Figure 7: The resulting implementation of for a speed-independent version of the target-send burst-mode portion of the SCSI controller.

Figure 5: A portion of the cyclic constraint graph for the target-send burst-mode portion of the SCSI controller after a successful decomposition.

Figure 6: The resulting implementation of the target-send burst-mode portion of the SCSI controller.

Figure 3: The generalized C-element structure and corresponding production rules.

Figure 4: A portion of the cyclic constraint graph for the target-send burst-mode portion of the SCSI controller after decomposition.

Figure 1: A portion of the cyclic constraint graph for the target-send burst-mode portion of the SCSI controller.

Figure 2: Production rules for several common gates.

Example *(max_size)*	Original Specification | E |	| R |	ATACS Original Area (xtors)	ATACS Original Latency (inv)	ATACS Original CPU (sec)	ATACS Decomposed Area (xtors)	ATACS Decomposed Latency (inv)	ATACS Decomposed Iter (num)	ATACS Decomposed CPU (sec)	SIS Area (xtors)	SIS Latency (inv)	SIS CPU (sec)
tsbm(4)	29	41	86	8.4	1.9	90	4.1	1	1.5	149	11.1	22.8
tsbm(3)	29	41	86	8.4	1.9	90	4.1	1	1.5	121	9.2	19.0
tsbm(2)	29	41	86	8.4	1.9	114	4.6	3	8.4	131	7.5	18.9
tsbmSI(4)	29	41	94	8.4	21.4	110	7.6	18	64.7	213	15.1	21.5
tsbmSI(3)	29	41	94	8.4	21.4	118	6	19	98.3	197	11.9	19.0
tsbmSI(2)	29	41	94	8.4	21.4	182	9.2	21	161.4	195	9.6	19.7
mmuSI(4)	17	27	54	5.6	0.0	62	5.3	1	0.1	134	10.3	2.9
mmuSI(3)	17	27	54	5.6	0.0	78	5.3	3	0.6	103	10.0	1.7
mmuSI(2)	17	27	54	5.6	0.0	121	4.9	5	1.6	131	8.5	2.1
scsi1(4)	49	66	118	5.9	5.8	146	5.9	1	9.6	-	-	-
scsi2(4)	49	77	117	5.9	5.8	125	5.9	1	6.8	-	-	-
scsi2SI(4)	49	77	124	5.9	2.7	140	5.9	1	1.4	313	16.1	25.7
scsi3(4)	49	72	129	12.1	6.0	147	9.6	12	123.5	-	-	-
4-bit counter(4)	93	93	94	7.9	26.5	104	7.6	3	88.5	96	11.5	0.8

Table 1: Comparision of designs generated by ATACS and those generated by SIS. In SIS, Lavagno's asynchronous synthesis procedure is used for the speed-independent designs, and the synchronous synthesis procedures are used for the 4-bit synchronous counter design. The timed asynchronous designs could not be synthesized by SIS.

Example *(max_size)*	Original Specification | E |	| R |	SYN Original Area (xtors)	SYN Original Latency (inv)	SYN Original CPU (sec)	SYN Decomposed Area (xtors)	SYN Decomposed Latency (inv)	SYN Decomposed Iter (num)	SYN Decomposed CPU (sec)	SIS Area (xtors)	SIS Latency (inv)	SIS CPU (sec)
tsbmSIGL(4)	29	41	172	10.8	52.6	174	10.7	18	549.6	177	16.9	21.5
tsbmSIGL(3)	29	41	172	10.8	52.6	194	13.6	19	1002.5	169	13.1	19.0
tsbmSIGL(2)	29	41	172	10.8	52.6	356	18.1	21	3102.6	171	11.3	19.7
mmuSIGL(4)	17	27	90	6.6	0.5	98	8.4	1	1.2	118	12.3	2.9
mmuSIGL(3)	17	27	90	6.6	0.5	100	7.1	3	6.0	91	11.6	1.7
mmuSIGL(2)	17	27	90	6.6	0.5	126	10.3	5	26.2	115	10.1	2.1

Table 2: Comparision of designs generated by SYN and those generated by SIS.

[9] Al Davis, Bill Coates, and Ken Stevens. The post office experience: Designing a large asynchronous chip. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, pages 409–418. IEEE Computer Science Press, 1993.

[10] P. Siegel, G. DeMicheli, and D. Dill. "Automatic Technology Mapping for Generalized Fundamental-Mode Asynchronous Designs". In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993.

[11] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[12] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, 1991.

[13] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[14] K. Y. Yun and D. L. Dill. Unifying synchronous/asynchronous state machine synthesis. In *Proceedings IEEE 1993 ICCAD* Digest of Papers. IEEE Computer Society Press, 1993.

[15] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical asynchronous controller design. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.

efficient implementations. Implementations generated using SYN still tend to do better than SIS for libraries with gates of at least a maximum fanin of 3, but they do not do as well when the library size is very small.

The last example in Table 1 is a 4-bit synchronous counter which illustrates another possible application of this work implementing synchronous circuits using asynchronous sequential logic such as generalized C-elements. For a 4-bit synchronous counter, ATACS generates an implementation with a gate that has 5 transistors in series which is easily decomposed resulting in an implementation which is slightly larger, but it is more than 30 percent faster than a design produced by SIS's synchronous design procedures [11].

