in Figure 1 is shown such that the allocation/scheduling step is fully decoupled from the synthesis step, it may be necessary to redo the allocation/scheduling step using the more accurate delay information obtained from the synthesized gate-level circuits, or in some cases, the placed-and-routed cells, and it is actually possible because our tool is fully automated. On the other hand, we are also aware that our tool flow, especially the allocation/scheduling step, is too simplified, i.e., there are many issues to be considered to derive good implementations, such as optimization through code motion. However, many optimization techniques developed for the allocation/scheduling step in synchronous high-level synthesis tools can be applied also to our tool without major changes, because there are no significant gaps between synchronous and asynchronous design in this level. Therefore, this paper focuses on steps 2 and 3 of the flow, and the tighter integration of the above optimization techniques is considered in future work.

3. Early Acknowledgment Protocol

The early acknowledgment protocol is a generalized version of the local clock method proposed in [18]. In the local clock method, the acknowledgment signal is raised when about half of the matched delay has passed after the request signal goes high, and is lowered after the same delay. In this paper, the semantics of the acknowledgment signal is modified such that the acknowledgment is achieved when the

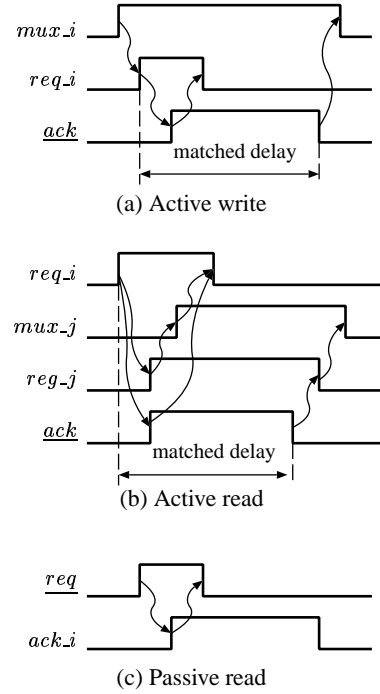


Figure 4. Early Acknowledgment Protocol.

acknowledgment signal goes low, i.e., it can go high anytime after the request signal goes high. Thus, usually the acknowledgment signal is raised as soon as possible. This is why we call it “early acknowledgment”.

This early acknowledgment protocol is similar to the 2-phase signaling protocol. In 2-phase, however, some kind of translation mechanism from 2 phase to 4 phase is necessary somewhere, which makes the circuits a little more complicated. On the other hand, the early acknowledgment protocol is purely based on 4-phase signaling, and so, it can fully inherit the simplicity of 4-phase signaling. The quantitative comparison between both methods based on several nontrivial case studies is, however, one of our future work.

This section presents how to implement the early acknowledgment protocol for active port write/read and passive port read operations.

(1) Active Port Write

The active port write operation is expressed by

$$port \leftarrow req$$

in the Balsa language. As shown in Figure 4(a), this operation starts when the i -th path of a multiplexer is selected by mux_i , and the request signal, req_i , is raised. The index i is used to distinguish the port write operations on the same port. A multiplexer is used for each output port, and the registers that send data to this output port are connected to this

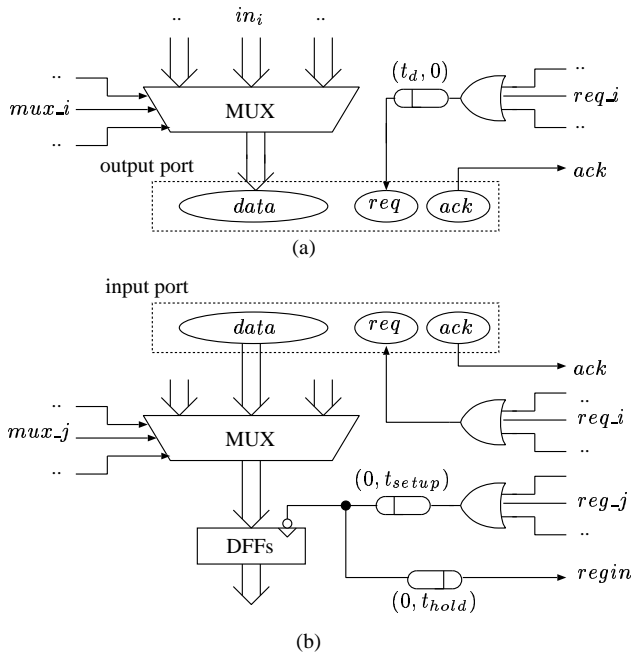


Figure 5. Ports and Registers.

multiplexer. Figure 5(a) shows the interface circuit around the output port. For (a, b) of the delay element, a and b denote the rising delay and the falling delay, respectively. The delay t_d is used to avoid asserting the request signal before the multiplexer output becomes valid, when the multiplexer is large and slow.

