Decomposing Specifications with Concurrent Outputs to Resolve State Coding Conflicts in Asynchronous Logic Synthesis

Hemangee K. Kapoor
kapoorhk@lsbu.ac.uk

Mark B. Josephs
josephmb@lsbu.ac.uk

Centre for Concurrent Systems and VLSI,
Faculty of BCIM, London South Bank University,
London SE1 0AA, UK

ABSTRACT

Synthesis of asynchronous logic using the tool Petrify requires a state graph with a complete state coding. It is common for specifications to exhibit concurrent outputs, but Petrify is sometimes unable to resolve the state coding conflicts that arise as a result, and hence cannot synthesise a circuit. A pair of decomposition heuristics (expressed in the language of Delay-Insensitive Sequential Processes) are given that helps one to obtain a synthesisable specification. The second heuristic has been successfully applied to a set of nine benchmarks to obtain significant reductions both in area and in synthesis time, compared with synthesis performed on the original specifications.

Categories and Subject Descriptors:
B.6.3 [**Logic Design**]: Design Aids – *Automatic Synthesis, Hardware description languages*

General Terms:
Design, Experimentation, Languages

Keywords:
Asynchronous logic synthesis, delay-insensitive decomposition

1. INTRODUCTION

Petri nets, interpreted as Signal Transition Graphs (STGs) [2], are widely used to specify asynchronous control circuits. The tool Petrify [3] inputs such a description and converts it into a state graph (SG) prior to synthesis. Construction of Petri nets manually is cumbersome and error prone. More conveniently, the front-end tool di2pn takes a program in the language of Delay-Insensitive Sequential Processes (DISP) and automatically generates a Petri net [6].

In order to synthesise a circuit, Petrify requires an SG to have a complete state coding (CSC). That is, no two reachable markings of the net may be encoded by the same signal values (i.e. correspond to the same state) unless the same output signals are excited for each of them [2].

When an SG does not have CSC, the specification needs to be modified. One possibility is to change the dependencies between external signal transitions, e.g. [9], which may or may not be acceptable. Another approach is to introduce internal signals. Petrify takes the latter approach to solve CSC. It employs heuristics to insert internal signals (extra state variables) in order that different markings might correspond to different states. These heuristics are based upon an analysis of the quiescent and excitation regions of the SG [4].

In the situation of specifications with concurrent outputs, it is sometimes difficult for Petrify to solve CSC and hence synthesise a circuit. In this paper we provide the designer with heuristics that can be readily applied to decompose such specifications into a form in which Petrify can solve CSC. The language of DISP in which this decomposition is carried out is high level compared to SGs and Petri nets. It facilitates detection and manipulation of such outputs, tasks that would be non-trivial working with a graphical structure.

1.1 Related Work

Decomposition, not specific to concurrent outputs, is considered in [11], where the input STG is decomposed into a set of components. This is achieved by partitioning the set of output signals and generating components that produce these outputs. A similar approach was taken previously in [2].

Petrify itself automatically applies a set of heuristics to solve CSC conflicts, but sometimes the results are sub-optimal. Alternatively, interactive insertion of state signals can be tried out. A tool has been developed [8] which helps the user to visualise sets of transitions causing conflicts and facilitates manual refinement of an STG at the level of unfolding prefixes.

Finally, CSC conflicts can be avoided by performing structural transformations on Petri nets [1].

1.2 Summary

The remainder of this paper is organised as follows. We outline the description language DISP in section 2. Section 3 shows how concurrent outputs can cause CSC conflicts. In section 4 we propose a pair of decomposition heuristics to resolve CSC conflicts and illustrate their application. The

DAC 2004, June 7–11, 2004, San Diego, California, USA.

second heuristic was found to be applicable to some benchmark examples and the results of doing so are evaluated in section 5.

2. DISP

Delay-Insensitive Sequential Processes (DISP) is a variant of CSP [5] and DI-Algebra [7]. It is a description language that can be used in the synthesis of asynchronous control circuits with the help of the tools di2pn and Petrify. The input DISP specification is first translated into a Petri net using the tool di2pn and then Petrify is used to obtain a netlist for the circuit. More information about the language and the synthesis procedure can be found in [6].

We claim that it is more convenient for the designer to describe circuits in DISP (a high-level language), rather than in the graphical notation of Petri nets. The tool di2pn automatically translates from the former to the latter.

2.1 Language syntax

The following syntax will be used in this paper to describe processes:

proc	::=	*ioburst* | `select` *choice* `end` | *proc* ; *proc*
		| *proc* `par` *proc* | `forever do` *proc* `end`
choice	::=	*ioburst* [`then` *proc*] [`alt` *choice*]
ioburst	::=	*siglist*/*siglist*

siglist is a list of signal names; these must be distinct and their order is unimportant. The simplest process is an input/output burst (*ioburst*) where transitions of all the signals in the input burst must be absorbed before transitions of all the signals in the output burst are produced.

