Universität Augsburg

STG Decomposition: Optimised Backtracking and Component Reduction

Mark Schaefer

Report 2005-13 September 2005

Institut für Informatik
D-86135 Augsburg

Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE

STG Decomposition: Optimised Backtracking and Component Reduction*

Mark Schaefer

University of Augsburg, Germany
schaefer@informatik.uni-augsburg.de

Abstract. The synthesis of asynchronous circuits is a difficult and time-consuming task. Outgoing from a Petri net based behavioural description – signal transition graphs – there exist methods to decompose this description into smaller components in order to improve synthesis.
This paper deals with the decomposition method of [VW02,VK05] and introduces several methods for the improvement of efficiency. These methods are discussed and compared by the means of benchmark examples.

Keywords: Asynchronous circuit, STG, Petri net, decomposition, speed-independent

1 Introduction

Asynchronous circuits are a promising type of digital circuits. They perform better, use less energy and emit less radiation than conventional synchronous circuits. A widely used formalism for their modelling are *signal transition graphs* or *STGs*, which are interpreted Petri nets.

The main drawback of this model is the inefficient and complex synthesis into real-life circuits; for this, the reachability graph of an STG is needed, which could lead to state explosion and - even worse - the synthesis needs an effort which is at least quadratic in size of this reachability graph.

One way to avoid this, is to decompose an STG into several smaller ones which perform together in the same way as the original one. The advantages are a faster synthesis and a reduced peak memory usage. Other methods are for example synthesis with net unfoldings[KKY04] and direct mapping[Ebe87,Hol82]. This paper deals with the decomposition method of [VW02,VK05]. In particular, four methods to improve the efficiency and quality of the components are introduced and discussed.

The next section gives a condensed overview of the field of asynchronous circuits, STGs and decomposition. The third section introduces the new decomposition methods followed by the results of some benchmark examples and their discussion. The paper ends with a conclusion and an outlook to future work.

For more information about asynchronous circuits, STGs and decomposition, see [VW02,VK05,CKK+02].

2 Circuits, STGs and Decomposition

A *signal* is a model of a physical wire, it has a boolean value of *0* or *1* depending on the interpretation of the voltage of the wire. In the following we will abstract from the physical reality and only talk about signals and boolean values.

* This work was supported by the DFG-project 'STG-Dekomposition' Vo615/7-1.

A *signal edge* is a change of the value of a signal, either from 0 to 1 called *rising edge* and denoted by a '+' after the signal name, or from 1 to 0 called *falling edge* and denoted by a '−'.

An *asynchronous circuit* or just *circuit* is an electrical device with *input* signals which are controlled by the *environment* of the circuit and *output* signals which are controlled by the circuit itself; *internal* signals are output signals, which are not observed by the environment, e.g. signals for internal communication. A circuit calculates a boolean function depending on its input *and* (usually) its output signals. This function is a sufficient description of the circuit. Normally, a circuit is built up of some elementary circuits – called *gates* – which calculate basic boolean functions like *and*, *not* or *xor*. Every output of a gate is an output of the circuit, either a real one or an internal one.

An STG (which is a Petri net, see below) may contain transitions labelled with λ called *dummy* transitions. They are a design simplification and describe no physical reality. They play an important part in our decomposition algorithm where they are called *divining* transitions. To *lambdarise* a transition means to change its label to λ, to *delambdarise* it means to change the label back to the initialvalue.

To keep the notation short, input/output/internal signal edges are just called input/output/internal edges. The set of transitions labelled with a certain signal is sometimes identified with the signal itself, e.g. lambdarising signal a means to change the label of all transitions labelled with $a+$ or $a-$ to λ.

Synthesis is the calculation of a function describing a circuit from a formal behavioural description, e.g. an STG, under observance of some *(timing) constraints*. We use the *speed-independent* model with the following properties:

- Input and outputs edges can occur in an arbitrarily order[1].
- Signals (wires) are considered to have no delay, i.e. a signal edge is received immediately by all listeners.
- The circuit must work properly according to its formal description under arbitrarily delays of each gate.

An *STG* is an interpreted Petri net, which describes the behaviour of an asynchronous circuit *and* assumptions on the environment (Figure 1). The transitions are labelled with signals edges and the interpretation is as follows:

- The firing rule is as usual.
- If a transition labelled with an input edge is activated, the circuit described by the net must be ready to receive this signal edge from the environment.
- If a transition labelled with an output/internal edge is activated, the circuit described by the net must produce this signal edge.

STGs can model more behaviour than a real-life circuit can show. The most important problem are *dynamic conflicts*, i.e. two transitions of an STG are enabled under some marking, and firing one would disable the other. This is a form of nondeterminism, which in most cases cannot be handled by a digital circuit. There are three problematic cases.

[1] For example, the *fundamental mode* allows only alternating input and output edges with a minimum temporal distance.