5 Conclusions

We have proposed an efficient procedure for the decomposition of high-fanin gates in the synthesis of asynchronous circuits. Our procedure takes an event-rule system specification and a production rule set implementation which contains large gates that must be decomposed. A subset of valid decompositions are examined by generating a new specification and resynthesizing it. Our procedure derives implementations with a minimum logic depth that are hazard-free without addition of delay elements. The resulting decompositions are both area efficient and deliver high performance. Using ATACS, our asynchronous designs require up to 50 less area and are more than 50 percent faster than designs generated by SIS. We also have shown that our procedure can be used to design synchronous circuits using asynchronous sequential logic.

Our results also show that using generalized C-elements can yield smaller and faster implementations than using standard gates. In the past, the use of generalized C-elements has been limited since they can only be built with up to 4 transistors in series. By applying our decomposition procedure, however, hazard-free decompositions can be found which allows the applicability of using generalized C-elements to real examples. Since generalized C-elements do not currently appear in standard cell libraries, we have also shown that our procedure can be applied to gate-level synthesis procedures such as SYN yielding circuits with higher performance and comparable area than SIS for gate libraries with a maximum fanin of at least 3.

Acknowledgments

The authors would especially like to thank Peter Beerel of Stanford University for numerous invaluable discussions on the decomposition of high-fanin gates. We would also like to thank Professor David Dill and Polly Siegel of Stanford University for their advice and comments on this manuscript. Our thanks also go to Tom Rokicki of Stanford University for assistance in verifying several of our designs.

References

[1] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[2] A. J. Martin. Programming in VLSI: from communicating processes to delay-insensitive VLSI circuits. In C.A.R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, 1990.

[3] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[4] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messershmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.

[5] P. A. Beerel and T. H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings IEEE 1992 ICCAD* Digest of Technical Papers, pages 581–586, 1992.

[6] P. A. Beerel and T. H.-Y. Meng. Logic transformations and observability don't cares in speed-independent circuits, 1993. In collection of papers of the *ACM International Workshop on Timing Issues in the Specification of and Synthesis of Digital Systems*.

[7] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *International Conference on Computer Design, ICCD-1991*. IEEE Computer Society Press, 1991.

[8] K. Y. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D asynchronous state machines. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.

It is possible, however, that a single production rule could be used to generate multiple rising or multiple falling transitions. In this case, a new signal would be added which has multiple rising and falling transitions associated with it.

3.4.4 Incorporation with Other Synthesis Procedures

So far, we have only described using our decomposition procedure with ATACS, but it is sufficiently general to use with any procedure which can have its specification translated to and from an ER system and can produce an implementation in the form of a PR set. For example, it is straightforward to transform an ER system to a STG, a specification form used by many synthesis methods [3] [4] [12], by simply removing the timing constraints, and from a STG to an ER system by adding timing constraints of $[0, \infty]$. Also, since ATACS produces state graphs as an intermediate step, these state graphs can be passed to SYN, a system for synthesis of gate-level speed-independent circuits, which uses them as its specification, and SYN can then produce PR set implementations. Examples of both timed and speed-independent circuits generated by ATACS and gate-level speed-independent circuits generated by SYN are described in the next section.

4 Results

The decomposition procedure described in this paper has been automated in a CAD tool which has been incorporated into the synthesis procedures used by ATACS and by SYN, and all results reported in Tables 1 and 2 were compiled using this tool. The gate library used for each example contained gates with a maximum fanin size as specified in parentheses next to the name of the example. To indicate an example's size, the number of events and rules in the original specification is given. Area is reported as the implementation's transistor count. Latency is the longest delay through a block of logic generating an output transition driving a fanout of 4, and it is reported normalized to the delay of a single inverter (about $300ps$ for $0.8\mu m$ CMOS process at nominal conditions). The latency for implementations generated by ATACS and SYN are determined using HSPICE simulations. The CPU time reported is for a DECstation 5000. For each example, both the implementation before and after decomposing high-fanin gates is presented. Note that the original implementation is the same regardless of the size of the library. Finally, the number of iterations necessary to decompose the high-fanin gates is shown.

We compare our results to implementations generated using SIS with a library that includes: an inverter, 2 to *max_size* input NAND gates, 2 to *max_size* input NOR gates, and generalized C-elements with any combination of 1 to *max_size* transistors in the pullup and 1 to *max_size* transistors in the pulldown. For the timed circuits, environment delay information is given to SIS. All gate delay numbers in the library were derived using HSPICE. Area and delay calculations are performed using SIS's *print_area* and *print_delay* commands.

Using ATACS, both a timed and speed-independent version of the target-send burst-mode (*tsbm* and *tsbmSI*) cycle of a SCSI data transfer controller [14] are synthesized. Using SYN, a gate-level speed-independent implementation (*tsbmSIGL*) is synthesized. For each, we report the decomposition results for gate libraries with a *max_size* of 4, 3, and 2 inputs. The tables also include a speed-independent version of a memory management unit (*mmuSI*) [1] generated by ATACS and a gate-level implementation (*mmuSIGL*) generated by SYN. The next four examples in Table 1 are versions of the complete SCSI data transfer controller (*scsi1*, *scsi2*, *scsi2SI*, and *scsi3*) [15]. The example *scsi1* most closely resembles the specification given in [15]. In order to implement it speed-independently, the concurrency must be constrained yielding the implementations *scsi2* and *scsi2SI*. Finally, *scsi3* is a very concurrent implementation that can only be implemented with timing.