The environment responds to the request signal by raising the acknowledgment signal immediately according to the early acknowledgment protocol. In Figure 4 (and also in Figure 7, 8, 9, and 10), input signals are underlined. Then, the control circuit lowers the request signal, but the multiplexer is kept open until the acknowledgment signal goes low. After the matched delay, the environment indicates the completion of the data read by lowering the acknowledgment signal, which causes the control circuit to lower mux_i and complete the active write operation. The environment can generate the acknowledgment signal as shown in Figure 6. This acknowledgment signal generation circuit is more complicated and slower than that for the ordinary 4 phase signaling. However, the delay in raising the acknowledgment signal does not cause any performance overhead in the early acknowledgment protocol as shown in the end of next section, and the falling delay can be included in the matched delay.

(2) Active Port Read

The active port read operation is expressed by

$$port \rightarrow reg$$

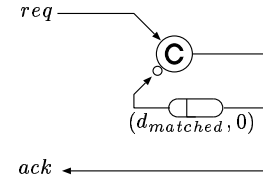


Figure 6. Ack Signal Generation.

in the Balsa language. As shown in Figure 4(b), the control circuit first raises req_i . Then, it raises reg_j to prepare the register write operation, and selects the multiplexer by mux_j+ , where i is the index of the read operations of this port and j is the index of the write operations to this register. The interface circuit for this operation is shown in Figure 5(b). The environment raises the acknowledgment signal, and lowers it after the matched delay. The control circuit responds to it by writing data to the register by reg_j- , and then closes the multiplexer by mux_j- . To ensure the setup time of the DFF, a delay element with falling delay t_{setup} is used for the case that the multiplexer delay is large. Similarly, the hold time of the DFF is ensured by waiting for the falling edge of reg_in- , which is the output of another delay element with the falling delay t_{hold} , before lowering mux_j . It should be assumed that the data on the data path is kept by the environment for at least the sum of the setup time and the hold time of DFFs after $ack-$.

(3) Passive Port Read

The passive port read operation is expressed by

$$select \ port \ then \ \dots \ end$$

in the Balsa language. As shown in Figure 4(c), the environment raises the request signal, and the control circuit responds to it by raising the acknowledgment signal as well as starting the execution of the select body between “then” and “end”. When all of the statements in the body (including the actual port read) are completely executed, the control circuit lowers the acknowledgment signal.

4. Generating STGs for Control Circuits

This section presents how the STGs for control circuits are generated from the Balsa descriptions. The STG generation is done inductively from the inner most statements. For a Balsa statement X , let N_{X+} and N_{X-} denote the sub-STGs that raise and lower the signals related to X . The transitions from N_{X+} to N_{X-} (e.g., p_{req-} and q_{req-} in Figure 8(a)) are considered to belong to both N_{X+} and N_{X-} . In the normal 4 phase signaling, N_{X+} and N_{X-} represent the behavior of the working phase and the resetting phase,

respectively. In the early acknowledgment protocol, however, even when N_X+ completes, the actual data transfer continues, and it finally finishes when N_X- completes.

The algorithm described below guarantees that the generated STGs are output semi-modular and have CSC. To prove this property, we use the following two propositions, where input (output) transitions and dummy transitions denote the transitions that are related to input (output) signals and the transitions that are related to no signals.

Proposition 1 In the considered sub-STG (i.e., N_X+ or N_X-), every transition that actually conflicts with other transitions is either input transitions or dummy transitions. In the latter case, every path from those dummy transitions in the sub-STG has an input transition before any output transitions.

Proposition 2 In the considered sub-STG (i.e., N_X+ or N_X-), every state vector that causes different output signal changes is different.

The following shows N_X+ and N_X- for each statement X . The signal names are mainly in the form of $name_{func_index}$, where $name$ is a port name or a register (variable) name, $func$ is one of req , ack , reg , $regin$, mux by the same naming conventions as in the previous section, and $index$ is the index for the same port or register access. Also assume that when N_X+ or N_X- is considered, all the sub-STGs included in it satisfy both Proposition 1 and Proposition 2.

(1) Active Port Write/Read

The protocol shown in the previous section can be directly implemented as shown in Figure 7(a) and (b), where the port name is “ p ”, the register name is “ a ”, and $\$$ represents a dummy transitions. N_X+ and N_X- are connected by an outer statement such as a sequential construct. These N_X+ and N_X- apparently satisfy both Proposition 1 and Proposition 2.

(2) Register Write

For the register write operation below, the sub-STG shown in Figure 7(c) is generated in the similar way to the active port read.

$$X \equiv a := b$$

These also satisfy Proposition 1 and Proposition 2.

(3) Select

Consider the following select statement.

$$X \equiv \text{select } p \text{ then } Y_1 \mid q \text{ then } Y_2 \text{ end}$$

Note that the port p should not assert the request when the other port q is working. Similarly, q should not assert a request when p is working. For handling simultaneous requests from two or more input ports, the Balsa language