`select` and `end` delimit a process from a choice between *ioburst*-guarded processes; these are separated by `alt` and their order is unimportant. Choice is restricted to those guarded processes for which all required input transitions are available. For example, the process `select` a/b `alt` $-/c$ `end` eventually outputs c and terminates, unless input a arrives, in which case it may instead output on b and terminate.

Processes can be composed in sequence and in parallel, and infinite repetition is provided by the `forever` construct.

Asynchronous control circuits must be modelled by *non-terminating* processes. This is the smallest class of processes satisfying the following rules: an infinite repetition is non-terminating; the sequential or parallel composition of two processes is non-terminating if either process is non-terminating; a choice is non-terminating if all of its guarded processes are non-terminating.

3. CSC CONFLICTS

To synthesise circuits, Petrify requires an SG to have CSC. If there are conflicting states in the graph, additional (internal) signals are required and their corresponding up-going and down-going transitions are inserted in the SG. This modification is based on the theory of regions [4] and involves computing the excitation region for the new event. The new SG is then checked for CSC and the process is repeated until CSC is resolved.

The state of confusion leading to CSC conflicts arises when distinct outputs are required of the circuit from states with the same coding. We have identified one common cause, namely, when the specification requires concurrent outputs and these outputs are absorbed by different components in the environment. In such a situation the environment may respond to these outputs with corresponding input events in an arbitrary order. This can lead to CSC conflicts, as can be seen from the following example.

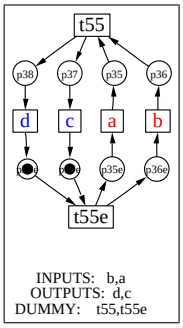

Figure 1: Petri net for P in environment E1

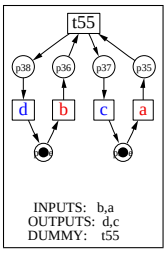

Figure 2: Petri net for P in environment E2 par E3

3.1 Illustration of conflict from concurrent outputs

Consider a circuit specified by the following DISP process (P) and its corresponding environment (E1).

E1 = forever do -/a,b ; c,d/- end
#environment E1
P = forever do a,b/c,d end

The process P synchronises on a and b (before outputting c and d), whereas E1 synchronises on c and d. The Petri net generated by di2pn for this specification is shown in Figure 1. It corresponds to an SG that has no CSC conflicts and Petrify synthesises a speed-independent (SI) circuit with an estimated area of 18 units (the number of literals in the Boolean equations generated by Petrify).

Consider once more process P, but this time assume its environment has more parallelism, i.e., instead of one process (E1) now two parallel processes (E2 par E3) represent the environment.

E2 = forever do -/a ; c/- end
E3 = forever do -/b ; d/- end
#environment E2 par E3

Figure 2 shows the Petri net resulting from this specification. Here the two environment components respond to c and d independently. It is now the case that Petrify needs to add one extra signal to resolve CSC conflicts before it can synthesise an SI circuit with an estimated area of 11 units.

Thus failure of the environment to synchronise on the concurrent outputs led to CSC conflicts. These are shown as shaded states in Figure 3. When in state 1001 (0110) the circuit is confused as to whether to produce output event c+(-) or d-(+). The situation has arisen because the environment handles the concurrent outputs independently: it is free to supply an input as soon as it receives the corresponding output. For example, from state 1100, after d+ has been output, E3 can produce the transition b- before the transition c+ has been observed.

4. DECOMPOSITION

In this section we give a pair of heuristics that separate out Forks from the specification. These heuristics reduce concurrency in the output bursts of DISP processes.

(H1) Consider a non-terminating process P and a list $y = y_1, \ldots, y_n$, where $1 < n$, of distinct signal names

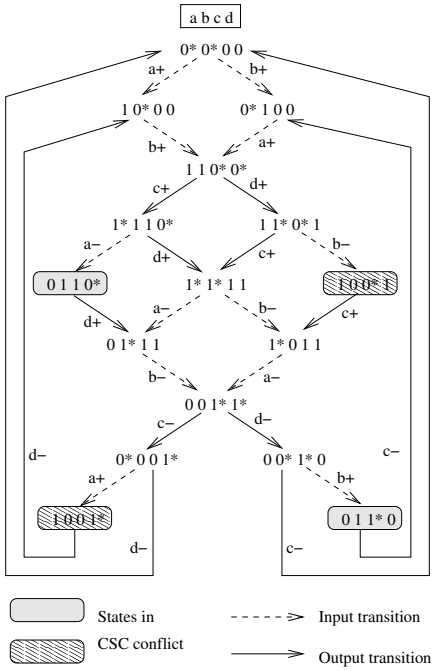

Figure 3: State graph for P in environment E2 par E3

that always occur together in its output bursts, i.e., if y_i occurs in an output burst of P then so does y_j, $0 < i, j \leq n$. Let x be a fresh signal name, let P' be the process formed by substituting x for each occurrence of y in P, and let $FORK =$ `forever do` x/y `end`. Then one can decompose P into P' `par` $FORK$.

