Strategies for Optimised STG Decomposition

Mark Schaefer[1] Walter Vogler[1] Ralf Wollowski[2] Victor Khomenko[3]
[1]Fakultät für Informatik, University of Augsburg, Germany
E-mail: {schaefer,vogler}@informatik.uni-augsburg.de

Hasso-Plattner-Institut (HPI) für Softwaresystemtechnik GmbH,
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany.
E-mail: ralf.wollowski@hpi.uni-potsdam.de

School of Computing Science, University of Newcastle upon Tyne, UK.
E-mail: Victor.Khomenko@ncl.ac.uk

***Abstract*— When synthesising an asynchronous circuit from an STG, one often encounters the state explosion problem. In order to alleviate this problem one can decompose the STG into smaller components.**

This paper deals with the decomposition method of [11], [12] and introduces several strategies for efficient implementations, proves them correct and compares them by means of benchmark examples.

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

I. 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* (*STGs*), which are interpreted Petri nets.

While STGs are relatively simple and well-studied, the issue of computational complexity for highly concurrent STGs is quite serious due to the state space explosion problem. This puts practical bounds on the size of control circuits that can be synthesised using such techniques, which are often restrictive, especially if the STG models are not constructed manually by a designer but rather generated automatically from high-level hardware descriptions.

One way to alleviate state explosion is to decompose an STG into several smaller ones which behave together in the same way as the original one. The advantages are a faster synthesis and a reduced peak memory usage. In this paper, we deal with the decomposition method of [11], [12], which is non-deterministic, leaving a lot of choices for implementation. We introduce four strategies to improve its efficiency and the quality of the components.

The next two sections give a condensed overview of STGs and decomposition. The fourth section introduces the new decomposition strategies, which is followed by the results of their application to some benchmark examples. After this, possible applications of the new strategies to the STG decomposition methods of Carmona and Cortadella [1], [2] and Yoneda, Onda and Myers [13] are discussed. The paper ends with a conclusion and an outlook for future work.

For more information about asynchronous circuits, STGs and decomposition see [4], [11], [12].

II. Basic Definitions

This section provides the basic notions for Petri nets and STGs, for a more detailed explanation cf. e.g., [4].

A *Petri net* is a 4-tuple $N = (P, T, W, M_N)$ where P is a finite set of *places* and T a finite set of *transitions* with $P \cap T = \emptyset$. $W : P \times T \cup T \times P \to \mathbb{N}_0$ is the *weight function* and M_N the *initial marking*, where a *marking* is a function $P \to \mathbb{N}_0$ which assigns a number of *tokens* to each place. A Petri net can be considered as a bipartite graph with weighted and directed edges between places and transitions.

The *preset* of a place or transition x is denoted as ${}^\bullet x$ and defined by ${}^\bullet x = \{y \in P \cup T \mid W(y,x) > 0\}$, the *postset* of x is denoted as $x^\bullet$ and defined by $x^\bullet = \{y \in P \cup T \mid W(x,y) > 0\}$. These notions are extended to sets as usual. We say that there is an *arc* from each $y \in {}^\bullet x$ to x.

A transition t is *enabled under a marking M* if $\forall p \in {}^\bullet t : M(p) \geq W(p,t)$, which is denoted by $M[t\rangle$. An enabled transition can *fire* or *occur* yielding a new marking M', written as $M[t\rangle M'$, if $M[t\rangle$ and $M'(p) = M(p) - W(p,t) + W(t,p)$ for all $p \in P$.

A transition sequence $v = t_0 t_1 \ldots t_n$ is *enabled under a marking* M (yielding M') if $M[t_0\rangle M_0[t_1\rangle M_1 \ldots M_{n-1}[t_n\rangle M_n = M'$, and we write $M[v\rangle$, $M[v\rangle M'$ resp.; v is called *firing sequence* if $M_N[v\rangle$. The empty transition sequence λ is enabled under every marking.

M is called *reachable* if a transition sequence v with $M_N[v\rangle M$ exists. We only consider Petri nets N such that the set $[M_N\rangle$ of reachable markings is finite (i.e. N is *bounded*).

An *STG* is a tuple $N = (P, T, W, M_N, In, Out, l)$ where (P, T, W, M_N) is a Petri net and In and Out are disjoint sets of *input* and *output signals*. For $Sig := In \cup Out$ being the set of all signals, $l : T \to Sig \times \{+,-\} \cup \{\lambda\}$ is the *labelling* function. $Sig \times \{+,-\}$ or short $Sig\pm$ is the set of *signal edges* or *signal transitions*; its elements are denoted as $a+$, $a-$ resp. instead of $(a,+)$, $(a,-)$ resp. A plus sign denotes that a signal value changes from *logical low* (written as 0) to *logical high* (written as 1), and a minus sign denotes the other direction. We write $a\pm$ if it is not important or unknown which direction takes place; if such a term appears more than once in the same context, it always denotes the same direction. To keep the notation short, input/output signal edges are just called input/output edges.

An STG may initially contain transitions labelled with λ called *dummy transitions*. They are a design simplification and describe no physical reality, and they have to be removed before our decomposition algorithm can be applied. However, the algorithm itself labels certain transitions with λ at intermediate stages; since their nature is different from that of dummy transitions, they are called *divining transitions*[1]. This relabelling of a transition is called *lambdarising* a transition. The set of transitions labelled with a certain signal is frequently 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.

In a graphical representation, places are drawn as circles containing a number of tokens corresponding to their marking. Transitions are drawn as rectangles together with their labelling, the weight function as directed arcs xy (labelled with $W(x,y)$ if $W(x,y) > 1$).

An STG can be taken as a specification formalism for *asynchronous circuits*. Such a circuit has input signals, which are controlled by the environment, and output signals, whose values are changed by the circuit. The STG describes which output signals should be performed; at the same time, it describes assumptions about the environment, which should perform input signals only if they are allowed by the STG.

We lift the notion of enabledness to transition labels: we write $M[l(t)\rangle\rangle M'$ if $M[t\rangle M'$. This is extended to sequences as usual; note that λ-labels are deleted since λ is the empty word. An STG has *auto-concurrency* if there are transitions t_1 and t_2 with $l(t_1) = l(t_2) \neq \lambda$ such that for some reachable marking M $\forall p \in P : M(p) \geq W(p,t_1) + W(p,t_2)$.

A sequence $v \in (Sig\pm)^*$ is called a *trace of a marking* M if $M[v\rangle\rangle$, and a *trace* of N if $M = M_N$. The *language of* N is the set of all traces of N denoted by $L(N)$.