Mapping designs generated by ATACS to a library with a maximum size of 4 has a cost of about 15 percent in area compared to designs with an unlimited fanin, but designs are over 15 percent faster. If we reduce the library to a maximum size of 3, the cost in area goes up to 20 percent, and the performance is slightly improved. Finally, if we reduce the library to a maximum size of 2, we find that the area cost is now nearly 80 percent, and performance is worse than for larger libraries. Mapping designs generated by SYN to a library of limited fanin has a cost in area from 4 percent to 80 percent and a cost of in delay from 10 percent to 65 percent depending on the size of library used. This is due to the fact that each signal is generated by 2 to 3 levels of logic before optimizations, and currently, logic optimizations are not being fully employed. We believe using them would improve our results.

When we compare our results to those obtained by SIS using a gate library with a maximum size of 4, we find that our implementations generated by ATACS require about half the area and perform more than 50 percent faster. For smaller libraries, the area and performance benefits diminishes, but ATACS consistently produces more

Algorithm 3.1 sends this new specification to the synthesis procedure ATACS to produce a new implementation. The new implementation contains a production rule with 7 literals:

$$DSend_i \wedge \neg DSend_o \wedge \neg Ready_o \wedge \neg DRel_o \wedge \neg Req_o \wedge \neg Done_o \wedge \neg Empty_o \quad \mapsto \quad x_1 \uparrow,$$

so there are still high-fanin gates. Therefore, Algorithm 3.2 is called again to find the next valid decomposition for this production rule which is D_6 with $X = \{\neg DAck_i, \neg DSend_o\}$. This decomposition also does not succeed, as neither do D_9 and D_{10}. The decomposition D_{12} with $X = \{\neg DSend_o, \neg Ready_o\}$ depicted in Figure 5, however, does produce a successful decomposition. The resulting implementation using generalized C-elements is shown in Figure 6.

3.4 Extensions

3.4.1 Searching Context Decompositions First

In order to speedup the search for a successful decomposition, we first try decompositions which use only literals from the context set in the extracted set. This is a small subset of the decompositions in which we have found most of our successful decompositions. This is accomplished by adding a fourth property to Definition 3.4:

4. $X \subseteq C$,

during the first pass. If no successful decomposition is found, we remove the fourth property and make another pass.

Using the fourth property, the first valid context decomposition in our example would be D_{12} which also leads to a successful decomposition. Thus, we only need to examine one decomposition rather than five. In fact, in this case, none of the decompositions including trigger signals in the extracted set lead to a successful decomposition.

3.4.2 Handling Exceptions by Using Multiple Level Decompositions

Algorithm 3.1 attempts to decompose each high-fanin gate by adding only one additional signal, and if it is not successful it returns an exception. In order to handle this situation, we iterate our breakup procedure to produce multiple levels of logic. If Algorithm 3.1 fails, we take the decomposition which had the minimum gate size and use its corresponding specification and implementation as input to a new iteration of Algorithm 3.1. This step is repeated until an implementation is returned that either uses only gates in the library or its minimum gate size is no longer decreasing. In the second case, our procedure is unable to generate an implementation using the given specification and gate library. To handle this situation, the requirements in either the specification or the gate library must be relaxed. So far, in our experimentations, the only examples that we have come across which could not be decomposed were when using small gate libraries that consisted of only 2 and 3-input gates.

To illustrate multiple level decompositions, let us return to our example, but derive a speed-independent implementation by assigning all timing constraints to be $[0, \infty]$. Now, Algorithm 3.1 is unable to successfully decompose the gate needed for $DReq_o$ to gates with a maximum gate size of 4. The minimum gate size of the various implementations found is 5, such as the one given below:

$$\begin{aligned} DSend_i \wedge \neg DAck_i \wedge \neg DSend_o \wedge \neg Empty_o \wedge x_1 &\mapsto DReq_o \uparrow \\ \neg x_1 &\mapsto DReq_o \downarrow \\ \neg Ready_o \wedge \neg DRel_o \wedge \neg Req_o \wedge \neg Done_o &\mapsto x_1 \uparrow \\ Ready_o &\mapsto x_1 \downarrow \end{aligned}$$

Using this modified ER system and PR set, we apply Algorithm 3.1 again. The resulting implementation meeting the constraints is shown in Figure 7.

3.4.3 Handling Multiple Rising and Falling Transitions

In our example, there is only a single rising and falling transition on each signal in a cycle, but our procedure can easily be applied to specifications in which signals are specified to have multiple rising and falling transitions in a cycle. If there is a different production rule associated with each of these transitions, then our procedure does not need any modification as it simply adds a new signal for each production rule that needs to be decomposed.

implementation in which both the guard being decomposed and the guard for the new signal are not too large, then this extracted set is used again. Otherwise, the extracted set corresponding to the next valid decomposition is found. If no remaining decompositions remain to be checked, an exception is raised.

Algorithm 3.3 *(Find Next Extracted Set)*
literal_set find_next_extracted_set(rule_set R_{prev}*; PR_set* P_{prev}*; literal_set* T*,* G*; event* t_{s_j}*,* t_{x_j}*; int max_size)* {
    int v*;*
    if $(size(G) \leq max_size)$ *then return*(T);
    if $(R_{prev} == \emptyset)$ *then* $X = \emptyset$*;*
    else {
        take $\langle G' \mapsto t_{s_j} \rangle$ *and* $\langle D \mapsto t_{x_j} \rangle$ *from* P_{prev}*;*
        $X = \{l_i \mid (\langle event(l_i), t_{x_j}, \varepsilon, \tau \rangle \in R_{prev})\}$;
        if $((size(G') \leq max_size)$ *and* $(size(D) \leq max_size))$ *then return*(X);
    }
    $v = val(X, G)$;
    do {
        if $(v < (2^n - 1))$ *then* v^{++} *else return(EXCEPTION);*
        $X = \{l_i \mid (l_i \in G)$ *and* $(2^{i-1} \wedge v)\}$;
    } *while* (*not valid*(X, T, max_size));
    return(X);
}

3.3.4 Modifying the Rule Set

Algorithm 3.4 modifies the rule set of a specification to reflect the decomposition of a guard defined by the extracted set. It first adds rules from the events corresponding to the literals in the extracted set to a transition on a new signal. It then removes all rules from these literals to the guarded transition associated with the guard being decomposed. Finally, to retain the ordering specified by these the rules, the algorithm adds a rule from the transition on the new signal to the guarded transition.