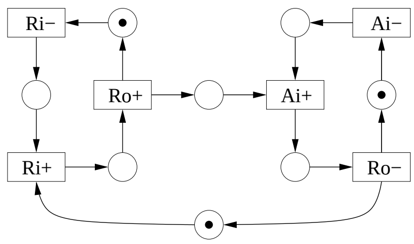

Fig. 1. Example of an STG: `pastor`. Inputs: `Ai, Ri` - Output: `Ro`

1. One transition is labelled with an input edge, the other with an output edge. This conflict is very hard to implement, since both signal edges are independently generated and may occur at the same time.
2. Both transitions are labelled with an output edge. A circuit which can handle such conflicts is called *arbiter* and cannot be implemented as a purely digital circuit. STGs with such conflicts can be handled by our decomposition method and new conflicts are not introduced. For a detailed discussion see [VW02,VK05].
3. An *auto-conflict*, i.e. both transitions are labelled with the same signal edge. This is a non-deterministic choice, which can hardly be handled by circuits. During our decomposition algorithm we consider such a newly generated conflict as an indication that too many signals were lambdarised in an STG. In this case *backtracking* is performed and a signal is delambdarised, see below for more details.

Observe that conflicts between dummy-transitions are ignored.

However, to detect dynamic conflicts one has to generate the reachability graph, which we want to avoid. Instead, we look for *structural conflicts*, i.e. two transitions with a common place in their presets. This is a necessary precondition for dynamic conflicts, which can be checked structurally. Consequently, in the decomposition method of [VW02] it is only looked for structural conflicts, each of them treated as a dynamic one.

The improved decomposition algorithm of [VK05] makes it possible to ignore structural auto-conflicts, i.e. to consider them not as indications for a dynamic auto-conflicts. This results in three different strategies for the handling of structural conflicts:

1. *Conservative strategy*: As in [VW02] every structural conflict is considered as a dynamic one.
2. *Risky strategy*: Structural conflicts are ignored.
3. *Interactive strategy*: Ask a human if a structural conflict is dynamic or not.

Despite of its name the risky approach seems quite sensible: structural conflicts are very often only this and not dynamic ones, which leads to unnecessary backtracking when using the conservative method; furthermore, the decomposition algorithm preserves dynamic auto-conflicts, thus accidently generated ones will be detected by the synthesis tool and no erroneous circuit will be generated.

For a detailed description of the decomposition process see [VW02,VK05]. For this paper it is only important to know that we start with a collection of STGs called *initial components*, each of them a copy of the original STG N with some lambdarised signals and some former output signals being considered as input signals. The following operations are applied to each component; this process is called *reduction*:

– *Secure contraction* of dummy transitions
– Deletion of redundant places and redundant transitions
– *Backtracking*

The contraction of a transition t generates a set of new places: $\{(p,q)|p \in {}^\bullet t, q \in t^\bullet\}$ (each one of them inherits the tokens and arcs of its 'inner' places) and removes t, ${}^\bullet t$ and $t^\bullet$ from the net. Contractions are only performed if they are '*secure*' (implying language preservation) and *no new structural auto-conflict is generated.* It is easy to see that the contraction of a transition t increases the number of places by $|{}^\bullet t| \cdot |t^\bullet| - (|{}^\bullet t| + |t^\bullet|)$.

Backtracking means to delambdarise a signal of the initial component, to consider it as an input signal and to start reduction anew. This is applied if there are still dummy transitions left but none of the other operations can be performed. In particular, if a contraction of a dummy-transition would generate a new structural auto-conflict, this is considered as an indication that too many signals of a component were lambdarised to produce their output signals appropriately; this can be changed by adding another input signal and – informally speaking – providing more information to the circuit.

The decomposition algorithm itself is non-deterministic. However, for some examples the order of operations is crucial for the final result in terms of circuit size or the number of added signals. The question is how to find a good order of operations to get the best possible result. Furthermore, backtracking means to undo all operations performed so far, which is very inefficient and the question is whether this is really needed. Viewing the reduction of all components together, a lot of work is done several times and the question is whether it is possible to reuse intermediate results for the reduction of other components. Answers to these questions are given in the next section.

3 Optimised Decomposition Algorithms

3.1 Version 2 - Ordering Transition Contractions

Although reduction is meant to be performed automatically it can be done with pen and paper. To keep this simple one would contract those transitions first, which generate the smallest number of new places. In the optimal case a dummy transition has only one place in its pre- and postset, thus its contraction would generate one new place while removing both old ones. But the contraction of a transition, with for instance 4 places and its pre- and 6 places in its postset would increase the number of places by 14. These 14 places are maybe adjacent to other dummy transitions and so on. Hence, contracting transitions in an unsuitable order can lead to an enormous increase in the number of places.

Contracting 'easy' transitions first turned out to be a good heuristic also for the automatic reduction. In version 2 the dummy transitions are sorted by the number of newly generated places if they would be contracted in the initial component. Then reduction works as in version 1, following this precalculated list of transition contractions. In order to avoid sorting after every redundant place deletion operation, this list is not updated during reduction.

3.2 Version 3 - Lazy Backtracking

In the original implementation, backtracking was performed by restarting the reduction of a component from the initial component. Of course, this method is quite natural and plays an important part in the proof of correctness in [VW02,VK05]. On the other hand, it can obviously be rather inefficient, e.g. in extreme cases backtracking might occur for the last dummy transition.

