Synthesis of control circuits from STG specifications
Practical Exercise Manual
J. Cortadella  M. Kishinevsky  A. Kondratyev  L. Lavagno
A. Yakovlev
ASYNC 2003, Vancouver

1 Task 1: Handshake communication

1.1 What is Half-handshake?

Figure 1 shows a data processing structure consisting of two computation blocks, A and B, and a control circuit. Signals Ri and Ai are inputs, and Ao and Ro are outputs of the control. Output Ro can be seen as a latch enable signal for the data path. Output Ao is an acknowledgement signal sent to the previous control stage. The goal is to design the speed independent control circuit. A number of different control disciplines are possible. Let us choose a discipline based on handshaking between adjacent stages, for instance the one described by the Timing Diagram shown in Figure 2. Let us call it half-handshake. This discipline assumes the following:

1. The datapath includes latches which are transparent to input data when the control signal is low and which are opaque, i.e., insensitive to its data input, when the control is high, e.g., the latch in B is transparent when \( Ro = 0 \) and opaque when \( Ro = 1 \).

2. The fact that \( Ri \) becomes 1 indicates that the data in the previous stage, stage A, is captured and stable (after the previous stage has become opaque).

Additional assumptions could be made if needed, depending on our knowledge of the implementation of the latches and delays in the datapath. For example, we may need to assume that there is sufficient delay between the appearance of data on the data bus between stages and the rising edge on \( Ri \), and hence on \( Ro \), in order to guarantee the appropriate setup conditions for the latch in stage B.

The STG specifying the half-handshake control circuit is given in Figure 3. Signals \( Ri \) and \( Ai \) are inputs, and \( Ao \) and \( Ro \) are outputs. Output \( Ro \) can be seen as a latch enable signal for the data path. Output \( Ao \) is an acknowledgement signal sent to the previous control stage.

Figure 1: A data processing structure.

Figure 2: Half-handshake discipline: timing diagram.

Figure 3: An STG for the half-handshake: (a) independent handshakes at the left and right ports (b) unbounded version with \( Ri \rightarrow Ro \), (c) version with CSC violation, (d) final version with CSC and a redundant arc removed.
1.2 Exercise

1. Construct an STG specification (in astg format) for half-handshake control following steps presented in Figure 3. Save this specification in file half-hand.g. (Alternatively, find a sample solution half-hand.g in directory solutions and examine it carefully. Note the lines which will need to be commented out to illustrate various effects, such as unboundedness and CSC violations.)

2. Display your STG using the drawastg tool. For that, use the following commands and compare their effects:

```plaintext
drawastg half-hand.g | ghostview -
drawastg -noinfo half-hand.g | ghostview -
drawastg half-hand.g -o half-hand.g.ps
```

The -nofold option draws vertically as much as possible, while the -noinfo option removes the signal legend.

3. Generate and display the state graph for your STG using the write_sg tool. For that, run the following commands and compare their effects:

```plaintext
write_sg half-hand.g | drawastg -sg | ghostview -
write_sg half-hand.g -o half-hand.sg
drawastg -nofold half-hand.sg | ghostview -
write_sg -bin half-hand.g -o half-hand.bin.sg
drawastg -sg half-hand.bin.sg | ghostview -
drawastg -bin half-hand.bin.sg | ghostview -
drawastg -sg -noinfo -omomem -bw half-hand.bin.sg | ghostview -
```

The -bin option of write_sg generates a binary encoding for all states and checks for CSC violations, if any (i.e., pairs of states with the same code and different enabled output signals). The -bin option of drawastg understands this notation and highlights violations.

4. Check that the version corresponding to Figure 3 (b) (comment out appropriate lines in the half-hand.g file) is unbounded:

```plaintext
write_sg half-hand.g -o half-hand.sg
```

5. Check that the version in Figure 3 (c) (un-comment out appropriate lines in the half-hand.g file) has CSC-violations:

```plaintext
write_sg -bin half-hand.g | drawastg -bin | ghostview -
```

6. Check that the final version in Figure 3 (d) is directly implementable with a spec-independent circuit:

```plaintext
write_sg -bin half-hand.g | drawastg -bin | ghostview -
```

7. Derive netslists of logic equations for output signals of the final version (Figure 3 (d)) and the third version (Figure 3 (c)) using the -cg option of petrify (that generates one complex gate per output signal). Examine the equations [half-hand.cg.eqn] and the output STG [half-hand.out] files. Note where petrify introduces an additional signal csc0 to resolve CSC conflicts in the third version (Figure 3 (c)).

```plaintext
petrify half-hand.g -cg -eqn half-hand.cg.eqn -o half-hand.out
more half-hand.cg.eqn
```

drawastg half-hand.out | ghostview -

Note also how petrify removed the redundant arc $Rt \rightarrow Ao$ in the half-hand_out output STG.