Algorithm 3.4 *(Modify Rule Set)*
rule_set modify_rule_set(rule_set R*; literal_set* X*; event* t_{s_j}*,* t_{x_j}*; int max_delay)* {
    foreach $l_i \in X$ {
        $R = R + \{\langle event(l_i), t_{x_j}, \varepsilon, [0, max_delay] \rangle\}$;
        $R = R - \{\langle event(l_i), t_{s_j}, \varepsilon, [0, max_delay] \rangle\}$;
    }
    $R = R + \{\langle t_{x_j}, t_{s_j}, \varepsilon, [0, max_delay] \rangle\}$;
    return(R);
}

3.3.5 Example

In our example, Algorithm 3.1 determines that there is one production rule, the one for the transition $DReq_o \uparrow$, with a guard which contains more than 4 literals, so it calls Algorithm 3.2 to find a new specification corresponding to a valid decomposition. Algorithm 3.2 first adds the events $x_1 \uparrow$ and $x_1 \downarrow$ to the set of events. Next, it calls Algorithm 3.3 to find the extracted set for the first valid decomposition of the guard on the event $DReq_o \uparrow$. As determined earlier, the first valid decomposition is D_5 with $X = \{DSend_i, \neg DSend_o\}$. For the falling transition $DReq_o \downarrow$, since the guard has a size of 1 which is less than our *max_size*, the trigger set $T = \{Ready_o\}$ is the extracted set for this guard. Next, Algorithm 3.4 is called to modify the set of rules for our decomposition. First, it adds the rule $\langle DSend_i \uparrow, x_1 \uparrow, 0, [0,7] \rangle$ and removes the rule $\langle DSend_i \uparrow, DReq_o \uparrow, 0, [0,7] \rangle$ since $DSend_i$ is in the extracted set of the decomposition. It also adds the rule $\langle DSend_o \downarrow, x_1 \uparrow, 1, [0,7] \rangle$ because $\neg DSend_o$ is in the extracted set. In this case, there is no rule that needs to be removed since the literal $\neg DSend_o$ is a context signal in the original guard. Lastly, it adds the rule $\langle x_1 \uparrow, DReq_o \uparrow, 0, [0,7] \rangle$. Applying Algorithm 3.4 for the falling transition, the rule $\langle Ready_o \uparrow, x_1 \downarrow, 0, [0,7] \rangle$ is added, and the rule $\langle Ready_o \uparrow, DReq_o \downarrow, 0, [0,7] \rangle$ is removed because $Ready_o$ is in the extracted set for the guard on the falling transition. The rule $\langle x_1 \downarrow, DReq_o \downarrow, 0, [0,7] \rangle$ is also added. A portion of our new constraint graph is shown in Figure 4.

system to a hazard-free implementation describable by a PR set. The search is terminated once an implementation which contains no high-fanin gates is found or once all valid decompositions have been examined. We handle the second case by using multi-level decompositions discussed later.

Algorithm 3.1 *(Decompose High-Fanin Gates)*
PR_set decompose(ER_system $\langle E, R\rangle$*; PR_set* P*; int max_size, max_delay)* {
  rule_set $R' = \emptyset$;
  PR_set $P' = P, P_{prev} = P$;
  while $(\max_{\langle G \mapsto t\rangle \in P_{prev}}\{size(G)\} > max_size)$ {
    $\langle E', R'\rangle = find_next_decomps(\langle E, R\rangle, R', P, P_{prev}, max_size, max_delay)$;
    if $(\langle E', R'\rangle == EXCEPTION)$ *then return(EXCEPTION);*
    $P' = SYNTHESIS(\langle E', R'\rangle)$;
    if $(P' == EXCEPTION)$ *then* $P_{prev} = P'$;
  }
  return(P_{prev});
}

3.3.2 Finding the Next Decompositions

The core of our decomposition procedure is Algorithm 3.2 which finds the next valid decomposition for each high-fanin gate and the corresponding new specification consisting of a new event set and a new rule set. The algorithm examines the guards associated with each signal's rising event and falling event. If either guard, contains more literals than the maximum allowed, the algorithm augments the original event set with two new events. These two events are the rising and falling transitions of a new internal signal which is used to to decompose the large guards. Algorithm 3.2 then calls Algorithm 3.3 to find the extracted sets for the next decomposition of each guard. If either extracted set cannot be found, then all decompositions have been examined and have failed; hence, an exception is raised. Finally, Algorithm 3.2 calls Algorithm 3.4 to modify the original rule set to reflect the decompositions defined by the extracted sets.