Naturally, if the reduction should not start anew at the beginning one has to introduce *savepoints* for intermediate STGs. Since backtracking affects signals rather than single transitions *lazy backtracking* contracts all transitions of signal a_0, then all transitions of signal a_1 and so on. After a signal was successfully contracted the resulting intermediate STG is used as a savepoint.

If backtracking has to be performed, it is unnecessary to start from the very beginning. Instead, it is possible to use the last suitable savepoint. This leads to the algorithm depicted in Figure 2.

$$N \stackrel{\lambda}{\Longrightarrow} N_0 \xrightarrow{a_0} N_1 \xrightarrow{a_1} \cdots \underbrace{N_k}_{\text{no conflict}} \xrightarrow{a_k} \underbrace{N_{k+1} \xrightarrow{a_{k+1}} \cdots N_{j-1} \xrightarrow{a_{j-1}} N_j}_{\text{conflict}} \underset{\text{for } a_j}{\xrightarrow{a_j}}$$

Fig. 2. Backtracking of Version 3

Starting from N, all initially useless signals are lambdarised yielding STG N_0. Instead of contracting them in an arbitrary order as in version 1, the dummy transitions are contracted grouped by their former signals as described above.

If this is possible, i.e. all contractions were secure and no new structural auto-conflict was generated, save the resulting STG as N_1. Next, try to contract signal a_1 in N_1 and so on. This results in a sequence (N_i) of STGs and a sequence (a_i) of contracted signals.

Probably, in some STG N_j the contraction of signal a_j is not possible. In version 1 one would delambdarise a_j in N_0 and start anew from there. Instead, delambdarise a_j in N_j resulting in N'_j and look for a structural auto-conflict of a_i.[2]

If there is no such conflict, proceed from N'_j with a new signal a'_j to be contracted.

If there is a conflict for a_j, one has to find the signal whose contraction caused it. To do this, consider STG N_{j-1} with a_j delambdarised resulting in N'_{j-1}. Doing this means to undo the last signal contraction of a_{j-1}. If the conflict for a_j disappeared, it is clear that this contraction caused the conflict.

If the conflicts still exits in N'_{j-1}, go back another step to N_{j-2} (undoing the two last contractions), delambdarise a_j again and check again for a conflict for a_j. Observe that the signals $a_{j-1}, a_{j-2}, \ldots$ are not delambdarised while going back.

If eventually an STG N_k, N'_k resp. is reached which does not have a structural auto-conflict for a_j, it is clear that the contraction of signal a_k caused the conflict of

[2] Such a conflict might exist, because conflicts between dummy transitions are ignored during reduction.

a_j, which becomes visible in N_j. Therefore, a_k has to be delambdarised in N'_k, too, resulting in N''_K. But now it is possible that there is a structural auto-conflict for a_k in N''_k. If there is none, proceed with the reduction from N''_k with a new dummy signal a''_k. Otherwise, go back to N_{k-1}, delambdarise a_j *and* a_k and look for a conflict of a_k and so on. This is performed with a growing set of signals to be delambdarised until an STG N_l is reached without a structural auto-conflict of the delambdarised signals, from which the reduction goes on.

If N_l is N_0, N contains a structural auto-conflict initially for some signal. This is possible for the improved decomposition version from [VK05], which allows such conflicts *provided that they are not dynamic ones*. In this case the respective signal can be delambdarise in N_0 safely.

Important for this method is that only the signals which could not be contracted and signals causing a structural conflict auto-conflict are delambdarised and therefore added to the final component while performing backtracking. The other signals whose contraction is only undone during backtracking are contracted again if the reduction is continued[3].

3.3 Version 4 - Tree Decomposition

The methods described so far are improvements for the decomposition of a single component. This section deals with a method for improving the *overall* efficiency of the reduction of all components.

If we take a look at examples of decomposition, it turns out that in most cases two components have many lambdarised signals in common. Therefore the existence of an intermediate STG C' should be possible, from which two or more components could be derived: instead of reducing both components independently, it is sufficient to generate C' only once and to proceed separately with each component afterwards, thus saving a lot of work.

We introduce *tree decomposition* by the means of an example: let N be an STG with the signal set $\{1, 2, 3, 4, 5\}$. Furthermore, let there be 3 components C_1, C_2, C_3 and the signals which were lambdarised intially in each component $\{1, 2, 3\}$, $\{2, 3, 4\}$, $\{3, 4, 5\}$. A possible intermediate STG C' for C_1 and C_2 would be the STG in which signals 2 and 3 have been contracted, see Figure 3.

In (a) the initial situation is depicted. There are three independent leafs labelled with the signals which should be contracted to get a component. In (b) C' is introduced as a common intermediate result of C_1 and C_2. In the (c) one can see nearly the same situation as in (b), but signals which were already contracted are embraced. This is a more operational view: each node is labelled with the signals which should be contracted when it is entered with some STG, see below. In (d) we merged C' and C_3 with the possible common intermediate result lablled with 3, yielding the final decomposition tree. In (i) there is a possible different tree for the same components.