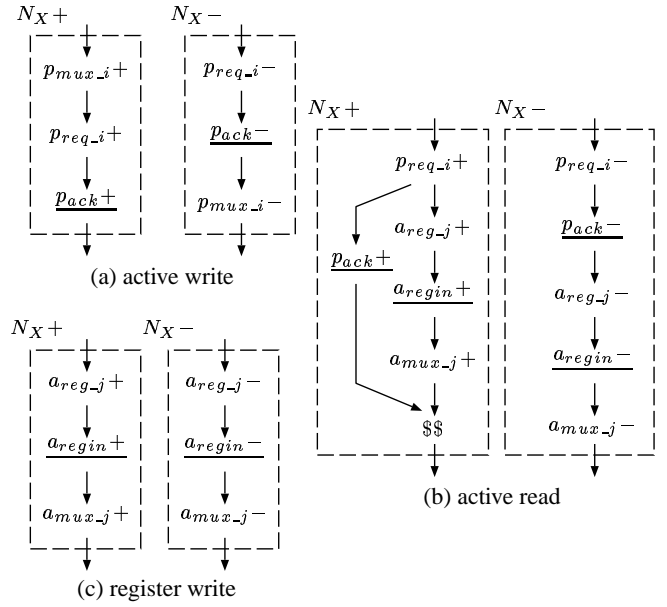


Figure 7. STG generation (1).

has the “arbitrate” construct, but our work does not support it. Thus, if it is really necessary to handle such simultaneous requests, the users should be responsible for implementing arbiters in the environment. For the above “select” construct, the sub-STGs shown in Figure 8(a) are generated. In N_X+ , the acknowledgment is asserted in response to either request signal, and the corresponding body (N_{Y_1+} or N_{Y_2+}) is executed. In N_X- , after N_{Y_1-} or N_{Y_2-} is executed, the acknowledgment signal is lowered if the request signal is low.

It seems that two output transitions $pack_{-i}+$ and $pack_{-j}+$ are in conflict in N_X+ . They are, however, not actually in conflict with each other, because both p and q never assert the request at the same time. A similar argument holds for the two dummy transitions in N_X- . Thus, N_X+ and N_X- satisfy Proposition 1.

In N_X+ , when $preq-$ occurs, $preq$ has the same value as the initial state. But, since $pack_{-i}$ is 1 at that time, its state vector is different from that of the initial state. The two dummy transitions in N_X- may cause different output signal changes, but the state vectors after firing those dummy transitions can be distinguished again by the value of $pack_{-i}$ and $pack_{-j}$. Hence, N_X+ and N_X- satisfy also Proposition 2.

A similar argument holds for select statements with three or more ports.

(4) Parallel

Consider the following statement with parallel construct.

$$X \equiv Y_1 \parallel Y_2$$

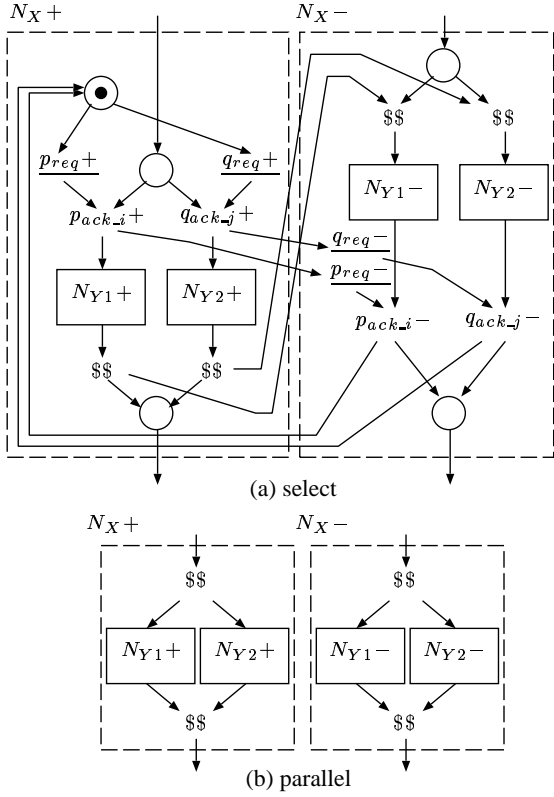


Figure 8. STG generation (2).

The sub-STGs for this X are shown in Figure 8(b). From this construction, Proposition 1 and Proposition 2 are apparently satisfied. A similar argument holds for the statements with three or more parallel constructs.

(5) Sequential

Consider the following statement with sequential construct.

$$X \equiv Y_1 ; Y_2 ; \dots ; Y_n$$

The sub-STGs for this X with $n = 3$ and $n = 4$ are shown in Figure 9. As shown in the figures, CSC variables are inserted such that there exists a change of a CSC variable between every N_{Y_i+} and N_{Y_i-} , and every CSC variable goes high and low once in N_{X+} and N_{X-} . When n is odd, a change of one CSC variable is in excess, and so, it is consumed concurrently at the end of N_{X+} . Since there are no conflicting transitions, Proposition 1 holds clearly. Furthermore, from the following facts, Proposition 2 also holds in N_{X+} and N_{X-} .