Algorithm 3.2 *(Find Next Decompositions)*
ER_system find_next_decomps(ER_system $\langle E, R\rangle$*; rule_set* R_{prev}*; PR_set* P*,* P_{prev}*; int max_size, max_delay)* {
  literal_set $T^{\uparrow}, T^{\downarrow}, X^{\uparrow}, X^{\downarrow}$;
  foreach $((\langle G^{\uparrow} \mapsto s_j\ \uparrow\rangle \in P)$ *and* $(\langle G^{\downarrow} \mapsto s_j\ \downarrow\rangle \in P))$
    if $((size(G^{\uparrow}) > max_size)$ *or* $(size(G^{\downarrow}) > max_size))$ *then* {
      $E = E + \{x_j\ \uparrow, x_j\ \downarrow\}$;
      $T^{\uparrow} = \{l_i \mid (\langle event(l_i), s_j\ \uparrow, \varepsilon, \tau\rangle \in R)\}$;
      $T^{\downarrow} = \{l_i \mid (\langle event(l_i), s_j\ \downarrow, \varepsilon, \tau\rangle \in R)\}$;
      $X^{\uparrow} = find_next_extracted_set(R_{prev}, P_{prev}, T^{\uparrow}, G^{\uparrow}, s_j\ \uparrow, x_j\ \uparrow, max_size)$;
      if $(X^{\uparrow} == EXCEPTION)$ *then return(EXCEPTION);*
      $X^{\downarrow} = find_next_extracted_set(R_{prev}, P_{prev}, T^{\downarrow}, G^{\downarrow}, s_j\ \downarrow, x_j\ \downarrow, max_size)$;
      if $(X^{\downarrow} == EXCEPTION)$ *then return(EXCEPTION);*
      $R = modify_rule_set(R, X^{\uparrow}, s_j\ \uparrow, x_j\ \uparrow, max_delay)$;
      $R = modify_rule_set(R, X^{\downarrow}, s_j\ \downarrow, x_j\ \downarrow, max_delay)$;
    }
  return$(\langle E, R\rangle)$;
}

3.3.3 Finding the Next Extracted Set

Algorithm 3.3 finds the next extracted set which defines the next decomposition to be used for a large guard. It is called twice for each high-fanin gate by Algorithm 3.2, once for the guard on the rising event and once for the guard on the falling event, but only one of these guard must be large. For guards which are not large, the extracted set is set equal to the trigger set of the original guard. If the guard does need to be decomposed, the first step is to determine the extracted set corresponding to the previously attempted decomposition. In the case of the first attempted decomposition (i.e., $R_{\text{prev}} == \emptyset$), the extracted set is initialized to the empty set. In all other cases, the extracted set is initialized using the previous specification and implementation. If this extracted set led to an

and its context set is $C = \{ \neg DSend_o, \neg Ready_o, \neg DRel_o, \neg Req_o, \neg Done_o, \neg Empty_o \}$ with the value 252. The guard for the production rule on the event $DReq_o \downarrow$ is $G = \{ Ready_o \uparrow \}$, its trigger set is $T = \{ Ready_o \uparrow \}$ with the value 1, and its context set is $C = \emptyset$ with the value 0.

3.2 Valid Decompositions

Each production rule in a PR set that has more literals in the guard than the maximum allowed by the gate library needs to be decomposed. A decomposition of a production rule as defined in Definition 3.3 is a partition of the set of literals in its guard into two subsets: an *extracted set* and a *reduced set*. The literals in the extracted set are used as trigger signals for a transition on a new signal that is added to decompose the guard. The literals in the reduced set along with the new signal are used in the new implementation of the guard being decomposed.

Definition 3.3 *(Decomposition) A decomposition of the production rule $\langle G \mapsto t \rangle$ is any partition of the set of literals in the guard G into two subsets, an extracted set X and a reduced set G' (i.e., $G' = G - X$). A decomposition is denoted by D_v where $v = val(X, G)$.*

For a particular production rule with n literals in its guard, there are 2^n different possible decompositions. Fortunately, we do not need to examine all of them. First, since the goal of our procedure is to find a decomposition which leads to an implementation with the minimum logic depth, and since all decompositions lead to exactly one additional level of logic over the original implementation, we can terminate with the first successful decomposition found. We can also eliminate many decompositions which will never lead to a successful decomposition. For example, we can eliminate all decompositions where the extracted set only includes one literal. In this case, the decomposed guard is not reduced in size since only one literal is removed from the original guard and it is replaced with a literal from the new signal. We can also eliminate decompositions where the extracted set of the decomposition includes more literals than the maximum allowed, since it will result in a guard for the new signal which is too large. Finally, if all the trigger signals are used in the extracted set of the decomposition, the high-fanin problem is simply transferred to the new signal, since all the same context signals will be needed. Therefore, only *valid decompositions* as defined in Definition 3.4 below are examined.

Definition 3.4 *(Valid Decomposition) For a production rule $\langle G \mapsto t \rangle$ with a trigger set T, a decomposition with an extracted set X is valid if it has the following properties:*

1. *size*$(X) \geq 2$,
2. *size*$(X) \leq$ *max_size, and*
3. $T \not\subseteq X$.

It is important to note that a valid decomposition does not always lead to a successful decomposition. The reason for this is that after making a decomposition, we only know the size of the trigger set for each guard. To find the size of the context sets, the circuit must be resynthesized.