Tree decomposition according to a decomposition tree works as follows (for a node u let $s(u)$ the signals with which it is labelled): enter the root node with the initial STG N without lambdarised signals. Whenever entering a node u with an

[3] Of course it is possible that they are delambdarised during another backtracking

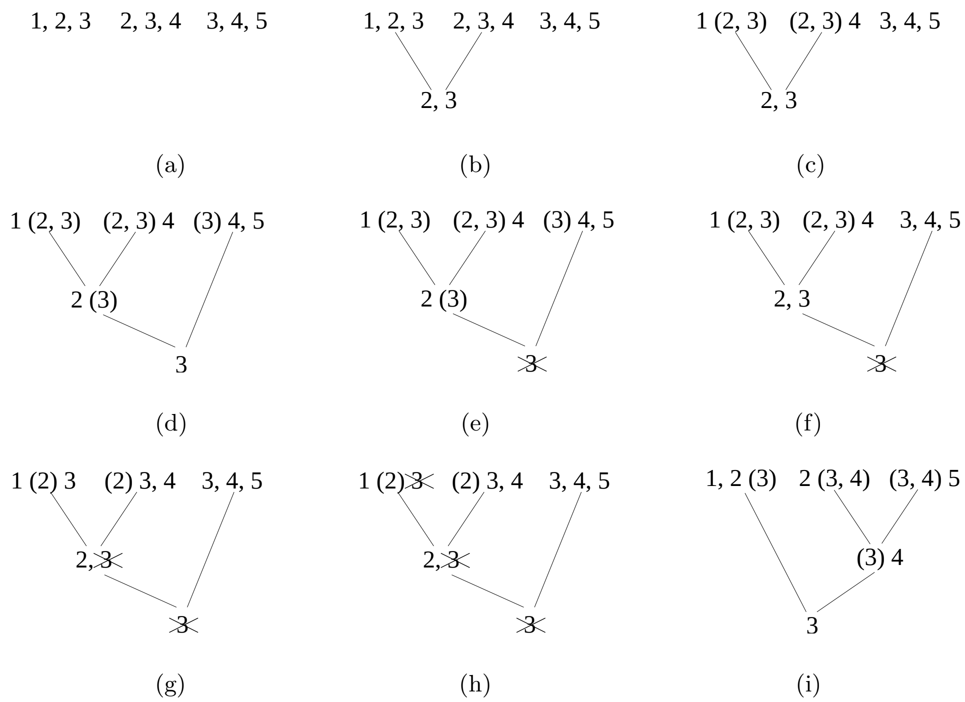

Fig. 3. Building of a simple decomposition tree. Leafs from left: components C_1, C_2, C_3. (a) initial situation (b) two components merged (c) already contracted signals embraced (d) final tree with all components (e)-(g) contraction of signal 3 not possible in root node and is therefore postponed to the childs (i) alternative tree

STG N', lambdarise the signals $s(u)$ in N', perform reduction as usual and enter each child node with its own copy of the resulting STG. If u is a leaf, the resulting STG is a final component.

Since this tree is precalculated from the initial components, it is very likely that not all signal contractions are possible in every node. If during the reduction of some node, a signal $a \in s(u)$ could not be contracted in N', it is postponed, i.e. the signal a is added to every child node of u (if there are any). This is reasonable, because the contraction of a may have caused an structural auto-conflict for a signal a', which is lambdarised deeper in the tree. After a' is eventually contracted the contraction of a could be possible. Moving signals in this way between nodes of the decomposition tree is also called backtracking.

For instance, assume that the contraction of signal 3 in the root node is not possible, because its contraction causes a conflict for signal 4, see Figure 3 (e). Signal 3 is therefore added to the inner node and the rightmost leaf in (f). In the rightmost leaf the contraction of signal 3 becomes eventually possible after the contraction of signal 4, but not in the inner node (g). Signal 3 is therefore added to the left and middle leaf, in the first one the contraction is again not possible, but in the latter one it finally is (h). Therefore the components C_2 and C_3 were generated as prearranged, only component C_1 has the additional signal 3.

Postponing signals in this way has two important properties: On the one hand it changes the precalculated decomposition tree in a way that it is possibly not optimal in the sense that overall as less as possible signal contractions were performed. Of course, there is no way to know such things in advance (this is the reason why backtracking is needed for the other versions) and more important on the other hand postponing is absolutely needed to keep the final components small.

Observe that – in contrast to lazy backtracking – once the decomposition of a node is finished, it is not necessary to come back to this node and to delambdarise additional signals. Since signals are lambdarised just in time when entering a node, there are no dummy transitions left after the reduction in a node is finished and every potential auto-conflict has become visible.

A decomposition tree is a special case of a *preset tree* [KK01]. Finding an optimal preset tree is NP-complete, but in [KK01] a heuristic bottom-up algorithm is described which performs reasonably well and which works roughly as in the example above. We use this algorithm for the automatic calculation of decomposition trees.