The *reachability graph* RG_N of an STG N is an edge-labelled directed graph on the reachable markings with M_N as root; there is an edge from M to M' labelled $l(t)$ whenever $M[t\rangle M'$. RG_N can be seen as a finite automaton (where all states are final), and $L(N)$ is the language of this automaton. N is *deterministic* if its reachability graph is a deterministic automaton, i.e. if it contains no λ-labelled transitions and if for each reachable marking M and each signal edge $s\pm$ there is at most one M' with $M[s\pm\rangle\rangle M'$.

An important property of STGs is *consistency*[2]. An STG is consistent if it satisfies for every signal s the following two conditions: (i) in all traces, the first occurrence of s has the same sign (either rising of falling); (ii) the rising and falling edges of s alternate in every trace. From an *inconsistent* STG, one cannot synthesise a circuit and in this paper we assume that all STGs we want to decompose are consistent; in particular, consistency is needed for the correctness proof of the new decomposition strategy LAZYBACK. Note also that a consistent STG cannot have auto-concurrency, since this would imply that e.g., $s+$ can fire twice under some reachable marking.

[1]Motivated by their treatment in the correctness proof where some sort of angelic bisimulation plays the role of an invariant [12].

[2]This is a simplified notion of consistency, see [9] for a more elaborated one.

III. STG DECOMPOSITION

Synthesis with STG decomposition works roughly as follows: a partition of the output signals of the given specification N is chosen, and the decomposition algorithm decomposes N into component STGs, one for each set in this partition. Then, a circuit is synthesised from each component, and the interconnection of these circuits has a behaviour that conforms to the specification.

This correctness is formally proven in [11], [12] on the level of STGs. Interconnection on the physical level simply means to connect the circuits with a wire for each common signal, i.e. if an output x of a component C_1 is also an input of a component C_2. On the STG level, interconnection corresponds to the ordinary parallel composition for Petri nets, which fuses transitions with the same label.

The formal correctness notion used in [11], [12] is a form of bisimulation between the specification and this parallel composition. There are two main differences: first, *computation interference* [5] between the components is forbidden, i.e. in the example above, C_1 can only produce $x\pm$ when C_2 is ready to receive it. Secondly, the components can allow input edges which are not allowed by the specification N. This is intuitively correct, since N also describes assumptions on the environment, i.e. a proper environment will not produce these additional input edges. An extension of this correctness notion to internal signals is given in [7].

To describe the decomposition algorithm in more detail, we discuss in the following subsection the notion of *auto-conflicts*, which plays an important part during decomposition. The second subsection deals with the decomposition algorithm itself.

A. Auto-Conflicts

STGs can model more behaviour than a real-life circuit can show. For example, inconsistent STGs cannot be implemented although they are allowed in principle. Another problem are *dynamic conflicts*, i.e. two transitions of an STG enabled under some reachable marking, where firing one would disable the other.

If the conflicting transitions correspond to different input signals then they model a choice made by the environment, and this is not a problem. However, if at least one of the signals is an output, then the specification cannot be implemented as an asynchronous digital circuit. There are three problematic cases:

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. Nevertheless, our decomposition method and our tool DESIJ cover such conflicts, but we will not discuss them here any further.
2. Both transitions are labelled with different output edges. A circuit which can handle such conflicts is called an *arbiter* and cannot be implemented as a purely digital circuit. STGs with such conflicts can also be handled by our decomposition method, which does not introduce

new conflicts of this kind. For a detailed discussion see [11], [12].

3. Both transitions are labelled with the same signal edge, a so-called *auto-conflict*. Such a non-deterministic choice can hardly be handled by circuits, and we assume that decomposition is only applied to STGs without auto-conflicts. During our decomposition algorithm, auto-conflicts could be generated; this is considered as an indication that too many signals were lambdarised in an STG. In this case *backtracking* is performed and a signal is delambdarised as described below.
   Note that conflicts between λ-labelled transitions are ignored.

Auto-conflicts are dynamic in nature, i.e., to detect them one has to generate the reachability graph, which we want to avoid. A much simpler notion is that of a *structural auto-conflict*, where two equally labelled transitions have a common place in their presets. This is a necessary precondition for auto-conflicts and can be checked structurally. Consequently, the decomposition algorithm of [12] checks only for structural conflicts, conservatively treating them as dynamic ones.

The improved decomposition algorithm of [11] makes it possible to ignore structural auto-conflicts to some degree in order to avoid unnecessary backtracking. This results in three different strategies for handling structural auto-conflicts:

1. *Conservative strategy*: As in [12] every structural conflict is considered as a dynamic one.
2. *Risky strategy*: Structural conflicts are completely ignored.
3. *Interactive strategy*: Ask a human if a structural conflict is dynamic or not.

Despite its name, the risky approach is feasible since the decomposition algorithm essentially preserves (dynamic) auto-conflicts. Thus, accidentally generated ones will be detected by the synthesis tool and at least no erroneous circuit will be generated; also see below.

We have implemented the conservative and the risky strategy, and it turned out that the risky strategy in its pure form does not look practical: for most decompositions there is at least some final component with an auto-conflict, and the runtimes are not much smaller than for the conservative conflict detection. Therefore, we restrict ourselves to the conservative strategy from [12] in the following. Still, for the correctness proof of LAZYBACK, it is important to know that the decomposition algorithm gives also a correct result if structural-but-not-dynamic auto-conflicts are ignored.

B. The Decomposition Algorithm

In the following, we assume that we are given a deterministic, consistent specification N without structural auto-conflicts; first, one chooses a *feasible partition*, i.e. a family $(In_i, Out_i)_{i\in I}$ for some set I such that the sets Out_i are a partition of Out, $In_i \subseteq Sig \setminus Out_i$ for each i and furthermore:

- If two output signals x_1, x_2 are in structural conflict in N, then they have to be in the same Out_i.
- If there are $t, t' \in T$ with $t' \in (t^\bullet)^\bullet$ (t is called *syntactical trigger of* t'), then $l(t') \in Out_i$ implies $l(t) \in In_i \cup Out_i$.

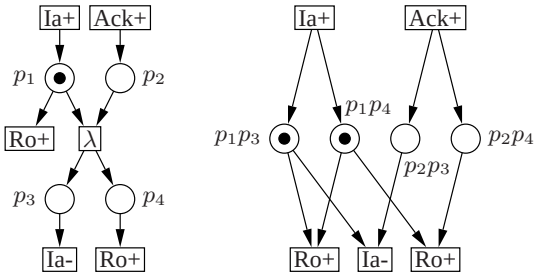

Fig. 1. Transition contraction with generation of structural auto-conflict. **(a)** Initial net. **(b)** After contraction of the λ-labelled transition.