For the production rule on the event $DReq_o \uparrow$ from our example, there are 256 possible decompositions $D_0, D_1, D_2, \ldots, D_{255}$. The literals are ordered as in the production rule in the previous section, $l_1 = DSend_i$, $l_2 = \neg DAck_i$, ..., $l_8 = \neg Empty_o$. The first decomposition is D_0 with $X = \emptyset$ which is not valid since the extracted set does not include at least two literals. Similarly, the decompositions D_1 with $X = \{DSend_i\}$ and D_2 with $X = \{\neg DAck_i\}$ are not valid because they also do not include at least two literals. The decomposition D_3 with $X = \{DSend_i, \neg DAck_i\}$ has two literals, but it is not valid because it contains all the trigger signals (i.e., $T = X$). The decomposition D_4 with $X = \{\neg DSend_o\}$ is also not valid because it only contains one literal in the extracted set. Therefore, the first valid decomposition found is D_5 with $X = \{DSend_i, \neg DSend_o\}$ which satisfies the three conditions given in Definition 3.4.

3.3 Algorithms for Decomposing High-Fanin Gates

3.3.1 Searching the Valid Decomposition Space

In order to decompose high-fanin gates, Algorithm 3.1 searches through the space of valid decompositions. For each set of valid decompositions examined, it constructs a new specification using Algorithm 3.2 described in the next subsection. This new specification is resynthesized using any synthesis procedure which translates an ER

In the example specification shown in Figure 1(a), the production rule implementation generated by ATACS for the output signal $DReq_o$ is given in Figure 1(b). Note that the guards include additional literals added by the synthesis procedure to remove hazards, as well as, the literals associated with rules in the specification. A direct implementation of these production rules would result in a gate with 8 transistors in series which is not practical in existing technologies. In the following section, we use this gate as an example to show how high-fanin gates can be decomposed and implemented from a given gate library.

2.3 Gate Library

The gate library used by our procedure is assumed to contain either single gate generalized C-elements as depicted in Figure 3(b) or ANDs, ORs, and the 2-input C-element necessary to construct them as shown in Figure 3(a). The library is characterized by two numbers. The *max_size* is the maximum number of transistors in series in a pullup or pulldown transistor stack in a generalized C-element or the maximum fanin of the gates in the library. The *max_delay* is the maximum delay for the largest generalized C-element in the library whether implemented as a single gate or multiple gates.

For our timed implementations synthesized using ATACS, we use a library of single gate generalized C-elements with a *max_size* of 4 and a *max_delay* of 7 inverter delays which was determined by HSPICE simulations of the largest gates in the library. For our gate-level speed-independent implementations synthesized using SYN, we use a library of ANDs, ORs, and 2-input C-elements with a *max_size* of 4 and an unbounded *max_delay*.

3 Procedure for the Decomposition of High-fanin Gates

Given an ER system specification and a PR set implementation, we want to implement the circuit using only gates in the given gate library. Our procedure searches for a decomposition of each high-fanin gate into the minimum number of levels of logic necessary such that all gates needed are found in the given gate library. This is accomplished by adding new signals to the original specification guided by the original implementation. The rules governing transitions on the new signals are added in such a way that the new signal can be used to reduce the size of the guard being decomposed without changing the concurrency originally specified. Each possible decomposition results in a modified specification which is then resynthesized to obtain a new implementation which is guaranteed to be hazard-free since it is derived by a hazard-free synthesis procedure. If no decomposition results in an implementation without high-fanin gates, then the specification which results in the implementation that requires the smallest fanin is taken as the new starting point. To this specification, an additional signal is added to decompose each remaining high-fanin gate. This procedure terminates either when all high-fanin gates have been successful decomposed, or when the minimum fanin of the best implementation is no longer decreasing. In this section, we explain our decomposition procedure in more detail.

3.1 Trigger and Context Signals

Before describing our decomposition procedure, we need to introduce the notions of *trigger signals* and *context signals* in a production rule. A trigger signal is a literal which has a corresponding rule in the specification and is needed in the guard to guarantee that the guarded transition occurs after the enabling event in the rule. A context signal is an additional literal which is added by the synthesis procedure to prevent hazards. Breaking down the guard into these two sets of signals will facilitate the search for a decomposition. These sets can be easily found as described in Definition 3.1. Also, in dealing with subsets of a guard, it is useful to associate a value with each subset as defined in Definition 3.2.

Definition 3.1 *(Trigger Signals and Context Signals) Given an ER system $\langle E, R\rangle$ and a production rule $\langle G \mapsto t\rangle$, a literal l_i is in the set of trigger signals T if the rule $\langle event(l_i), t, \varepsilon, \tau\rangle$ is in R, where $event(l_i) = s\uparrow$ if $l_i = s$ and $event(l_i) = s\downarrow$ if $l_i = \neg s$, and a literal l_i is in the set of context signals C if it is in G but not in T (i.e., $C = G - T$).*

Definition 3.2 *(Subset Value) Given a guard $G = \{l_1, l_2, \ldots, l_n\}$ and a subset S of the literals in the guard, the value of this subset is: val$(S, G) = \sum_{(i \text{ s.t. } l_i \in S)} 2^{i-1}$.*

In our example, the guard for the production rule on the event $DReq_o\uparrow$ is $G = \{\ DSend_i, \neg DAck_i, \neg DSend_o, \neg Ready_o, \neg DRel_o, \neg Req_o, \neg Done_o, \neg Empty_o\ \}$, its trigger set is $T = \{\ DSend_i, \neg DAck_i\ \}$ with the value 3,

2.1 Event-Rule System

ER systems were initially introduced for performance analysis of asynchronous circuits by Burns [13] and were later modified for specification of timed asynchronous circuits by Myers [1]. An ER system is composed of a set of atomic actions, *events*, and the causal dependencies between them, *rules*. In circuit specifications, events are *transitions* of specified signals from one value to another. There are two transitions associated with each specified signal s, namely, $s\uparrow$ where $\uparrow$ denotes that the signal s is changing from a low to high value, and $s\downarrow$ where $\downarrow$ denotes that the signal s is changing from a high to low value. Each rule is composed of an *enabling event*, an *enabled event*, and a *bounded timing constraint*. A rule is said to be *satisfied* if the amount of time which has passed since the enabling event has exceeded the lower bound of its timing constraint. A rule is said to be *expired* if the amount of time which has passed since the enabling event has exceeded the upper bound of its timing constraint. An event cannot occur until *all* rules enabling it are satisfied. An event must always occur before *every* rule enabling it has expired. Thus, the causal dependency is *conjunctive*.