(H2) Consider a non-terminating process P and a list $y = y_1, \ldots, y_n$, where $0 < n$, of distinct signal names that always occur together in its output bursts. Let x and z be fresh signal names, let P' be the process formed by substituting x followed by an input/output burst (z/t) for each output burst (y, t) that contains y in P, and let $FORK =$ `forever do` $x/y, z$ `end`. Then one can decompose P into P' `par` $FORK$.

Forks provide the cheapest way of generating concurrent outputs. H1 reduces concurrency in the specification by replacing multiple outputs (y) by a single one (x). H2 reduces concurrency by removing one or more outputs (y) from each burst in which y occurs. Note that H2 is often effective for $n = 1$.

4.1 Example 1 : Application of H1

Applying H1 to the process P (of section 3.1 with environment E2 par E3), we can resolve the CSC conflicts. To decompose this specification, we replace the output-burst c,d by using a fresh signal name s1 and a FORK component as shown below (E2 and E3 are unchanged):

FORK = forever do s1/c,d end

P1 = forever do a,b/s1 end

Figure 4 shows the Petri net for P1. The state graphs for P1 and FORK are shown in Figure 5. Neither component has any CSC conflicts and synthesis using Petrify gives SI circuits with area 7 and 2, respectively. (There is no need to run Petrify to synthesise a Fork, of course!).

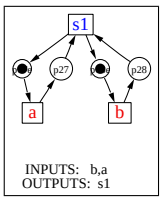

Figure 4: P1

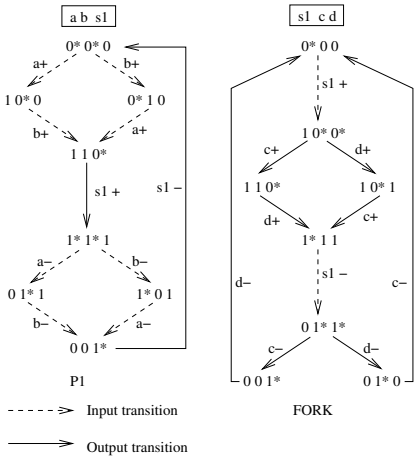

Figure 5: State graph for P1 and FORK

4.2 Example 2 : Application of H2

Consider a 2x1 Decision-Wait element with forked outputs. This can be described in DISP as follows:

L = forever do select -/a0 then d0/- alt -/a1 then d1/- end end

R = forever do -/b ; c/- end

#environment L par R

Q = forever do select a0,b/d0,c alt a1,b/d1,c end end

In each cycle the process L chooses non-deterministically between a0 and a1. If the process Q receives input on a0 and b, it produces output on d0 and c. Similarly, if it receives input on a1 and b, it produces output on d1 and c. Petrify can synthesise a circuit with area 88 units after adding 2 state variables.

We use this example to illustrate the application of H2. Here we might select the signal c as a candidate to be forked out. We use the fresh signals x and z to perform the decomposition as shown below:

Q' = forever do

  select a0,b/x then z/d0 alt a1,b/x then z/d1 end end

FORK = forever do x/c,z end

Q' has no CSC conflicts and synthesis using Petrify gives an SI circuit with area 45.

5. EXPERIMENTAL RESULTS

We found that H2 could be successfully applied to a number of benchmark examples [10, 12, 13] and obtained the results shown in Table 1. These experiments were performed using Petrify 4.2 on a 1.4 GHz Pentium 4 machine with 256 MB RAM. The table shows the number of state signals added by Petrify to synthesise the circuit, the estimated area of the circuit for a generalised C element implementation, the total number of set and reset pins required and the time required to generate the solution. They are given for the specifications before and after decomposition.

In the case of pscsi-trcv [12] and SCSI-fast-initiator-send [13], Petrify could not solve CSC on the original specification, but could successfully synthesise the decomposed specification.