4 Results

In this section the results of some benchmark examples circulating in the STG community can be found. They were made with the tool DesiJ, which can work in a commandline mode and also provides a graphical user interface for interactive decomposition and STG editing. The main purpose for its development was to provide an easy-to-use decomposition tool and an easy-to-extend STG/decomposition framework, the latter guaranteed by a strictly object-oriented design. DesiJ and a collection of benchmark examples can be downloaded from http://www.informatik.uni-augsburg.de/lehrstuehle/swt/ti/ mitarbeiter/mark/projekte/desij.

Each version of the decomposition algorithm was tested with the conservative and the risky auto-conflict detection, the results are listed in Table 1. The runtime is given in seconds, in columns labelled with '$\sum$' the overall number of signals of all final components is printed, in the 'D' columns (only for 'risky') the overall number of signals for which an undetected dynamic auto-conflict exists in the final components can be found.

Of course, version 2 is a special version of version 1, in the latter case the order of transitions contractions results from the order of transitions in the input file. Therefore, the results of version 1 must be handled with care; it is possible (but unlikely) that this 'random' order results in the same or even better results than version 2 (see case 48), or on the other hand there might be an even worse order of contractions. Nevertheless, since version 2 turned out to be rather successful it is used as reduction algorithm by version 3 and 4.

For most STGs the risky conflict detection is not very successful, i.e. there is at least one dynamic auto-conflicts in a final component. Since the runtimes of this approach and the conservative method do not vary much, the risky method seems to be inappropriate. The best time and the smallest number of signals are printed in bold face for the conservative method.

In 42 of 67 cases version 4 performs best, i.e. finishes with the smallest components while using the smallest runtime. If version 4 does not finish with the smallest components, in 9 cases version 2 does, in 8 cases version 3 does and in 1 case version 1 does, but the result of version 2 is never much worse than the best result.

Considering runtime and quality separate, in 50 of 67 cases version 4 returns components which were minimal in the number of signals, and in 60 of 67 cases version 4 has the smallest runtime, although in only two cases the difference is significant.

Only for STG 48 the basic decomposition algorithm is better than version 2. In cases 59-67 (which are very small STGs) it is some 1/1000 sec faster, probably because of the overhead of sorting the transitions first, but normally version 2 is several times faster. (Up to 4400% for case 22.)

Comparing version 2 with version 3, in 18 cases version 2 gives the best results and in 25 cases version 3 does. The runtime seems to depend on the structure of the STG for one group version 2 is faster and for another group version 3 is.

Summing it up, the clear winner is version 4 (tree decomposition) followed by and version 3 (lazy backtracking) and version 2 (ordering transition contractions). Furthermore, in three examples only version 4 was able to finish decomposition, the other algorithms terminated abnormally due to lack of memory[4]. The risky conflict detection turned out be useless in most cases while not saving much time.