An ER system can be specified using an *ER schema* and initialization information. An ER schema defines a cyclic constraint graph which is a weighted marked graph in which the vertices are the events, the arcs are the rules, and the weights are the bounded timing constraints. Each rule of the form $\langle e, f, \varepsilon, \tau \rangle$ is represented in the graph with an arc connecting the enabling event e to the enabled event f. The arc is weighted with the bounded timing constraint τ. In other words, each rule corresponds to a graph segment, $e \xrightarrow{\tau} f$ (or $e \overset{\tau}{\bullet\!\!\rightarrow} f$ when the rule is initially marked, i.e., $\varepsilon = 1$). ER schemas can be easily derived from many other asynchronous specification languages such as communicating sequential processes, burst-mode state machines, and signal transition graphs [1]. The ER schema is defined more formally as follows:

Definition 2.1 *(Event-Rule Schema) An event-rule schema is a pair $\langle E, R \rangle$ where E is a finite set of events, and R is a finite set of rules. Each rule is denoted as $\langle e, f, \varepsilon, \tau \rangle$, where e and f are two events, ε is defined to be 1 if the rule has an initial marking and 0 otherwise, and τ is the timing constraint of the rule in the range $[l, u]$ where l is the lower-bound and u is the upper-bound.*

In order to construct an ER system from an ER schema, each event in the ER schema is mapped onto an infinite number of event occurrences, each corresponding to a different occurrence of that event. The rules are similarly mapped. The result is an *infinite acyclic constraint graph*. This transformation is important for timing analysis [1], but for the purposes of specification in this paper, we deal exclusively with the cyclic constraint graph formed from the ER schema.

An example of a portion of a cyclic constraint graph is shown in Figure 1(a). This specification is derived from the target-send burst-mode cycle of a SCSI data transfer controller described in [14]. An example of a rule in this graph is: $\langle DAck_i\downarrow, DReq_o\uparrow, 1, [0,7] \rangle$. In this rule, $DAck_i\downarrow$ is the enabling event, $DReq_o\uparrow$ is the enabled event, and $[0,7]$ is the bounded timing constraint. Since $DAck_i$ and $DReq_o$ are initially low, this rule is initially marked so $\varepsilon = 1$.

2.2 Production Rule Set

PR sets were introduced by Martin to describe circuit implementations derived using his asynchronous synthesis procedure [2]. Each production rule in the PR set as defined in Definition 2.2 is composed of a *guard* and a *guarded transition*. The guard is a conjunction of literals that describes an implementation in which the guarded transition can occur when the guard evaluates to true.

Definition 2.2 *(Production Rule) A production rule is composed of a guard G and a guarded transition t, and it is denoted $\langle G \mapsto t \rangle$. The guard is a set $G = \{l_1, l_2, \ldots, l_n\}$ which represents a conjunction of literals, $G = l_1 \wedge l_2 \wedge \ldots \wedge l_n$. A literal is either a signal, $l_i = s$, or its negation, $l_i = \neg s$. The size of a guard is the number of literals in the set G, and it is denoted by size(G). The guarded transition is either a rising event, $t = s\uparrow$, or a falling event, $t = s\downarrow$, on a signal s.*

Figure 2 shows the production rules for an inverter, an *AND* gate, an *OR* gate, a *Muller C-element*, and a simple *generalized C-element* [2]. For the generalized C-element, an input is annotated with a "+" if it is only used in the guard for the rising transition, a "-" if it is only used in the guard for the falling transition, and no annotation means it is used for both guards. Our procedure requires that the circuit implementation initially be given in a generalized C-element structure as depicted in Figure 3(a). Note that while this structure is shown as a multi-level gate implementation, it can be implemented as a single compact gate as shown in Figure 3(b).

1 Introduction

In recent years, there has been a resurgence of interest in the design of *asynchronous circuits* due to increasing concerns in synchronous design relating to clock skew and power consumption. Synchronous design, however, has remained dominant for two reasons: difficulties in asynchronous design related to hazards and a lack of asynchronous CAD tools. The goal of our work is to address both of these issues by providing a fully automated procedure for the design of hazard-free asynchronous circuits using practical gate libraries for area efficient, high performance, and low power designs.

In previous work, Myers [1] described ATACS, a system which automatically synthesizes timed asynchronous circuits from specifications annotated with timing constraints. By analysis of the given timing constraints, it is possible to generate designs which are significantly simpler than designs using other asynchronous methodologies. The implementations generated, however, use gates with unlimited fanin which is not practical for existing technologies. In CMOS, for example, gates with more than four transistors in series are typically considered to be too slow, and they must be decomposed. In asynchronous design, however, decomposition of these large gates needs to be done is such a way as to not introduce hazards. This paper addresses the problem of finding such a hazard-free decomposition.