Clearly, for each STG N there is a *minimal feasible partition* Υ_N such that the Out_i are minimal and only necessary inputs are included in In_i.

If we have a feasible partition, we can build another feasible one by adding additional input signals to one of the members or by merging two members (In_1, Out_1) and (In_2, Out_2) to a new one $((In_1 \cup In_2) \setminus (Out_1 \cup Out_2), Out_1 \cup Out_2)$.

All possible partitions can be generated by applying these operations repeatedly to Υ_N.

For each member (In_i, Out_i) of a partition an *initial component* is generated from N: in a copy of the original STG N, every signal not in $In_i \cup Out_i$ is lambdarised and the signals in In_i are considered as inputs of this component — even if they are outputs of N.

The following operations are applied to each of these components; this process is called *reduction*:

- *secure contraction* of divining transitions
- deletion of redundant places
- deletion of redundant transitions
- *backtracking*

We call the first three of these operations *reduction operations*. The reduction of an initial component leads to a component-STG without λ-transitions. Each component-STG is then synthesised, usually by constructing its reachability graph. Very often, adding up the sizes of these graphs gives a number much smaller than the size of the reachability graph of N, in which case the decomposition can be seen as successful. Actually, it might already be beneficial if each reachability graph is smaller than the one of N, in particular for reducing peak memory.

We will now describe the above operations in more detail. The *contraction of a transition* t generates a set of new places $\{(p, q) \mid p \in {}^\bullet t, q \in t^\bullet\}$ (each one of them inherits the tokens and arcs of its 'inner' places) and removes t, ${}^\bullet t$ and $t^\bullet$ from the net; cf. Figure 1.

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 $|{}^\bullet t| \cdot |t^\bullet| - (|{}^\bullet t| + |t^\bullet|)$.

Redundant places are a subclass of *implicit places* which can be deleted without changing the firing sequences of the STG. The difference is that looking for implicit places requires

the reachability graph while redundant places can be detected structurally; hence, we only look for the latter ones during decomposition.

There are two kinds of *redundant transitions*. First, if there are two λ-labelled transitions which are connected to every place in the same way, one of them can be deleted without changing the traces of the STG. Second, a λ-labelled transition t with $W(t,p) = W(p,t)$ for every place p can also be deleted, since its firing does not change the marking and is not visible on the level of traces; observe, that this is valid for any marking of the adjacent places.

These two operations may seem trivial, but especially the deletion of redundant places is essential for getting small components, since very often the existence of such places prevents further transition contractions. The same is also true to some extent for redundant transitions.

Backtracking means to choose a signal that was lambdarised in the initial component, to add it to the input signals, and to *delambdarise* it, i.e. to restore the original labels of the corresponding transitions; then reduction is started anew. If there are still divining transitions left but none of the reduction operations can be performed, then an arbitrary one of these transitions is chosen and backtracking is performed to its former signal.

In particular, if the contraction of a divining transition t would generate a new structural auto-conflict, this is considered as an indication that too many signals of a component were lambdarised to produce its output signals appropriately; this can be changed by delambdarising the former signal of t and — informally speaking — providing more information to the circuit.

After the last backtracking, when enough signals are added to the initial component, only the reduction operations have to be applied to get the final component. This means that backtracking is only needed to detect these additional signals; if they are known in advance, one can perform decomposition completely without backtracking. This point of view plays an important part in the correctness considerations for the new decomposition strategies.

All three reduction operations (even if secure contraction is applied in cases where structural auto-conflicts arise) preserve the language [12] and therefore also consistency, and the latter is obviously also true for delambdarising signals. Hence, all intermediate and final results of reduction are consistent. Furthermore, the operations are *auto-cc-preserving* [11], i.e. applied to some STG N with auto-concurrency or (dynamic) auto-conflicts, the resulting STG N' has also auto-concurrency or auto-conflicts. This is the precise argument why the risky approach is feasible. It also implies the following result, which we will use in Section IV-B:

(*) Assume the reduction operations are applied to some initial component, *ignoring structural auto-conflicts*, and this gives an STG N' without structural auto-conflicts. Then, no structural auto-conflict encountered during this partial reduction is dynamic, and hence — as shown in [11] — the reduction is correct so far.

To prove this, assume that some intermediate $\hat{N}'$ has an auto-conflict. Then, since the operations are auto-cc-preserving, N' has to contain an auto-conflict or auto-concurrency. The former is impossible since N' has no structural auto-conflicts; the latter implies that N' is not consistent, which is also impossible.

The decomposition algorithm itself is non-deterministic, i.e. the operations can be applied in any order; the results have been proven to be always correct. In the original version of the algorithm called BASIC here, the order of contractions depends on the ordering of items in the input file, which can be considered as random.

However, for some examples the order of operations is crucial for the final result in terms of the number of added signals, the overall number of reachable markings and the time needed for synthesis. 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 on this component 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. Finally, it is possible to choose between several possible partitions of the output signals but it is unclear how a good one can be found.

Answers to these questions are given in the next section.

IV. Optimised Decomposition Strategies

In this section we will introduce four new strategies to improve upon the basic decomposition algorithm. In the subsection headings, the name of the corresponding strategy is given in capitalised letters; this name is used as an identifier of the underlying concept as well as for the concrete implementation.

A. Ordering Transition Contractions (REORDERING)

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 divining 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 in its preset and 6 places in its postset would increase the number of places by 14. These 14 places may be adjacent to other divining 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 REORDERING, the divining transitions are sorted by the number of additional places their contraction would generate in the initial component. Then reduction works as in BASIC, following this pre-calculated list of transition contractions. In order to avoid repeated calculation and sorting after every reduction operation, this list is not updated during reduction.

As mentioned above, the original decomposition algorithm is completely non-deterministic and its correctness has been

proven under this assumption. This is an advantage for proving the newly introduced methods correct: one simply has to show that any final component generated by a new method could have been generated by the basic algorithm for some order of operations.

It is perfectly clear that REORDERING is correct in this sense, because the chosen order of contractions is just a concrete instance of an arbitrary one.

*B. Lazy Backtracking (*LAZYBACK*)*

In the original implementation, backtracking was performed by discarding all the operations performed so far and restarting the reduction for an initial component with an enlarged input set. This method plays an important part in the proof of correctness in [11], [12]. But it can obviously be rather inefficient, e.g., in extreme cases backtracking might occur for the last divining transition and result in repeating a large number of operations.