The implementation that petrify obtained for the STG with increased concurrency and the CSC problem in this case has exactly the same complexity (but a slightly different form) as the one obtained for the case without the CSC problem. This does not happen in general, and concurrency reduction is often a good way to trade off logic complexity and performance.

8. Derive implementations for generalized C elements \(1\) using the -gc option of petrify (you could also prevent petrify from generating the output STG file by using option -no instead of -c) and examine the petrify.log report file about alternative implementations:

```plaintext
petrify half-hand.g -gc -eqn half-hand.gc.eqn -o half-hand.gc.eqn
more half-hand.gc.eqn
more petrify.log
```

The petrify.log file also reports technology independent performance information such as:

- Average and maximum number of output events between any ordered pair of input events. If input events are considered “slow” then this determines in a very abstract manner (by considering all GC elements to have the same delay) the performance of the system, and is determined only by the input specification.

In the part of the file labeled Input -> Input Delays the numbers in parentheses denote the number of output events between the corresponding pair of input events. You should look at the output STG to understand the numbers, an consider that the path $Rt \rightarrow Ro \rightarrow Ao \rightarrow Ro \rightarrow Ai$ is not a worst-case path under the hypothesis of slow input events, because $Ai$ will always determine the firing time of $Ro$.

\(1\) i.e., logic blocks of the form $y = ab + ac + cb$ where $a, b, c$ can be arbitrary product terms. The generalized C elements have a very efficient transistor-level implementation (cf. lecture notes).
12. Find implementation without latches at all, using combinational gates with possible feedbacks using -molatch option. Observe the result of signal insertion (map0) into the STG using draw_stg utility.

```
petrify half-hand.g -molatch -tm -eqn half-hand.molatch.eqn -o half-hand,eqn more half-hand.molatch.eqn
```

To summarise our use of latches, here is the comment from the petrify manual page:

The -latch str option specifies a restricted set of latches to be used for synthesis; str can contain any string of characters from the set CDRS, which respectively corresponds to the following latches: Muller C element, D latch, reset dominant SR latch and set dominant SR latch. These latches are only used if found in the library. In case this option is not specified, any asynchronous latch in the library is used.

13. Find a timed circuit implementation using the relative timing option -topt. Specify timing constraints for half-hand, eqn based on the assumption that input events are slower (i.e., occur later) than the output events that have common predecessors (an exactly the same predecessors) with those inputs, e.g., time A<<0AI, which corresponds to introducing a partial fundamental mode. Examine the contents of report file petrify.log. In this case there is only one timing constraint, so petrifyeither uses it or does not. In general, only some constraints might be needed to justify the correctness of a given solution.

```
petrify half-hand.g -topt -eqn half-hand.topt.cg.eqn -o more half-hand.topt.cg.eqn
```

```
more petrify.log
```

Explain why adding other timing assumptions between inputs and outputs, such as time A<<0AI and time R<<B, may be wrong if we have no more information about relative timing between the left and right hand side parts of the environment.

2 Task 2: 2-to-4 and 4-to-2 phase converters

2.1 What are 2-to-4 phase and 4-to-2 phase converters?

Figure 4 depicts the interface of 2-to-4 and 4-to-2 phase converters.

A 2-to-4 phase converter can be described by the following regular expression:

```
(req2;req4;ack4;req1;ack1;ack2)*
```

Here events req2 and ack2 stand for either rising or falling transitions on the 2-phase handshake.

A 4-to-2 phase converter can be described similarly by the following regular expression (as well as by other, more convenient, specifications described in the 4-to-2 exercise):

```
(req4;req2;ack2;ack4;req1;ack1;ack2)*
```
2.2 Exercise

2.2.1 Part 1: 2 to 4 phase converter

1. Specify the behavior of a 2 to 4 phase converter in astg format. Save this specification in file conv24.g. (Alternatively, find a sample solution conv24.g in directory solutions and examine it carefully.)

2. Check if your STG has CSC-violations:

   `write_sg -bin conv24.g | draw_astg -bin | ghostview -`

3. Obtain STG specifications with only rising (+) and falling (-) transitions, using the `-untog` option:

   `petrify -untog conv24.g -o conv24.untog.g`

4. Resolve CSC-conflicts manually and compare the solution with the one given by petrify. For the latter, use the following script:

   `petrify conv24.g -cg -eqn conv24.cg.eqn -o conv24.out`

   `more conv24.cg.eqn`

   `draw_astg conv24.out | ghostview -`

5. Obtain an implementation with SRD latches:

   `petrify conv24.g -latch SRD -tm -eqn conv24.srd.eqn -no`

   `more conv24.srd.eqn`

6. Derive monotonic covers for the output signals (`petrify -mc`):

   `petrify conv24.g -mc -eqn conv24.mc.eqn -o more conv24.mc.eqn`

   Compare this solution with the previous one.