Many asynchronous synthesis methodologies such as the work by Martin [2], Chu [3], Meng [4], and Beerel [5] use a speed-independent delay model which assumes unbounded gate delay. This model produces robust designs since correctness is guaranteed independent of the delays associated with the gates in the implementation. However, these methods also have the limitation that their implementations may require gates with an unlimited fanin. Beerel [6] has investigated heuristic techniques guided by logic transformations to produce hazard-free decompositions of high-fanin gates, but they are limited to only combinational transformations and are often unsuccessful.

Other asynchronous synthesis procedures rely upon variations on the *fundamental-mode* assumption such as the work by Nowick [7], Yun [8], and the HP Mayfly Group [9]. This assumption essentially states that the environment waits long enough for the circuit to stabilize before the inputs are changed. Siegel [10] has shown that under this assumption synchronous technology mapping techniques can be applied with small modifications to account for hazards, and thus, do not require unlimited fanin gates. The fundamental-mode assumption, however, limits the concurrency that can be specified, and when timing analysis shows that this assumption does not hold in practice, delay elements must be added to the feedback path to guarantee that the timing constraints are satisfied, degrading the performance.

In SIS [11], Lavagno's asynchronous synthesis procedure [12] derives a complex-gate implementation under the speed-independent model, and then uses synchronous technology mapping to map the design to a given gate library. The resulting implementation is then analyzed with the timing information from the library, and if potential hazards are detected, delay elements are added to prevent them. We show that the cost of these delay elements is substantial in terms of both area and delay and can be avoided by using a hazard-free decomposition.

In this paper, we describe a systematic procedure to decompose high-fanin gates in order to map asynchronous designs to practical gate libraries without needing to add delay elements. Our procedure begins with a specification in the form of an event-rule (ER) system, a circuit implementation in the form of a production rule (PR) set, and a given gate library. For the purposes of specification, an ER system can be thought of as a signal transition graph (STG) [3] to which timing constraints have been added. Our procedure adds new signals to the specification which are used to decompose gates that are larger than the maximum in the library. The procedure then repeatedly modifies and resynthesizes the specification, and it terminates when either all gates have been successfully decomposed into the minimum number of levels of logic, or it determines that no solution is possible using the given gate library. We have automated this procedure and applied it to designs synthesized by both ATACS [1] and SYN [5]. Several practical examples are described, and comparisons are made with implementations obtained using SIS. Our procedure results in implementations using ATACS that are up to 50 percent smaller with half the latency than those produced by SIS.

2 Specification, Implementation, and Gate Library Descriptions

In order to decompose high-fanin gates, our procedure requires a specification in the form of an ER system, an implementation in the form of a PR set, and a gate library. In this section, we briefly describe each requirement.

Contents

1 Introduction **1**

2 Specification, Implementation, and Gate Library Descriptions **1**
2.1 Event-Rule System . . . 2
2.2 Production Rule Set . . . 2
2.3 Gate Library . . . 3

3 Procedure for the Decomposition of High-fanin Gates **3**
3.1 Trigger and Context Signals . . . 3
3.2 Valid Decompositions . . . 4
3.3 Algorithms for Decomposing High-Fanin Gates . . . 4
3.3.1 Searching the Valid Decomposition Space . . . 4
3.3.2 Finding the Next Decompositions . . . 5
3.3.3 Finding the Next Extracted Set . . . 5
3.3.4 Modifying the Rule Set . . . 6
3.3.5 Example . . . 6
3.4 Extensions . . . 7
3.4.1 Searching Context Decompositions First . . . 7
3.4.2 Handling Exceptions by Using Multiple Level Decompositions . . . 7
3.4.3 Handling Multiple Rising and Falling Transitions . . . 7
3.4.4 Incorporation with Other Synthesis Procedures . . . 8

4 Results **8**

5 Conclusions **9**

Copyright ©1994

Chris J. Myers and Teresa H.-Y. Meng

AUTOMATIC HAZARD-FREE DECOMPOSITION OF HIGH-FANIN GATES IN ASYNCHRONOUS CIRCUIT SYNTHESIS

Chris J. Myers and Teresa H.-Y. Meng

Technical Report: CSL-TR-94-651

December, 1994

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055
pubs@shasta.stanford.edu

Abstract

In this paper we present an automated procedure to decompose high-fanin gates generated by asynchronous circuit synthesis procedures for technology mapping to practical gate libraries. Our procedure begins with a specification in the form of an event-rule system, a circuit implementation in the form of a production rule set, and a given gate library. For each gate in the implementation that has a fanin larger than the maximum in the library, a new signal is added to the specification. Each valid decomposition of the high-fanin gates using these new signals is examined by resynthesis until all gates have been successfully decomposed, or it has been determined that a solution does not exist. The procedure has been automated and used to decompose high-fanin gates from several examples generated by the synthesis tools ATACS and SYN. Our resulting implementations using ATACS, when compared with SIS which uses synchronous technology mapping and adds delay elements to remove hazards, are up to 50 percent smaller and have less than half the latency using library delays generated by HSPICE.

Key Words and Phrases:

Computer-aided design, asynchronous circuit synthesis, timed circuits, event-rule systems, production rules, generalized C-elements, decomposition of high-fanin gates, and technology mapping.

AUTOMATIC HAZARD-FREE DECOMPOSITION OF HIGH-FANIN GATES IN ASYNCHRONOUS CIRCUIT SYNTHESIS

Chris J. Myers
Teresa H.-Y. Meng

Technical Report: CSL-TR-94-651

December, 1994

This research was supported by an NSF fellowship, ONR Grant no. N00014-89-J-3036, and research grants from the Center for Integrated Systems, Stanford University and the Semiconductor Research Corporation, Contract no. 92-DJ-205.