Naturally, if the reduction should not start anew from 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 now to start from the very beginning. Instead, it is possible to use the last suitable savepoint. While this basic idea is simple, there is a complication to consider; to confine the complication, we do not delete redundant transitions in LAZYBACK.

The algorithm works as follows. Starting from N, all initially useless signals are lambdarised yielding the initial component N_0. Instead of contracting them in an arbitrary order as in BASIC, the divining transitions are contracted grouped by their former signals as described above and depicted in Figure 2.

If contracting all a_0-transitions is possible, i.e. all contractions are secure and no new structural auto-conflict is 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 savepoints and a sequence (a_i) of contracted signals. If every contraction is possible, LAZYBACK is obviously correct.

Assume now that backtracking has to be performed since the contraction of signal a_j is not possible in N_j. In BASIC, one would delambdarise a_j in N_0 and start anew from there. Instead, we delambdarise a_j in N_j resulting in N'_j; the critical point is that we have to check for a structural auto-conflict of a_j in N'_j now. (Such a conflict might exist, because conflicts between divining transitions are ignored during reduction.)

First, we study the case where no such conflict exists: in this case, delambdarise a_j also in all preceding savepoints and proceed from N'_j with a new signal a'_j to be contracted. This is correct due to the following more general claim which is also used later on.

(**) Let N'_j be obtained from an intermediate savepoint N_j by delambdarising some set of signals A not containing $a_0, \ldots, a_{j-1}$. If N'_j does not have any structural auto-conflicts, then it can be constructed during a correct reduction for the initial component N'_0 obtained from N_0 by delambdarising A.

We will argue inductively that actually the same operation sequence which reached N_j can be performed to reach N'_j, at least if we ignore structural auto-conflicts for the time being; during this, every original intermediate STG $\hat{N}$ is matched with some new intermediate STG $\hat{N}'$ obtained from $\hat{N}$ by delambdarising A. We clearly have this match before the first operation for N_0 and N'_0. Assume we have reached some $\hat{N}'$ matching $\hat{N}$ in the original sequence.

If the operation applied to $\hat{N}$ is a redundant place deletion, then this can also be applied to $\hat{N}'$ with a matching result, since place redundancy does not depend on the labelling. If the operation is the contraction of transition t, we note that the former signal of t is in $\{a_0, \ldots, a_{j-1}\}$ and that thus t is also a λ-transition in $\hat{N}'$. Furthermore, the contraction is still secure since this is independent of the labelling. Hence, the contraction can be applied with a matching result, which finishes the inductive proof.

To conclude the proof of (**), we simply point out that the operation sequence reaching N'_j is correct according to (*) in the discussion of the operations in Section III-B.

This argument shows that LAZYBACK can additionally help to keep the components small: since we assume that only N'_j is free of structural auto-conflicts but allow them for the intermediate results (which are not constructed actually), it is possible to avoid some unnecessary backtracking due to structural-but-not-dynamic auto-conflicts. This is sound, since real dynamic auto-conflicts of the intermediate results would appear also in N'_j.

Second, we look at the case where there is at least one structural auto-conflict for signal a_j in N'_j. Then we cannot proceed from this savepoint; instead, we have to find the signals whose contraction caused these conflicts. To do this, consider STG N_{j-1} with a_j delambdarised resulting in N'_{j-1}. If there is no conflict for a_j, it is clear that the conflicts were generated by the contraction of a_{j-1}. For example, look at Figure 1: in (a) there is no structural auto-conflict for the signal Ro, but the contraction of the λ-labelled transition introduced one in (b).

If some conflicts still exist in N'_{j-1}, go back to savepoint N_{j-2}, delambdarise a_j again and check for a conflict for a_j, and so on. Observe that the signals $a_{j-1}, a_{j-2}, \ldots$ are not delambdarised while going back in this way, they are contracted again if the reduction is eventually resumed.[3]

If eventually a savepoint N_k is reached, where the respective N'_k does not have a structural auto-conflict for a_j, it is clear that the contraction of signal a_k caused at least some conflict of a_j, which is visible in N'_{k+1}. Therefore, a_k has to be delambdarised in N'_k, too, resulting in N''_k.

At this point there are two possible sub-strategies:

- LAZYSINGLE: If there is no structural auto-conflict for a_k in N''_k, lambdarise a_j in N''_k again and proceed from there with reduction. If there is a structural auto-conflict for a_k in N''_k, go back to savepoint N_{k-1}, delambdarise

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

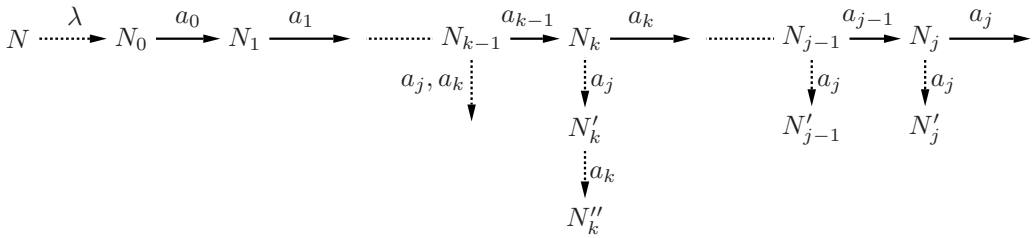

Fig. 2. Backtracking of LAZYMULTI. A dotted arc denotes that the corresponding signals are delambdarised. The λ-labelled one denotes the construction of the initial component. A normal arc denotes the contraction of the corresponding signals.