The other cases can be summarised by calculating the geometric mean: synthesis for "Decomposed" was on average 2.7-times faster than direct synthesis, saving 25% in area,

Table 1: Experimental Results

Circuit	Original				Decomposed				CSC Ratio	Area Ratio	SR Ratio	Time Ratio
	N i	A I	SR p1	T t1	N ii	A II	SR p2	T t2	ii/i	II/I	p2/p1	t2/t1
scsi-isend	4	103	7	135	2	77	4	70	0.5	0.75	0.57	0.52
scsi-tsend	2	84	3	30	1	65	3	13	0.5	0.77	1	0.43
scsi-trcv	2	90	3	47	1	72	2	17	0.5	0.80	0.67	0.36
pscsi-isend	5	83	5	50	2	59	2	12	0.4	0.71	0.4	0.24
pscsi-tsend	4	92	3	25	3	71	3	11	0.75	0.77	1	0.44
pscsi-ircv	2	37	3	10	1	24	1	2	0.5	0.65	0.33	0.20
pscsi-trcv	×	-	-	-	1	33	2	3	-	-	-	-
isend-fast	×	-	-	-	4	83	2	90	-	-	-	-
sbuf-send	3	72	5	12	1	57	3	6	0.33	0.79	0.60	0.50

× = Petrify could not solve CSC conflicts; N = number of state signals inserted by Petrify;
A = area in literals; SR = total number of set-reset pins; T = time taken in seconds for synthesis

52% in state variables inserted by Petrify, and 39% in set-reset pins required.

In all cases the original specifications required 2-5 state signals to be added by Petrify and 3-7 set-reset pins. Their decomposed versions required fewer state signals and at worst the same number of set-reset pins, in all cases decreasing the area of the circuit.

6. CONCLUSION

During the synthesis of asynchronous control circuits Petrify applies a set of heuristics to resolve state coding conflicts. In cases where the specification exhibits concurrent outputs, it is sometimes difficult for Petrify to resolve them. We have given a pair of decomposition heuristics which can be readily applied to help Petrify to rapidly synthesise area-efficient circuits.

The decomposition heuristics introduce Fork elements that preserve the delay-insensitive behaviour typically required of asynchronous controllers. They offer a practical approach to delay-insensitive decomposition of specifications, where each component can be implemented as a speed-independent circuit.

As future work we are investigating Wire-decomposition in addition to Fork-decomposition. Each wire would introduce a state variable, cf. z in H2. An application would be self-contained blocks, i.e., processes in which every signal is transitioned an even number of times and therefore the start state is indistinguishable from the finish state. Such blocks can be found in counters, for example.

Acknowledgements

The support of the European Commission under FP5 contract IST-1999-29119 (ACiD-WG) is acknowledged. The authors would like to thank Prof. Steven Nowick for providing the burst-mode specifications for the benchmark examples, Prof. Jordi Cortadella for providing the latest build of Petrify for synthesising the benchmarks, and Dr. Dennis Furey for the development of the tool di2pn. Development of di2pn was supported in part by the UK EPSRC grant reference M51567. The authors are grateful to the anonymous referees for their helpful comments.

7. REFERENCES

[1] J. Carmona, J. Cortadella, and E. Pastor. A Structural Encoding Technique for the Synthesis of Asynchronous Circuits. *Fundamenta Informaticae*, 34:1–23, 2002.

[2] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Department of Electrical Engineering and Computer Science, June 1987. MIT/LCS/TR-393.

[3] 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*, 3(E80-D):315–325, 1997.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. A Region-based Theory for State Assignment in Speed-Independent Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):793–812, August 1997.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

[6] M. B. Josephs and D. P. Furey. A Programming Approach to the Design of Asynchronous Logic Blocks. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design Advances in Petri-Nets, LNCS*, volume 2549, pages 34–60. Springer Verlag, 2002.

[7] M. B. Josephs and J. T. Udding. An Overview of D-I Algebra. *System Sciences, 1993, IEEE Proc. of the Twenty-Sixth Hawaii International Conference*, 1:329–338, January 1993.

[8] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev. Visualisation and Resolution of Encoding Conflicts in Asynchronous Circuit Design. *IEE Proc. on Computers and Digital Techniques*, 150(5):285–293, September 2003.

[9] R. Manohar. An Analysis of Reshuffled Handshaking Expansions. *Proc. of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 184–193, March 2001.

[10] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical Asynchronous Controller Design. *International Conference on Computer Design, ICCD*, pages 341–345, October 1992.

[11] W. Vogler and R. Wollowski. Decomposition in Asynchronous Circuit Design. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design Advances in Petri-Nets, LNCS*, volume 2549, pages 152–190. Springer Verlag, 2002.

[12] K. Y. Yun and D. L. Dill. Automatic Synthesis of 3D Asynchronous Controllers. *International Conference on Computer-Aided Design, ICCAD*, pages 576–580, November 1992.

[13] K. Y. Yun and D. L. Dill. A High-Performance Asynchronous SCSI Controller. *International Conference on Computer Design, ICCD*, pages 44–49, October 1995.