- A state in N_{Y_i+} and another state in N_{Y_i-} have different state vectors because a CSC variable certainly changes.
- As for a state in N_{Y_i-} and another state in $N_{Y_{i+1}+}$, care should be taken when the same port or variable

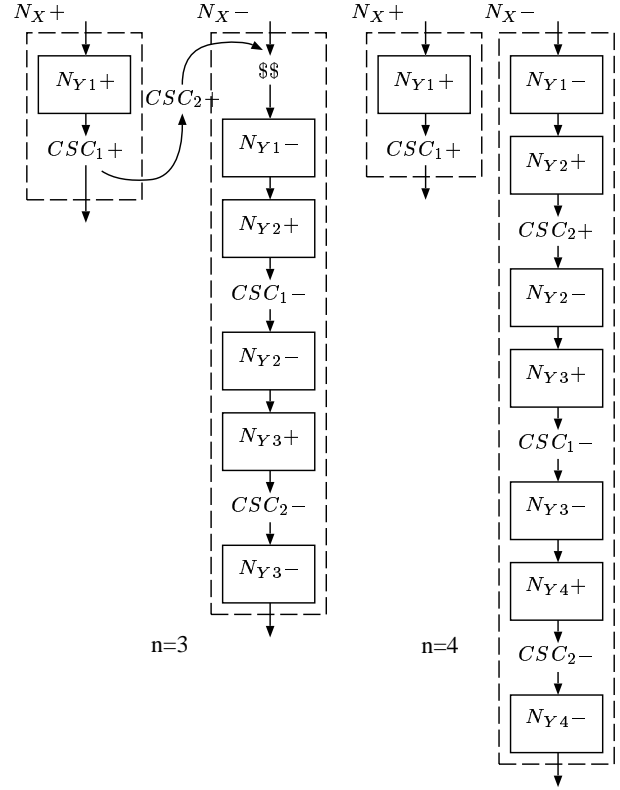


Figure 9. STG generation (3).

is accessed in N_{Y_i} and $N_{Y_{i+1}}$, because N_{Y_i-} and $N_{Y_{i+1}+}$ have the same state encoding with respect to the CSC variables. For example, when n is even, CSC_1 takes 1 in both N_{Y_1-} and N_{Y_2+} , while the other CSC variable take 0. Thus, if $preq_{-1}$ goes low at state s_1 in N_{Y_1-} and it again goes up in N_{Y_2+} with reaching s_2 , s_1 and s_2 may have the same state vectors, but the output behavior is apparently different. This problem can be solved by inserting a CSC variable change also between N_{Y_i-} and $N_{Y_{i+1}+}$. However, remember that different indices are given for the access to the same port or variable. This means that N_{Y_i} and $N_{Y_{i+1}}$ use different signals to access to the same port, i.e., $preq_{-1}$ and $preq_{-2}$. Thus, the falling signal in N_{Y_i-} is $preq_{-1}$, for example, and the rising signal in $N_{Y_{i+1}+}$ is $preq_{-2}$. Hence, a state in N_{Y_i-} and another state in $N_{Y_{i+1}+}$ can be distinguished without using another CSC variable change between N_{Y_i-} and $N_{Y_{i+1}+}$.

- CSC variable changes $CSC_{1+}, \dots, CSC_{k+}, CSC_{1-}, \dots, CSC_{k-}$ occur in this order in N_{X+} and N_{X-} . Thus, every state vector with respect to CSC variables is different in N_{X+} and N_{X-} .

(6) While

Consider the following while construct.

$$X \equiv \text{loop while } cond \text{ then } Y \text{ end}$$

Our method requires that when the conditional branch statements such as **while** and **if** are executed, the comparator with respect to their conditional expressions should be accessed beforehand. A comparator is considered as 1 bit register, i.e., it takes the two data to be compared and the relational operator, and puts the comparison result into the 1 bit register. The **while** and **if** constructs refer to the 1 bit register, and decide if the conditional branch takes place or not. In our framework, a comparator is accessed by

$$out_CMP \leftarrow \{data_0, data_1, rel_{op}\} \parallel in_CMP \rightarrow c$$

and this comparator access and the conditional branch should be ordered sequentially. Thus, when the value of c is tested by the conditional branch statement, it is stable. It should be noted that it is not practical to handle the actual comparison results during the controller synthesis because this requires handling real data. Instead, as shown in Figure 10(a), our sub-STG correctly models the timing when the comparison results are updated (i.e., between c_{reg-j-} and c_{regin-}), and decides comparison results themselves nondeterministically by the dummy transition conflicting with $c+$ and $c-$ (The upper-left dotted rectangles show a fragment of the active port read operation “ $in_CMP \rightarrow c$ ”). As shown in Figure 10(a), N_{X+} decides if the loop body formed by N_{Y+} and N_{Y-} should be executed or not according to the value of c . Since the value of c is decided nondeterministically, the state space includes both cases where $c = 1$ and $c = 0$, and so, the synthesized circuit can accept any possible change of the input c .

The signal c is an input for the control circuit, and the dummy transition conflicting with $c+$ and $c-$ directly reaches the input transition c_{regin-} . When the dummy transitions in N_{X+} fire, the value of c is stable. Thus, those dummy transitions as well as $c+$ and $c-$ do not actually conflict with each other. Hence, Proposition 1 holds.

Furthermore, from the following facts, Proposition 2 also holds.