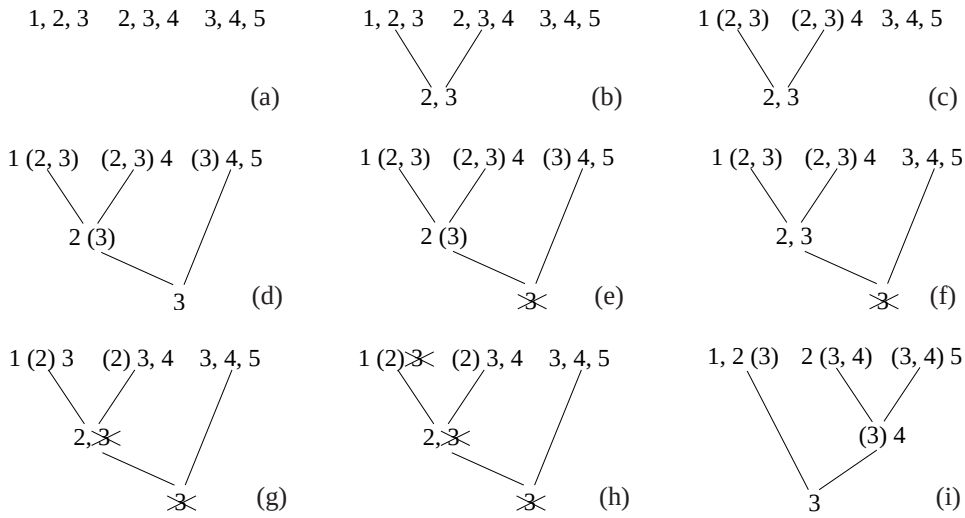

Fig. 3. Building of a simple decomposition tree. Leafs from the left: components C_1, C_2, C_3. (a) initial situation (b) two components merged (c) already contracted signals embraced (d) final tree with all components (e)-(g) contraction of signal 3 not possible in root node and therefore postponed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

a_k but not a_j and check for structural auto-conflicts, and so on. In general, go back with the *last delambdarised signal* until a suitable savepoint is found.

- LAZYMULTI: If there is no structural auto-conflict for a_k in N''_k, proceed from there; differing from LAZYSINGLE, a_j is not lambdarised again. If there is a structural auto-conflict for a_k in N''_k, go back to savepoint N_{k-1}, delambdarise a_k *and* a_j and check for structural auto-conflicts, and so on. In general, go back with *all signals delambdarised so far* until a suitable savepoint is found.

Since N does not have any structural auto-conflicts, we will eventually reach a savepoint which has the same property when we delambdarise the signals in the respective set A. We then proceed with reduction from this modified savepoint, after having modified all preceding savepoints in the same way.

Both strategies are correct due to (**). LAZYSINGLE is the more optimistic strategy: the hope is that preventing the 'initial' structural auto-conflict by making *one* signal visible might also prevent the resulting conflicts including the one for a_j. Backtracking in LAZYMULTI mimics BASIC, which would restart with a_j delambdarised after a structural auto-conflict was encountered, then restart with also a_k delambdarised etc.

Our implementation of LAZYBACK differs slightly from the above description: if during backtracking a savepoint is found from which reduction can proceed, the corresponding signals are not delambdarised in the preceding savepoints in order to improve runtime. It is in fact not hard to see that this is also correct due to (**).

C. Tree Decomposition (TREE)

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

Considering example decompositions, it turned out that in most cases some components had many lambdarised signals in common. Therefore there should be an intermediate STG C', from which these 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 means of an example (see Figure 3): 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 $\{1, 2, 3\}$, $\{2, 3, 4\}$, $\{3, 4, 5\}$ be the signals which are lambdarised initially in these components. A possible intermediate STG C' for C_1 and C_2 would be the STG in which signals 2 and 3 have been contracted.

In (a) the initial situation is depicted. There are three independent leaves 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 (c) one can see nearly the same situation as in (b), but signals which were already contracted earlier are commented out; they are embraced, and the labels of the leaves become $\{1\}$ and $\{4\}$. The following is a more operational view: each node u is labelled with the signals $s(u)$ which should be contracted

when it is entered with some STG, see below. In (d) we added a common intermediate result for C' and C_3 with the label $\{3\}$, yielding the final decomposition tree. In (i) there is another possible tree for the same components.

Tree decomposition according to a given decomposition tree works as follows: enter the root node with the initial STG N without lambdarised signals. Whenever entering a node u with an STG N_u, lambdarise the signals $s(u)$ in N_u, 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.

From this use of a decomposition tree, it is clear that in an *optimal* decomposition tree the sum of all $|s(u)|$ should be minimal. Because of this, a decomposition tree is the same as a *preset tree* in [6]. There it is shown that finding an optimal preset tree is NP-complete and a heuristic bottom-up algorithm is described which performs reasonably well and works roughly as in the example above. We use this algorithm for the automatic computation of decomposition trees.

However, there is a twist in our setting: since this tree is pre-calculated from the initial components, it is very likely that not all signal contractions are possible. If a signal $a \in s(u)$ cannot be contracted in N_u, the easiest thing would be to make a visible for the whole subtree of u. But there is a way to obtain better results: we *postpone* a, i.e. we add a to every child node of u (if there are any). This is promising for the following reason: the contraction of a may have caused a structural auto-conflict for a signal a', which is lambdarised deeper in this subtree. When a' is eventually contracted, the contraction of a may become possible, making at least some of the final components smaller.

In our example, assume that the contraction of signal 3 in the root node is not possible, because it 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, there can be no conflict for signal 4 since it is lambdarised; hence, the contraction of signal 3 might become possible after the contraction of signal 4 – but not in the inner node, and so 3 is added to the left and middle leafs (g). In the first one the contraction is again not possible, but in the latter one it is, and (h) shows the final situation. Therefore, the components C_2 and C_3 were generated as prearranged, only component C_1 has the additional signal 3, considered as an additional input.

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 divining transitions left after the reduction in a node is finished and every potential auto-conflict has become visible.

Backtracking, with or without postponing signals, changes the pre-calculated decomposition tree, possibly decreasing its quality. In future work we will study how to approximate an optimal decomposition tree in such a way that postponing is taken into account, refer Section VII.

We now argue why TREE is a correct strategy. Consider the final decomposition tree resulting from postponing some signals, cf. Figure 3(h); consider a final component C (possibly having additional input signals due to postponing) and the path from the root to the corresponding leaf. The reduction operations on this path can be performed in the same order without backtracking by BASIC, cf. the discussion of backtracking in Section III. The only difference is that in TREE signals are lambdarised 'just in time' and not at the beginning. This could only result in more structural auto-conflicts, but TREE does not encounter any. This implies correctness, which follows also from the correctness of *hierarchical decomposition* introduced in [7].

D. Component Aggregation (AGGREGATION)

An open problem of decomposition is how to find a good partition of the output signals of an STG N. A natural partition is of course the finest partition Υ_N, whose members usually contain only one output signal. As already mentioned, all other feasible partitions can be found by merging members of Υ_N.

Using TREE, such a coarser partition can be found by *aggregating nodes* of a decomposition tree. We perform TREE as described above for the partition Υ_N but with one difference: after a node u is reduced yielding an STG C, we check if we should stop at this point. If so, instead of generating all the components for the leaves of u in the original decomposition tree, only C is returned, which produces all the output signals of the leaves.