2.2.2 Part 2: 4 to 2 phase converter

Figure 5 shows three different specifications for a 4 to 2 phase converter, with different degrees of concurrency between inputs and outputs. The third version has the highest potential performance, because it has only one output transition between any two input transitions, and it performs the 4-phase reset (`ack4+` and `req4-`) completely in parallel with the 2-phase handshake. Of course, its logic cost is also greater.

![Diagram of 4 to 2 phase converters](image)

Figure 5: Specifications of 4 to 2 phase converters.

1. Obtain an STG specification for each one of them (the example shows that for Figure 5(a) with only rising (+) and falling (-) transitions (use the `-untog` option in petrify):

   `petrify -untog conv42_seq.g -o conv42_seq.untog.g`

   `draw_astg conv42_seq.untog.g | ghostview -`

2. Derive a netlist for each one of the specifications (the example shows that for Figure 5(a)) using:

   `petrify conv42_par.untog.g -cg -eqn conv42_par.eqn -o conv42_par.out`

   `more conv42_par.eqn`

   `draw_astg conv42_par.out | ghostview -`

Note how the number of CSC conflicts, and hence the number of state signals and the logic complexity, grows with the increase in concurrency. As usual, performance increase roughly implies area increase.
3. Derive a netlist for the same specification (the example shows that for Figure 5.(c)) using timing assumptions, such as simultaneity conditions:

   .time req2=ack4+/10ack2-, req4-/1
   .time req2+ack4+/ack2-, req4-

   The first constraint means that for both ack2- and req4-/1 firing times of req2+ and that of ack4+ are undistinguishable. Hence ack2- could be enabled by req4-/1 just as well (instead of req2+ as in the original specification) without changing the observable firing sequences.

   For that, use:

   petrify conv42_par, untag, s-topt -cg -eqn conv42_par.t.eqn -o conv42_par.t.o more conv42_par.t.eqn
   draw astg conv42_par.t.o | ghostview -

   Compare the solutions with the untimed implementation.

   Examine the contents of the petrify.log report file.

3 Task 3 (advanced): VME bus controller

Figure 6 depicts the interface of a slave device to a VME bus. What is shown here is the result of an abstraction of the main synchronization core between the bus and the device links, separately from all remaining logic. The latter performs address and opcode decoding, error detection and some other functions that are outside this controller.

The behavior of the controller is as follows: a request to read from or write into the device is received by one of the signals DSr or DSw respectively. In a read cycle, a request to read is sent to the device through signal LDS. When the device has the data ready (LDTACK), the controller must open the transceiver to transfer data to the bus (signal D), which is a Data Enable signal; it controls the transceiver together with a direction signal provided in the bus, namely it closes one latch and opens the tri-state in one direction, and opens the other latch in the other). In the write cycle, data is first transferred to the device by opening the transceiver (D). Next, a request to write is sent to the device (LDS). Once the device acknowledges the reception of the data (LDTACK) the transceiver must be closed to isolate the device from the bus. Each transaction must be completed by a return-to-zero of all interface signals, seeking for a maximum parallelism between the bus and the device operations.

1. Construct an STG specifying the behavior of the VME bus controller.

2. Solve CSC using petrify.

3. Obtain a complex gate implementation of the circuit (option -cg).

4. Derive a netlist of gates by first decomposing the circuit into 3 input gates and then mapping onto a gate library (-lit3 -tm).

5. Optimize the implementation (option -topt) by adding relative timing constraints to your STG model, such as the assumption that the bus and device handshakes are slower compared to output and internal signal transitions.

6. Modify the specification of the VME bus controller in such a way that, instead of two ‘dual rail’ input strobes DSw and DSr (an “already decoded” version), the interface to the bus consists of a single strobe DS and an opcode signal (WR). The value of WR determines what operation the controller is supposed to start when strobe DS goes high. Let it be write if WR = 1 and read if WR = 0. We can assume that in the environment that when the DS is low the value of WR can arbitrarily change by as soon as the DS goes high the state of the WR is stable. (Hint: Use a pair of complementary places to represent the state of WR. The token between these places can toggle by transitions WR+ and WR-, only when the value of DS is zero. Then the firing of transition DS+ must disable transitions WR+ and WR-. This disabling does not lead to output non-persistence because both WR and DS are inputs.)

7. Obtain a complex gate implementation for the modified STG. Compare it with the previous solution.