- The dummy transition conflicting with $c+$ and $c-$ does not cause different output changes.
- The conflicting dummy transitions in N_{X+} may cause different output changes, but the states after firing those dummy transitions are distinguished by the value of c .
- A CSC variable changes between N_{Y+} and N_{Y-} . This $CSC+$ is to distinguish the states in N_{Y+} and N_{Y-} as used for the sequential construct. On the other

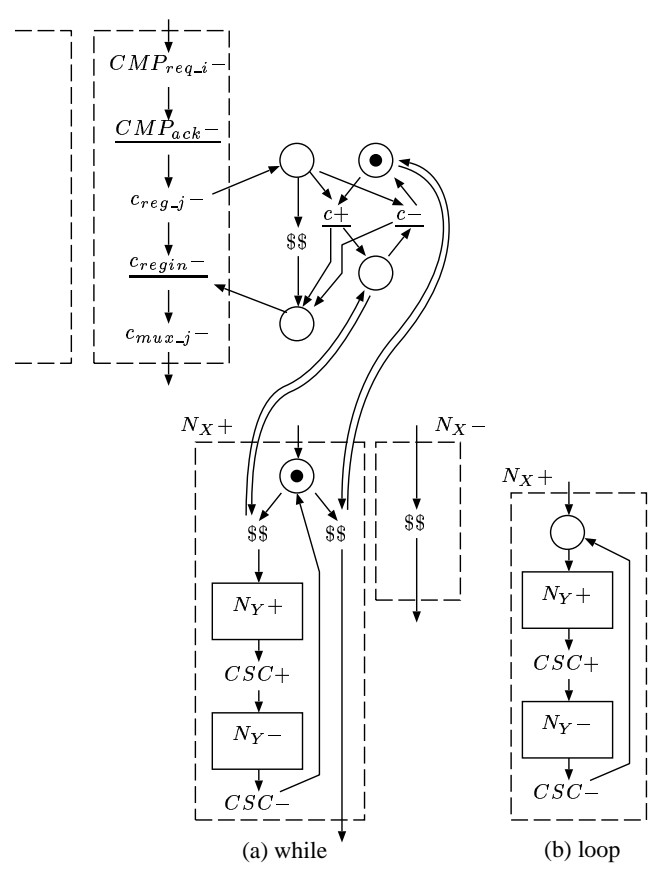


Figure 10. STG generation (4).

hand, $CSC-$ is inserted only for changing its value back to 0, not for distinguishing the states. This is for the following reason. The state after $CSC-$ and another state before N_{Y+} have the same state vector, if c is true. However, they cause the same output behavior, and therefore, no CSC violation occurs.

(7) Loop and others

For loop construct below, the sub-STG shown in Figure 10(b) is generated.

$$X \equiv \text{loop } Y \text{ end}$$

This is the same as the body of the while construct. Since the loop construct does not terminate, N_{X-} is empty. N_{X+} apparently satisfies Proposition 1. Proposition 2 also holds for similar reasons to those for the while construct.

Our method also supports if-then-else, procedure call, procedure declaration statements. The if-then-else statement is similar to the while construct, the procedure call and procedure declaration are similar to the active port write and passive port read operations except for the data path width is 0 bits. Thus, the details are omitted here.

(8) Discussion

First, from the facts shown above that Proposition 1 and Proposition 2 hold for N_X+ and N_X- of each statement or operation, the following theorem holds.

Theorem 1 The generated STG by the above construction is output semi-modular and has CSC.

Second issue to be discussed is the performance overhead due to the inserted CSC variables. From the above construction, CSC variables are inserted for sequential statements, while constructs, and loop constructs. Among them, consider those inserted between $N_{Y_i}+$ and $N_{Y_i}-$. For example, in the sub-STGs generated for the Balsa description

$$p \rightarrow a ; q \rightarrow b ; \dots,$$

CSC variables are inserted between $N_{Y_i}+$ and $N_{Y_i}-$. From Figure 7(b) and Figure 9, the CSC variables are triggered by the rising edges of the acknowledgment signal p_{ack} or q_{ack} and the multiplexer control signal a_{mux_i} or b_{mux_i} . From the early acknowledgment protocol, those acknowledgment signals are raised quickly, and the actual data transfer completes when they fall. Thus, if the two output signal changes (CSC_{1+} and $p_{req_i}-$, or CSC_{2+} and $q_{req_k}-$) complete before that time point, the overhead of changing CSC variables can be hidden. This delay to lower the acknowledgment signal is the matched delays in the cases of memory access and operational unit access, which is larger than those for two output signal changes. In the case of register-to-register write operation, the delay for those two output signal changes can be overlapped with the setup time of the register. Hence, these CSC variables cause almost no performance overhead in many cases. Note that $CSC-$ after N_Y- for the while and loop constructs delay the first signal changes of N_Y+ , and so, the performance overhead due to them is not zero.

The other methods based on STG-like generation techniques have been proposed, e.g., in [5]. Our method can also generate those similar STGs by defining small and many procedures. Thus, our method can be considered as a kind of generalization of those methods. Using those small STGs, however, causes quite large overhead in either method. Our method allows users to choose the sizes of those procedures to control the overhead. Furthermore, the sequential constructs cause some overhead in propagating the activation signals in many other methods. Those activation signals correspond to the CSC variables in our method. Those CSC variables, however, cause almost no overhead in our method as shown above.