By this method we get a reduced decomposition tree corresponding to a new 'coarser' partition which is also feasible, see the discussion at the beginning of Section III-B. Clearly, the correctness of AGGREGATION follows from the correctness of TREE.

It remains to explain under which conditions a node u should be aggregated. Since the main purpose of decomposition is to reduce the overall number of reachable markings of all components, there are two sensible criteria:

- A node can be aggregated if the STG C has not too many signals, so that synthesis can be performed in a reasonable time. In practice this is the same as not having too many transitions.
- Consider the case that u has a leaf u' which can be reached by contraction of only a small number of additional signals. This means that the component of u' has nearly the same size as that of u and the same might easily be true for the corresponding reachability graphs. Therefore, instead of generating the component C' of u' and some additional components, it might be better to aggregate u and to generate C, which is only slightly larger than C'. Furthermore, the potential advantage of generating C' might not materialise due to backtracking.

For our benchmark examples, we implemented the first criterion with bounds on the signal number ranging from 3 to 15. These values could be tailored to specific synthesis methods in future experiments.

V. RESULTS

This section presents the experimental results for a number of benchmark examples circulating in the STG community. They were obtained with the tool DESIJ, a new

implementation of the decomposition algorithm in Java. It provides a command-line mode as well as 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. DESIJ and a collection of benchmark examples can be downloaded from `www.informatik.uniaugsburg.de/lehrstuehle/swt/ti/research/desij` .

Originally, our main target for the new decomposition strategies was to improve the *runtime* used for decomposition. During our experiments, it turned out that the different strategies also influenced the *size* of a decomposition, i.e. the cumulated number of reachable markings. The latter measure is obviously more important for the subsequent synthesis procedure (not discussed here) and therefore used as the main criterion for comparing different strategies.

Each STG N was decomposed using the partition Υ_N, i.e. usually each component produces one output. Since REORDERING turned out to be rather successful, it is used as the reduction algorithm in the stages of LAZYBACK, TREE and AGGREGATION.

The risky auto-conflict detection was tested with BASIC, REORDERING, LAZYBACK and TREE, but — as mentioned before — turned out to be not very successful in general: for most decompositions, there are some final components with at least one auto-conflict; furthermore, the runtimes are not much smaller than for the conservative conflict detection. Nevertheless, most components are conflict-free and, thus, the risky approach might be useful for semi-automatic decomposition. Here we only present results for the conservative approach. For the benchmarks considered here, there is no difference between LAZYSINGLE and LAZYMULTI and therefore only the results for the former are given.

In Table I the different strategies are compared. On the left, for BASIC, REORDERING, LAZYSINGLE and TREE the runtime (in seconds) and the size of the resulting decomposition are given; the smallest time and smallest size in this group of strategies is annotated with $\circ$. The number of components according to Υ_N is the same for all strategies and can be found in the TREE column.

Furthermore, we applied AGGREGATION with bounds on the signal number ranging from 3 to 15. In the table, we report values for TREE, for bound 3 (column AGGREGATION 3) and bound 15 (column AGGREGATION 15), and, furthermore, for the bounds where the smallest size of the decomposition are obtained (column AGGREGATION BEST). For each of these, the size of the decomposition and the number of components are given; the smallest size in this group is annotated with $\bullet$, except for AGGREGATION BEST. The number of components is only meant to give a clue for the impact of aggregation; naturally, it goes down when the bound on the signal number goes up. Additionally, for AGGREGATION BEST, the bounds are given for which the least size was obtained (or 'all' when there is no difference).

We had hoped that AGGREGATION would reduce the runtimes since it saves some reduction steps. Actually, the runtimes do not differ much from the ones of TREE and are therefore omitted.

Using BASIC as a reference point for the other strategies is somewhat problematic: it is possible (but unlikely) that its 'random 'order results in strictly smaller components than the other strategies. Indeed, for our benchmark examples this happens only once (see case 8); usually, the runtimes for BASIC are quite large (up to 10 minutes) compared to at most 32 seconds for TREE, and they are as expected always longer than the ones of REORDERING. Therefore, we will not consider BASIC in the following discussion.

As expected, in nearly every case TREE has the smallest *runtime*, except for the cases 43, 48, 56 and 59 where LAZYSINGLE is faster . Clearly, TREE has to be preferred unless one wants to compute only a few components. There is no clear pattern when comparing the runtimes of REORDERING and LAZYSINGLE: in some cases the first one runs faster, in some cases the latter. It seems that LAZYBACK eliminates the advantages of REORDERING, since it enforces signal-wise contractions, and that this outweighs the advantage of the improved time-saving backtracking.

In 17 cases all strategies return decompositions of equal *size*. Considering the remaining cases, REORDERING returns the smallest components in 23 cases, in 12 cases it is the only strategy with such a result, LAZYSINGLE does so in 4 (3) cases and TREE in 26 (15) cases.

Considering AGGREGATION, the comparison with respect to the decomposition size has to be taken with a grain of salt: the first 4 strategies compute components for the same partitions of the respective output signals, whereas aggregation changes these partitions, i.e. the respective sizes are sums with fewer summands. Recall that it might already be beneficial, if each reachability graph (i.e. each summand) is smaller than the one of N. Still, taking size as the measure, it turns out that TREE is always worse than AGGREGATION, except for the cases 43, 48, 56 and 59 where all strategies return decompositions of equal size. Compared to the strategies other than TREE, AGGREGATION always returns a decomposition with the smallest size, except for cases 4 and 8.

VI. APPLICATION TO OTHER DECOMPOSITION APPROACHES

In this section, we discuss how the new strategies could be used to improve the decomposition methods of Carmona and Cortadella [1], [2] and Yoneda, Onda and Myers [13]. These methods use the concept of Complete State Coding (CSC), which intuitively means that it should be possible to assign to each reachable marking of the STG a binary vector (encoding) in such a way that no two markings enabling different outputs have the same encoding; see e.g. [3].

Both of these decomposition methods work roughly as follows: starting with an STG which initially has CSC, each final component produces exactly one output and will also have CSC. For this reason, AGGREGATION is not considered in the rest of this section.