	Version 1					Version 2					Version 3					Version 4				
	Conservative		Risky			Conservative		Risky			Conservative		Risky			Conservative		Risky		
Nr.	time	$\sum$	time	$\sum$	D	time	$\sum$	time	$\sum$	D	time	$\sum$	time	$\sum$	D	time	$\sum$	time	$\sum$	D
1	1.071	44	0.948	42	2	0.572	**34**	0.501	32	2	0.755	42	0.723	40	2	**0.28**	**34**	0.275	32	2
2	1.966	54	1.601	52	2	0.549	**32**	0.488	30	2	0.903	54	0.865	52	2	**0.292**	**32**	0.287	30	2
3	3.579	**52**	3.148	50	2	3.067	**52**	2.644	50	2	4.996	80	4.703	78	2	**0.696**	**52**	0.665	50	2
4	13.39	96	12.649	94	2	2.838	**50**	2.335	48	2	5.545	96	5.377	94	2	**0.849**	64	0.973	62	2
5	15.364	**70**	13.958	68	2	13.902	**70**	12.778	68	2	18.484	128	18.42	126	2	**2.424**	90	2.408	88	2
6	71.515	138	66.113	136	2	13.249	**68**	11.933	66	2	21.815	138	22.642	136	2	**2.833**	94	2.754	92	2
7	168.292	133	163.978	131	2	47.709	**88**	43.209	86	2	81.215	156	79.781	154	2	**5.754**	94	5.511	92	2
8	302.317	180	304.549	178	2	44.678	**86**	40.763	84	2	114.545	180	111.145	178	2	**9.32**	104	9.229	102	2
9	0.285	**19**	0.276	19	0	0.266	**19**	0.262	19	0	0.324	**19**	0.32	19	0	**0.204**	**19**	0.2	19	0
10	0.264	**19**	0.263	19	0	0.246	**19**	0.241	19	0	0.292	**19**	0.285	19	0	**0.2**	**19**	0.193	19	0
11	1.997	**37**	1.932	37	0	1.923	**37**	1.788	37	0	2.532	**37**	2.46	37	0	**0.553**	**37**	0.531	37	0
12	1.694	**37**	1.629	37	0	1.43	**37**	1.378	37	0	2.135	**37**	2.149	37	0	**0.481**	**37**	0.475	37	0
13	13.709	**55**	13.791	55	0	9.171	**55**	9.246	55	0	14.86	**55**	14.828	55	0	**1.968**	**55**	1.87	55	0
14	11.412	**55**	10.904	55	0	6.316	**55**	6.198	55	0	12.591	**55**	12.613	55	0	**1.427**	**55**	1.441	55	0
15	4.743	79	3.96	73	3	1.821	**53**	1.369	47	3	2.11	70	2.194	64	3	**0.474**	**53**	0.459	47	3
16	21.843	101	20.627	95	3	1.657	**50**	1.185	44	3	2.919	101	2.783	95	3	**0.48**	**50**	0.458	44	3
17	27.947	109	25.369	103	3	11.029	**80**	9.142	74	3	20.713	160	19.689	154	3	**1.672**	**80**	1.416	74	3
18	260.215	182	257.227	176	3	10.218	**77**	8.656	71	3	18.222	182	18.399	176	3	**1.489**	**77**	1.399	71	3
19	55.508	**107**	46.604	101	3	48.645	**107**	39.758	101	3	65.914	172	66.776	166	3	**4.275**	**107**	4.224	101	3
20	781.511	263	779.179	257	3	44.359	**104**	36.89	98	3	89.668	263	90.173	257	3	**4.169**	**104**	4.119	98	3
21	205.74	**134**	180.301	128	3	175.305	**134**	153.0	128	3	363.08	305	362.77	299	3	**27.648**	172	26.989	166	3
22	7066.425	344	7016.173	338	3	155.739	**131**	134.763	125	3	309.4	344	309.839	338	3	**27.389**	167	27.837	161	3
23	0.785	**28**	0.763	28	0	0.586	**28**	0.563	28	0	1.129	**28**	1.096	28	0	**0.301**	**28**	0.3	28	0
24	0.489	**28**	0.478	28	0	0.442	**28**	0.43	28	0	0.71	**28**	0.695	28	0	**0.269**	**28**	0.269	28	0
25	9.437	**55**	9.551	55	0	7.026	**55**	6.905	55	0	18.996	**55**	18.755	55	0	**1.157**	**55**	1.157	55	0
26	5.856	**55**	5.778	55	0	4.192	**55**	4.126	55	0	9.414	**55**	9.145	55	0	**0.882**	**55**	0.884	55	0
27	82.646	**82**	82.86	82	0	50.081	**82**	50.114	82	0	152.07	**82**	147.176	82	0	**6.779**	**82**	6.591	82	0
28	32.764	**82**	32.979	82	0	25.421	**82**	25.427	82	0	70.359	**82**	69.205	82	0	**10.232**	**82**	10.063	82	0
29	56.114	164	35.746	132	44	48.71	158	26.019	119	34	13.985	149	12.619	113	20	**12.159**	**141**	8.258	122	22
30	57.036	164	37.767	135	41	48.286	157	26.98	121	34	14.623	**154**	12.934	126	27	**11.842**	155	7.514	135	26
31	58.771	165	43.174	136	41	50.288	159	27.745	121	34	**14.707**	**155**	13.094	126	27	24.401	**155**	16.302	135	30
32	30.815	153	24.173	135	19	28.016	145	19.213	122	19	9.286	138	9.256	125	16	**3.259**	**133**	2.869	127	16
33	18.819	121	14.506	108	11	16.536	104	11.347	98	9	5.865	108	5.322	101	15	**2.845**	**100**	2.191	92	9
34	29.211	143	21.848	123	15	24.751	132	19.256	117	12	8.271	143	7.458	128	11	**5.196**	**129**	3.929	117	12
35	39.032	166	30.946	147	16	37.268	160	26.034	135	20	11.456	**142**	10.914	133	19	**5.24**	145	4.084	124	22
36	56.383	164	37.337	135	41	48.535	157	27.208	121	34	14.632	**154**	12.897	126	27	**11.872**	155	7.365	135	26
37	55.51	164	35.433	132	44	48.677	158	26.047	119	34	14.091	149	12.625	113	20	**12.103**	**141**	8.109	122	22
38	31.198	153	23.851	135	19	27.8	145	19.296	122	19	9.373	138	9.667	125	16	**3.153**	**133**	2.839	127	16
39	18.185	121	14.277	108	11	16.845	104	11.466	98	9	5.933	108	5.355	101	15	**3.219**	**100**	2.17	92	9
40	**0.102**	**5**	0.099	5	0	0.105	**5**	0.101	5	0	0.108	**5**	0.106	5	0	0.111	**5**	0.107	5	0
41	33.461	113	29.387	104	9	27.772	106	23.371	96	5	16.89	**101**	7.859	97	9	**6.689**	**101**	6.013	96	5

[4] The algorithm itself does not need much memory, but saving the intermediate STGs does and for inappropriate algorithms these STGs can get very large as described in Section 3.1