It should be noted that the number of inserted CSC variables is not optimal in our method. For example, for a small Balsa description shown in Figure 11, the STG G generated by our method contains four CSC variables. Let G' be the STG obtained from G by just replacing those CSC variables

```
loop
  select in_port0 then
    a := in_port0
  end;
  loop while a = 0 then
    begin
      out_port1 <- (1 as 1 bits) || in_port1 -> a;
      out_port2 <- (1 as 1 bits) || in_port2 -> a
    end
  end
end
```

Figure 11. A small example.

by dummy transitions. Thus, G' has no CSC. `petrify` [19] solves this CSC problem and obtains G'' , which has CSC, by adding three CSC variables automatically. This shows that our method is not optimal with respect to the number of inserted CSC variables. However, the performance of the synthesized circuits does not directly depends on the number of CSC variables. Actually, when in_port0 returns 0, and in_port_i loop-backs the value sent to out_port_i for $i = 1, 2$, the simulations show that the circuit synthesized from G has the cycle time 14.91nS, while the one from G'' has 16.42nS, under the delay information of a 0.25μ gate library and the loop-back delay with 3nS.

5. Timed Circuit Synthesis

The control circuit synthesized by the speed independent circuit synthesis may include redundant sub-circuits when the actual gate and environment delays are considered. The timed circuit synthesis removes those redundant sub-circuits through timed state space exploration by using the delay information. This delay information is usually given, for a generalized-C (gC) element or an atomic gate driving each non-input signal, as the lower and upper bounds on the delay from the time at which all inputs necessary to enable the gate arrive until the output of the gate changes. In a technology-mapped circuit, such a gC or atomic gate is implemented as an acyclic network of gates. The above delay corresponds to the delay from the AND gates in the input side to the final gate in the gate network. Even if the output of the final gate is fed back to the inputs of the gate network, the above delay is not affected.

The main technical issue in this approach is how to actually obtain these lower and upper bounds of the delays. Our method achieves this as follows, and obtains a timed STG from the original STG by adding those delay bounds to each transition.

1. By the speed independent circuit synthesis, a speed independent control circuit is obtained from the original STG.
2. The actual gate circuit is obtained by technology mapping.

3. For each non-input signal, do the following.
 - (a) For each acyclic gate network implementing the non-input signal, the maximal delay from the input to the output is computed from the maximal gate delays given by the gate-library vendor and the maximal number of stages in the gate network. Then, this can be considered to be the upper bound of the delays for the corresponding non-input transitions, because a timed circuit is always smaller than or equal to the original speed independent circuit.
 - (b) For its lower bound, using 0 is the most conservative. However, it may not reduce the circuit efficiently, because it is often too conservative. Thus, a better approach is to decide the lower bound from the delay of a single g-C or AND gate. If the circuit obtained by the timed circuit synthesis has larger delay than the above lower bound, it is done. Otherwise, re-synthesis is necessary using the lower bound obtained from the timed circuit.
4. As for the input signals in the active port write and read operations, the rising delay (d_1) of the acknowledgment signals can be guessed as one C-element delay from Figure 6, and their falling delays can be decided from the matched delays minus d_1 and the control circuit delays to lower the request signals. The lower and upper bounds can be decided with some margins. On the other hand, it is unknown when the request signals are issued for the passive port read operations. Thus, $[0, \infty]$ is used for those input transitions.

The reduction of circuits by timed circuit synthesis happens mainly in the parallel constructs. The speed independent control circuit must wait for the completion of all statements connected by the parallel construct, even if some of them finish much earlier than others. This generates AND gates with large fan-in in the input side of the gate network. The timed circuit synthesis can decide that too early acknowledgment (indicated by the falling edges of acknowledgment signals) are unnecessary, and smaller AND gates are generated, which increases the performance. For example, by the timed circuit synthesis, the total number of literals for the controller of Figure 2(c) is reduced from 255 to 214, and the cycle time is reduced from 24.58nS to 24.18nS, when it is assumed that a 0.25μ gate library is used and the addition and the multiplication take 3nS and 5nS, respectively.

6. Experimental Results

In this section, to demonstrate the applicability of the proposed method, an IIR filter, a FIR filter, and a portion

```

for(i=0;i<8;i++)
{
  aptr = x+i;
  bptr = aptr+56;

  a0 = (((*aptr*bptr)) << (2));
  c3 = (((*aptr-*bptr)) << (2));
  aptr += 8;
  bptr -= 8;
  a1 = (((*aptr*bptr)) << (2));
  c2 = (((*aptr-*bptr)) << (2));
  aptr += 8;
  bptr -= 8;
  a2 = (((*aptr*bptr)) << (2));
  c1 = (((*aptr-*bptr)) << (2));
  aptr += 8;
  bptr -= 8;
  a3 = (((*aptr*bptr)) << (2));
  c0 = (((*aptr-*bptr)) << (2));

  b0 = a0+a3;
  b1 = a1+a2;
  b2 = a1-a2;
  b3 = a0-a3;

  aptr = y + i;

  *aptr = (((362L*(b0+b1))) >> (9));
  aptr[32] = (((362L*(b0-b1))) >> (9));

  aptr[16] = (((196L*b2)+(473L*b3))) >> (9));
  aptr[48] = (((196L*b3)-(473L*b2))) >> (9));

  b0 = (((362L*(c2-c1))) >> (9));
  b1 = (((362L*(c2+c1))) >> (9));

  a0 = c0+b0;
  a1 = c0-b0;
  a2 = c3-b1;
  a3 = c3+b1;

  aptr[8] = (((100L*a0)+(502L*a3))) >> (9));
  aptr[24] = (((426L*a2)-(284L*a1))) >> (9));
  aptr[40] = (((426L*a1)+(284L*a2))) >> (9));
  aptr[56] = (((100L*a3)-(502L*a0))) >> (9));
}