In contrast to our decomposition method, in the method of Carmona and Cortadella all relevant signals are determined

		Basic		Reorder.		LazyS.		Tree			Agg. 3		Agg. Best			Agg 15	
Nr.	Name	t	Size	t	Size	t	Size	t	Size	C	Size	C	Size	C	Best	Size	C
1	2pp.arb.nch.03.csc	○0	○131	○0	○131	○0	455	○0	○131	10	123	8	92	3	7	960	1
2	2pp.arb.nch.03	○0	192	○0	○102	○0	530	○0	○102	10	94	8	77	4	5, 6	1088	1
3	2pp.arb.nch.06.csc	4	○179	3	○179	7	847	○0	○179	16	171	14	166	9	4	2048	2
4	2pp.arb.nch.06	7	804	2	○150	7	818	○0	458	16	458	16	317	6	6	8464	2
5	2pp.arb.nch.09.csc	58	4939	17	○227	23	1039	○2	○227	22	219	20	214	12	4	16384	2
6	2pp.arb.nch.09	71	2398	30	616	26	4274	○2	○198	22	190	20	185	12	4	18432	2
7	2pp.arb.nch.12.csc	158	3083	45	○275	73	5933	○6	○275	28	267	26	262	16	4	12160	3
8	2pp.arb.nch.12	151	○3026	143	3054	72	5662	○12	3610	28	3606	27	3204	10	7	53632	3
9	2pp.wk.03.csc	○0	○48	○0	○48	○0	○48	○0	○48	7	●40	5	40	3	3, 4	128	1
10	2pp.wk.03	○0	○52	○0	○52	○0	○52	○0	○52	7	●44	5	44	5	3	160	1
11	2pp.wk.06.csc	1	○96	1	○96	2	○96	○0	○96	13	●88	11	88	6	3, 4	8192	1
12	2pp.wk.06	1	○100	1	○100	2	○100	○0	○100	13	●92	11	92	6	3, 4	10240	1
13	2pp.wk.09.csc	14	○144	11	○144	14	○144	○2	○144	19	●136	17	136	9	3, 4	4608	2
14	2pp.wk.09	11	○148	7	○148	11	○148	○1	○148	19	●140	17	140	17	3	5632	2
15	2pp.wk.12.csc	151	○192	72	○192	112	○192	○32	○192	25	●184	23	184	14	3, 4	36864	2
16	2pp.wk.12	120	○196	37	○196	79	○196	○26	○196	25	●188	23	188	14	3, 4	45056	2
17	3pp.arb.nch.03.csc	5	939	1	○321	2	2093	○0	○321	14	309	11	273	5	7	2248	2
18	3pp.arb.nch.03	13	878	2	344	3	1906	○0	○254	14	250	13	172	5	7, 8	1152	2
19	3pp.arb.nch.06.csc	15	○393	12	○393	24	2025	○1	○393	23	●381	20	381	14	3, 4	18688	2
20	3pp.arb.nch.06	19	○278	14	○278	28	1910	○1	○278	23	●266	20	266	14	3, 4	19456	2
21	3pp.arb.nch.09.csc	187	6161	52	○465	69	2181	○4	○465	32	●453	29	453	19	3, 4	14536	4
22	3pp.arb.nch.09	281	13198	149	6046	71	5006	○4	○350	32	●338	29	338	19	3, 4	14560	4
23	3pp.arb.nch.12.csc	627	7289	156	○537	212	20347	○14	○537	41	●525	38	525	25	3, 4	24776	4
24	3pp.arb.nch.12	632	7142	566	7174	229	13970	○15	○422	41	●410	38	410	25	3, 4	11904	5
25	3pp.wk.03.csc	1	○76	○0	○76	1	○76	○0	○76	10	●64	7	64	4	3 - 5	1024	1
26	3pp.wk.03	○0	○90	○0	○90	○0	○90	○0	○90	10	●78	7	78	7	3	1664	1
27	3pp.wk.06.csc	14	○148	12	○148	21	○148	○1	○148	19	●136	16	136	10	3, 4	10752	2
28	3pp.wk.06	7	○162	5	○162	11	○162	○0	○162	19	●150	16	150	16	3	16128	2
29	3pp.wk.09.csc	66	○220	55	○220	87	○220	○11	○220	28	●208	25	208	15	3, 4	9728	3
30	3pp.wk.09	41	○234	25	○234	53	○234	○5	○234	28	●222	25	222	25	3	15104	3
31	dup.4.phase.data.pull.1	50	878	46	○789	○10	833	○10	860	15	860	15	370	5	14, 15	●370	5
32	dup.4.phase.data.pull.2	54	900	47	○799	12	837	○11	887	15	887	15	384	5	15	●384	5
33	dup.4.phase.data.pull.3	56	904	50	○747	12	824	○11	883	15	883	15	394	5	15	●394	5
34	dup.4.phase.data.pull.master.3	43	787	28	610	8	○550	○5	590	16	590	16	318	6	13 - 15	●318	6
35	dup.4.phase.data.pull.master.4.alt	16	531	15	516	4	527	○3	○471	11	471	11	249	4	14, 15	●249	4
36	dup.4.phase.data.pull.master.4	34	760	25	632	6	627	○4	○510	13	510	13	304	5	14, 15	●304	5
37	dup.4.phase.data.pull.slave.3	47	880	30	○574	9	729	○5	649	16	649	16	297	5	15	●297	5
38	dup.4ph.csc	55	900	48	○799	12	837	○10	887	15	887	15	384	5	15	●384	5
39	dup.4ph	51	878	46	○789	11	833	○10	860	15	860	15	370	5	14, 15	●370	5
40	dup.4ph.mtr.csc	43	787	28	610	8	○550	○5	590	16	590	16	318	6	13 - 15	●318	6
41	dup.4ph.mtr	16	531	15	516	5	527	○3	○471	11	471	11	249	4	14, 15	●249	4
42	dup.master.mod.1	37	962	27	709	13	844	○7	○700	10	700	10	327	3	15	●327	3
43	dup.master.mod.1.untog	46	○1115	40	○1115	○6	○1115	27	○1115	5	●1115	5	1115	5	all	●1115	5
44	dup.master.mod.2	22	724	16	598	7	600	○5	○594	9	594	9	324	3	14, 15	●324	3
45	dup.master.mod.2.untog	17	770	15	755	8	755	○7	○731	6	731	6	580	4	15	●580	4
46	dup.master.mod.3.1	71	1247	43	1166	16	○1150	○15	1151	11	1151	11	566	4	15	●566	4
47	dup.master.mod.3.3	91	1088	38	○840	18	978	○13	880	11	880	11	340	3	15	●340	3
48	dup.master.mod.3.3.untog	34	○968	27	○968	○9	○968	19	○968	5	●968	5	968	5	all	●968	5
49	dup.master.mod.3.4	116	1568	73	1043	41	1202	○21	○1013	14	1013	14	485	6	15	●485	6
50	dup.master.mod.3.5	201	1704	106	○1136	48	1187	○22	1186	15	1186	15	837	9	15	●837	9
51	dup.master.mod.3.6.1	213	1653	106	1126	48	1175	○23	○1094	15	1094	15	691	8	15	●691	8
52	dup.master.mod.3.6	202	1685	105	○1136	48	1187	○22	1186	15	1186	15	838	9	15	●838	9
53	dup.master.mod.3.7	233	1873	152	1657	60	1373	○30	○1286	16	1286	16	895	10	15	●895	10
54	dup.master.mod.3.8	235	1855	152	1639	56	1363	○32	○1306	16	1306	16	898	10	15	●898	10
55	dup.master.mod.3	72	1083	37	○845	16	952	○11	865	11	865	11	420	4	15	●420	4
56	dup.master.mod.3.untog	49	○1004	27	○1004	○7	○1004	20	○1004	5	●1004	5	1004	5	all	●1004	5
57	dup.mtr.mod.csc	201	1704	106	○1136	49	1187	○23	1186	15	1186	15	837	9	15	●837	9
58	dup.mtr.mod	37	962	27	709	13	844	○7	○700	10	700	10	327	3	15	●327	3
59	dup.mtr.mod.untog	46	○1115	40	○1115	○7	○1115	27	○1115	5	●1115	5	1115	5	all	●1115	5