	Version 1					Version 2					Version 3					Version 4				
	Conservative		Risky			Conservative		Risky			Conservative		Risky			Conservative		Risky		
Nr.	time	∑	time	∑	D	time	∑	time	∑	D	time	∑	time	∑	D	time	∑	time	∑	D
42	49.231	**93**	39.884	89	67	41.266	**93**	36.217	89	66	**13.841**	**93**	8.004	89	66	27.301	**93**	21.751	89	66
43	20.351	104	18.363	98	6	16.527	**92**	15.175	90	2	11.92	100	5.297	91	8	**5.127**	**92**	4.913	90	2
44	63.778	143	67.477	129	7	42.442	**134**	41.086	131	2	26.692	141	14.849	133	0	**15.546**	138	14.514	132	1
45																**26.046**	**108**	21.393	103	60
46	53.593	136	53.325	123	8	37.982	**129**	35.929	125	4	21.116	**129**	11.458	125	5	**13.881**	130	12.715	123	3
47																**19.603**	**110**	15.434	102	60
48	99.457	**171**	70.541	148	18	77.255	176	61.817	159	5	43.174	178	19.685	148	19	**18.461**	172	17.48	160	2
49	134.895	210	103.237	164	15	101.007	203	78.867	171	6	58.86	**193**	25.456	171	17	**22.597**	195	20.737	182	6
50	149.996	210	122.754	164	15	102.479	203	79.521	171	6	56.514	**186**	22.461	165	17	**22.995**	195	21.129	182	6
51	147.185	210	123.186	164	15	101.057	203	78.974	171	6	56.18	**187**	22.422	166	17	**22.618**	195	20.613	182	6
52	169.8	229	144.339	178	25	136.941	214	111.125	182	6	66.721	209	26.492	177	20	**29.962**	**204**	26.574	192	5
53	178.68	229	144.877	178	25	132.072	214	104.849	182	6	66.748	209	26.602	177	20	**33.974**	**206**	32.649	196	3
54	50.773	136	51.223	124	8	36.776	129	34.856	125	4	21.022	129	11.127	125	5	**10.835**	**128**	10.158	124	4
55																**19.462**	**112**	14.604	99	59
56	134.065	210	103.097	164	15	101.257	203	79.139	171	6	58.783	**193**	25.757	171	17	**22.507**	195	20.805	182	6
57	33.674	113	29.791	104	9	27.528	106	23.16	96	5	16.856	**101**	7.816	97	9	**6.8**	**101**	6.03	96	5
58	48.199	**93**	39.234	89	67	41.095	**93**	36.062	89	66	**13.763**	**93**	7.904	89	66	27.549	**93**	21.748	89	66
59	**0.134**	**13**	0.122	12	1	0.137	**13**	0.119	12	1	0.136	**13**	0.133	12	1	0.14	**13**	0.126	12	1
60	0.435	**26**	0.476	26	0	0.421	**26**	0.411	26	0	0.589	**26**	0.571	26	0	**0.28**	**26**	0.279	26	0
61	0.19	**17**	0.186	17	0	0.347	**17**	0.185	17	0	0.226	**17**	0.221	17	0	**0.18**	**17**	0.175	17	0
62	**0.131**	**8**	0.126	8	0	0.133	**8**	0.127	8	0	**0.131**	**8**	0.129	8	0	0.167	**8**	0.135	8	0
63	0.225	**18**	0.222	18	0	0.218	**18**	0.214	18	0	0.275	**18**	0.268	18	0	**0.195**	**18**	0.191	18	0
64	0.23	**22**	0.226	22	0	0.231	**22**	0.223	22	0	0.318	**22**	0.296	22	0	**0.214**	**22**	0.207	22	0
65	**0.131**	**13**	0.13	13	0	0.132	**13**	0.132	13	0	0.142	**13**	0.139	13	0	0.136	**13**	0.133	13	0
66	0.492	**20**	0.366	20	4	0.384	**20**	0.375	20	4	0.315	**20**	0.308	20	4	**0.3**	**20**	0.301	20	4
67	0.296	19	0.231	17	1	0.299	19	0.233	17	1	0.268	**18**	0.263	17	1	**0.226**	19	0.208	17	1

Table 1: Results for some benchmark examples.