```

Figure 12. A DCT example.

of the Discrete Cosine Transform (DCT) circuit are synthesized, and cycle times and circuit sizes are evaluated. These experiments are done on a 2.8 GHz Xeon workstation with 4 gigabytes of memory. Figure 12 shows the SpecC specification for our DCT example. This is just half of the actual DCT circuit. Another half portion has almost the same structure with the same loop count, and these two portions are sequentially connected. Thus, it is reasonable that each of them is declared as a procedure and synthesized separately. The memory access by pointers is implemented in the same way as operational units, i.e.,

out_MEM <- *address* || *in_MEM* -> *data*.

For each example, two types of circuits are synthesized under two different resource constraints. In Table 1, the column labeled by “Res.” shows the maximal number of available operational units for multiplication, ALU operation (addition/subtraction), and shift operation (“-” indicates that the operational unit is not used). It is commonly as-

Table 1. Synthesis and Simulation Results.

Circuits			Synthesis time (sec.)			Gate counts for Ctrl.	Cycle time (nS)	
Name	Res. ⁽¹⁾	Method	nutas	moebius ⁽²⁾	csat ⁽³⁾		total	cntl.
IIR	(1,1,-)	SI	16.3 (12.3)	(50.1)	112.5 (98.1)	198	87.8	31.8
		Timed	16.9	—	—	198	86.6	30.6
		Balsa	3.1 (Balsa system)			291	142.9	86.9
IIR	(2,2,-)	SI	27.7 (16.3)	(41.4)	41.5 (38.7)	238	48.6	17.6
		Timed	134.26	—	—	196	47.4	16.8
		Balsa	1.7 (Balsa system)			270	80.9	49.9
FIR	(1,1,-)	SI	285.1 (234.5)	(646.2)	(≥ 5h)	487	192.3	69.3
		Timed	288.7	—	—	469	187.9	64.9
		Balsa	12.0 (Balsa system)			684	311.4	188.4
FIR	(2,2,-)	SI	422.0 (279.4)	(546.8)	(≥ 5h)	675	105.0	39.0
		Timed	6859.1	—	—	475	102.9	36.9
		Balsa	5.6 (Balsa system)			644	169.9	103.9
DCT	(1,1,1)	SI	5106.2 (2458.9)	(5909.2)	(≥ 5h)	1082	3847.3	1363.3
		Timed	5420.7	—	—	1036	3797.4	1313.4
		Balsa	37.5 (Balsa system)			1399	6265.5	3781.5
DCT	(1,2,2)	SI	7474.6 (1710.5)	(6715.0)	(≥ 5h)	1041	2679.2	759.2
		Timed	7468.0	—	—	982	2637.8	717.8
		Balsa	23.5 (Balsa system)			1282	4206.9	2286.9

(1) Resource: (#MUL, #ALU, #shifter)

(2) STGs for while bodies are synthesized.

(3) SAT instance generation times are not included.

sumed that one comparator is available, and that the memory has one address port and one data port.

For each resource constraint, circuits with a speed independent controller and a timed controller are synthesized by the proposed method. A speed independent circuit is synthesized also by the Balsa synthesis system (ver.3.4) from the Balsa description expressing each resource allocation result. The rows “SI”, “Timed”, and “Balsa” show those results, respectively. The column “nutas” shows the synthesis times of our method. The speed independent controller synthesis is also done by two other tools, moebius [13]⁴, and csat [14]⁵, in order to compare the low-level logic synthesis performance of three methods. Since moebius has a problem in handling self-loops generated for the while construct, its column shows the results for synthesizing the while bodies. The corresponding synthesis times of nutas and csat are shown in the parentheses of their columns. The column “Gate counts” in the table shows gate counts of the control circuits. Each circuit is simulated using the delay information for a 0.25 μ gate library⁶ as well as 10nS for

the memory access time, 5nS for the multipliers, and 3nS for the ALUs, shifter, and comparator. The column “Cycle times” shows the cycle time of each circuit, where “ctrl” represents the control circuit delays occupied in the total cycle times.