TABLE I

RESULTS OF THE BENCHMARKS.

For BASIC, REORDERING, LAZYSINGLE and TREE the runtime (in seconds) and the size of the resulting decomposition are given; the smallest time and smallest size in this group of strategies are annotated with ○.

For TREE, AGGREGATION 3, AGGREGATION BEST and AGGREGATION 15 the size of the decomposition and the number of components is given; the smallest size in this group is annotated with ●. Additionally, for AGGREGATION BEST, the bounds are given for which the least size was obtained (or 'all' when there is no difference).

before reduction: starting with the syntactical triggers, integer linear programming problems are solved to repeatedly add additional relevant signals until CSC can be guaranteed for this component.

When these signals are determined, all other ones are lambdarised and a restricted subset of our reduction operations is applied. If there are non-contractible dummy transitions, they are removed later in the reachability graph with automata-theoretic methods. As a consequence backtracking is not needed for this method, and therefore LAZYSINGLE and LAZYMULTI cannot be applied.

On the other hand, REORDERING can be used to accelerate the reduction of the final component. Furthermore it might be possible to contract more dummy transitions, which is not as crucial as for our method, but can help to generate a smaller reachability graph.

TREE can also be applied, and since postponing is not needed (as backtracking is not needed), the hopefully optimal pre-calculated decomposition tree will not be changed during reduction. The application of TREE will therefore definitely increase the efficiency of this decomposition method.

In contrast to the previous approach, in the method of Yoneda, Onda and Myers the relevant signals are determined through repeated reduction: for some specification N, they also start with components corresponding to Υ_N and perform reduction similar to our reduction operations. If the resulting component does not have CSC, additional relevant signals are delambdarised in the initial component and reduction is performed again. This is repeated until a component with CSC is generated.

As above, REORDERING can be applied to increase the efficiency of reduction. TREE can be used to accelerate the overall component generation in the following way: calculate the decomposition tree for all initial components (only outputs and their triggers), perform TREE and determine for each component the additional signals. Then use this new information to update the decomposition tree, and so on. If a component has CSC eventually, it does not have to be included in the next iteration, thus making the decomposition tree smaller and smaller when approaching the final result.

VII. CONCLUSION AND FUTURE WORK

The prototype implementation of the decomposition algorithm of [12] 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 and AGGREGATION in combination with REORDERING turned out to be an excellent strategy for saving time and memory.

As mentioned in Section IV-C, the pre-calculated 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 consider the top-down algorithm for building preset trees in [6]. This strategy starts at the root node — as the tree decomposition does — and adds branches iteratively to the tree.

The idea is to interleave this building process with decomposition itself — including postponing — in order to get a better decomposition tree.

Another possibility for optimisation is to improve the detection of redundant places. DESIJ does not look for redundant places after every single transition contraction, but only when none of the remaining divining transitions can be contracted and also before the final components are returned. (In the former case, it is checked again if the transitions can be contracted.) Nevertheless, profiling runs showed that DESIJ spends about 60% of its runtime on this task. 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* [8], [10] which are a subclass of redundant places. Improving this more algorithmic part of DESIJ would reduce backtracking (since undetected redundant places can prevent secure transition contractions) and therefore improve runtime and quality of the components.

Acknowledgements: This research was supported by DFG-projects 'STG-Dekomposition' Vo615/7-1 and Wo814/1-1, and the Royal Academy of Engineering/EPSRC grant EP/C53400X/1 (DAVAC).

REFERENCES

[1] J. Carmona and J. Cortadella, "ILP models for the synthesis of asynchronous control circuits," in *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, 2003, pp. 818–825.

[2] J. Carmona, "Structural methods for the synthesis of well-formed concurrent specifications," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2003.

[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 Trans. Information and Systems*, vol. E80-D, 3, pp. 315–325, 1997.

[4] ——, *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer, 2002.

[5] J. C. Ebergen, "Translating programs into delay-insensitive circuits," Ph.D. dissertation, Dept. of Math and C.S., Eindhoven University of Technology, 1987.

[6] V. Khomenko and M. Koutny, "Towards an efficient algorithm for unfolding Petri nets," in *CONCUR 2001*, ser. Lect. Notes Comp. Sci. 2154, K. Larsen and M. Nielsen, Eds., 2001.

[7] M. Schaefer and W. Vogler, "Component refinement and CSC solving for STG decomposition," in *FOSSACS 05*, ser. Lect. Notes Comp. Sci. 3441, V. Sassone, Ed., pp. 348–363. Springer, 2005.

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

[9] A. Semenov, "Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding," Ph.D. dissertation, University of Necastle upon Tyne, 1997.

[10] M. Silva, E. Teruel, and J. Colom, "Linear algebraic and linear programming techniques for the analysis of place/transition net systems," in *Lectures on Petri Nets I; Basic Models*, ser. LNCS 1491, 309–373. Springer, 1998.

[11] W. Vogler and B. Kangsah, "Improved decomposition of signal transition graphs," in *ACSD 2005*, 2005, pp. 244–253, full version available: www/forschung/reports/vogler04.

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

[13] T. Yoneda, H. Onda, and C. Myers, "Synthesis of speed independent circuits based on decomposition," in *ASYNC 2004*. IEEE Computer Society Press, 2004, pp. 135–145.