Used STGs: `number: name places/transitions/arcs` 1: 2pp.arb.nch.03.csc, 40/24/84 2: 2pp.arb.nch.03, 38/24/80 3: 2pp.arb.nch.06.csc, 64/36/132 4: 2pp.arb.nch.06, 62/36/128 5: 2pp.arb.nch.09.csc, 88/48/180 6: 2pp.arb.nch.09, 86/48/176 7: 2pp.arb.nch.12.csc, 112/60/228 8: 2pp.arb.nch.12, 110/60/224 9: 2pp_wk.03.csc, 24/14/48 10: 2pp_wk.03, 23/14/46 11: 2pp_wk.06.csc, 48/26/96 12: 2pp_wk.06, 47/26/94 13: 2pp_wk.09.csc, 72/38/144 14: 2pp_wk.09, 71/38/142 15: 3pp.arb.nch.03.csc, 59/36/126 16: 3pp.arb.nch.03, 56/36/120 17: 3pp.arb.nch.06.csc, 95/54/198 18: 3pp.arb.nch.06, 92/54/192 19: 3pp.arb.nch.09.csc, 131/72/270 20: 3pp.arb.nch.09, 128/72/264 21: 3pp.arb.nch.12.csc, 167/90/342 22: 3pp.arb.nch.12, 164/90/336 23: 3pp_wk.03.csc, 36/20/72 24: 3pp_wk.03, 34/20/68 25: 3pp_wk.06.csc, 72/38/144 26: 3pp_wk.06, 70/38/140 27: 3pp_wk.09.csc, 108/56/216 28: 3pp_wk.09, 106/56/212 29: dup-4-phase-data-pull.1, 133/123/286 30: dup-4-phase-data-pull.2, 135/123/290 31: dup-4-phase-data-pull.3, 136/123/292 32: dup-4-phase-data-pull.master.3, 114/105/242 33: dup-4-phase-data-pull.master.4.alt, 109/96/234 34: dup-4-phase-data-pull.master.4, 113/100/242 35: dup-4-phase-data-pull.slave.3, 121/112/258 36: dup-4ph-csc, 135/123/290 37: dup-4ph, 133/123/286 38: dup-4ph-mtr-csc, 114/105/242 39: dup-4ph-mtr, 109/96/234 40: duplicator, 14/12/28 41: dup-master.mod.1, 129/100/296 42: dup-master.mod.1.untog, 116/165/669 43: dup-master.mod.2, 113/88/264 44: dup-master.mod.3.1, 140/100/321 45: dup-master.mod.3.1.untog, 126/134/460 46: dup-master.mod.3.3, 135/98/310 47: dup-master.mod.3.3.untog, 117/128/458 48: dup-master.mod.3.4, 145/107/330 49: dup-master.mod.3.5, 153/115/346 50: dup-master.mod.3.6.1, 153/115/346 51: dup-master.mod.3.6, 153/115/346 52: dup-master.mod.3.7, 159/119/359 53: dup-master.mod.3.8, 159/119/359 54: dup-master.mod.3, 134/98/308 55: dup-master.mod.3.untog, 121/128/456 56: dup-mtr-mod-csc, 153/115/346 57: dup-mtr-mod, 129/100/296 58: dup-mtr-mod-untog, 116/165/669 59: imec-alloc-outbound, 17/18/36 60: imec-master-read, 37/26/74 61: imec-nak-pa, 22/18/44 62: imec-nowick, 19/14/38 63: imec-ram-read-sbuf, 26/20/52 64: imec-sbuf-ram-write, 29/20/58 65: imec-sbuf-read-ctl, 14/12/28 66: tsend-bm, 45/39/94 67: tsend-csm, 34/29/70

5 Conclusion and Future Work

The prototype implementation of the decomposition algorithm of [VW02] was very successful compared to the former all-in-one synthesis approach. Nevertheless, the improved DesiJ implementation demonstrated that there are enough possibilities to improve performance. Especially tree decomposition turned out to be an excellent method for saving time and memory.

As mentioned in Section 3.3, the precalculated decomposition tree is not necessarily optimal for the final components, since signals might be moved from nodes to their children. Future work in this direction will be to adopt the top-down algorithm for building preset trees from [KK01]. This algorithm starts at the root node – as the tree decomposition does – and adds leafs iteratively to the tree. The idea is to interleave this building process with decomposition itself – including the results of a possible backtracking – in order to get a more optimal decomposition tree.

Another starting point for optimisation is to improve the detection of redundant places. Profiling runs showed that DesiJ spends about 60% of its runtime with this task, and improving this more technical part of DesiJ would surely improve the overall performance.

More important, for the time being DesiJ looks only for so called *shortcut places* [SVJ05] which are a subclass of redundant places. Improving this more algorithmical part of DesiJ would reduce backtracking (since undetected redundant places can prevent secure transition contractions) and therefore improving runtime and quality of the components.

References

[CKK+02] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer, 2002.

[Ebe87] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. PhD thesis, Dept. of Math and C.S., Eindhoven University of Technology, 1987.

[Hol82] L. A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, 1982.

[KK01] V. Khomenko, , and M. Koutny. Towards an efficient algorithm for unfolding petri nets. In K.G. Larsen and M. Nielsen, editors, *CONCUR 2001*, Lect. Notes Comp. Sci. 2154, 2001.

[KKY04] V. Khomenko, M. Koutny, and A. Yakovlev. Logic synthesis for asynchronous circuits based on petri net unfoldings and incremental sat. In Canada Kishinevsky M. and Ph. Darondeau, editors, *ACSD 2004*, pages 16–25, 2004.

[SVJ05] M. Schaefer, W. Vogler, and P. Jančar. Determinate STG decomposition of marked graphs. In G. Ciardo and P. Darondeau, editors, *ATPN 05*, Lect. Notes Comp. Sci. 3536, 365–384. Springer, 2005.

[VK05] W. Vogler and B. Kangsah. Improved decomposition of signal transition graphs. In *ACSD 2005*, pages 244–253, 2005.

[VW02] W. Vogler and R. Wollowski. Decomposition in asynchronous circuit design. In J. Cortadella et al., editors, *Concurrency and Hardware Design*, Lect. Notes Comp. Sci. 2549, 152 – 190. Springer, 2002.