From these results, one can see that the significant improvement in the circuit performance is obtained by the proposed method compared with the Balsa synthesis system. The synthesis times are sacrificed for this performance improvement, but those for our examples are acceptable. For larger specification, defining appropriate sizes of procedures may be necessary. Compared with moebius and csat, nutas works better for the large STGs generated by our Balsa compiler. The performance improvement by the timed circuit synthesis is from 1.2nS to 6.2nS (per iteration in case of DCT), which approximately corresponds to 4 to 20 stages of gates.

It should be noted that when the concurrency increases and the relevant input set (CSC support set) sizes increase, the cost for the decomposition based synthesis also increases. This problem can be solved by inserting additional state variables to the sub-STG for the parallel constructs to reduce the relevant input set sizes. If these state variables are inserted in non-critical paths, the performance overhead caused by them may be small.

⁴ Run on a Pentium 4, 2.5GHz, 512MB machine.

⁵ Run on a Pentium 4, 3.06GHz, 1GB machine.

⁶ The simulation is done by the gate level Verilog descriptions annotated with the rising and falling delays given in the gate library. No placement/routing is considered.

7. Conclusion

This paper presents a complete design flow from SpecC specifications to gate-level designs. The methodology includes resource allocation and scheduling tuned for asynchronous circuits, although it is still too simple. It also makes use of decomposition based synthesis for design of the timed circuit controllers. A major difference between this approach and similar ones is that it uses state-based logic synthesis rather than syntax-directed translation. This allows for global logic and timing optimization which is limited in previous works. This paper also introduces the early acknowledgment protocol. To support this protocol, techniques are presented for data path generation and STG generation for the control circuits. We have applied our method to the design of several circuits, and have shown that our method produces substantial improvements in delay as compared with syntax-directed methods.

We are planning to apply more sophisticated resource allocation/scheduling techniques used for the synchronous circuit synthesis. It is also very interesting to compare the results for several benchmark circuits with those by other optimized asynchronous synthesis tools such as Balsa-CUBE [20] as well as synchronous high level synthesis tools such as Spark.

Acknowledgment

We would like to thank Josep Carmona and Jordi Cortadella for giving us the experimental results of our examples, obtained by their tool, and thank Victor Khomenko and Alex Yakovlev for making their experimental software available.

References

- [1] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. In *Proc. Intl. Conf. on VLSI Design*, 2003.
- [2] G. Gopalakrishnan and V. Akella. A transformational approach to asynchronous high-level synthesis. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.3.1–5.3.10, September 1993.
- [3] Rosa M. Badia and Jordi Cortadella. High-level synthesis of asynchronous systems: Scheduling and process synchronization. In *Proc. European Conference on Design Automation (EDAC)*, pages 70–74. IEEE Computer Society Press, February 1993.
- [4] Hans Jacobson, Erik Brunvand, Ganesh Gopalakrishnan, and Prabhakar Kudva. High-level asynchronous system design using the ACK framework. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 93–103. IEEE Computer Society Press, April 2000.
- [5] Euseok 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.
- [6] Catherine G. Wong and Alain J. Martin. High-level synthesis of asynchronous systems by data-driven decomposition. In *Proc. ACM/IEEE Design Automation Conference*, pages 508–513, June 2003.
- [7] M. Sacker, A. Brown, P. Wilson, and A. Rushton. A general purpose behavioural asynchronous synthesis system. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 125–134. IEEE Computer Society Press, April 2004.
- [8] T. Yoneda, H. Onda, and C. Myers. Synthesis of speed independent circuits based on decomposition. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 135–145. IEEE Computer Society Press, April 2004.
- [9] D. Gajski. *High-Level Synthesis*. Kluwer Academic Publishers, 1992.
- [10] M. Kato. Asynchronous circuit synthesis from high-level specification and its evaluation based on a case study (in Japanese). *Master Thesis, Tokyo Institute of Technology*, 2004.
- [11] Brandon M. Bachman, Hao Zheng, and Chris J. Myers. Architectural synthesis of timed asynchronous systems. In *Proc. International Conf. Computer Design (ICCD)*, pages 354–363, 1999.
- [12] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.
- [13] J. Carmona and J. Cortadella. ILP models for the synthesis of asynchronous control circuits. *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 818–825, 2003.
- [14] A. Yakovlev V. Khomenko, M. Koutny. Logic synthesis for asynchronous circuits based on petri net unfoldings and incremental sat. *Proc. of ACS D*, 2004.
- [15] T. Yoneda, H. Onda, and C. Myers. Synthesis of timed circuits based on decomposition. *to be submitted*.
- [16] Curtis Nelson. *Technology Mapping of Timed Asynchronous Circuits*. PhD thesis, University of Utah, 2004.
- [17] C. Nelson, C. Myers, and T. Yoneda. Efficient verification of hazard-freedom in gate-level timed asynchronous circuits. *Proc. of ICCAD*, pages 424–431, 2003.
- [18] N. Sretasereekul, H. Saito, M. Imai, E. Kim, M. Ozcan, K. Thongnoo, H. Nakamura, and T. Nanya. A zero-time-overhead asynchronous four-phase controller. *Proc. of IS-CAS*, 2003.
- [19] 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.
- [20] T. Chelcea and S.M. Nowick. Balsa-cube: an optimising back-end for the balsa synthesis system. In *14th UK Async. Forum*, 2003.