Improved Decomposition of STGs

W. Vogler
Institut für Informatik
Universität Augsburg, Germany
vogler@informatik.uni-augsburg.de

B. Kangsah
FB Elektro- und Informationstechnik
Universität Kaiserlautern, Germany
kangsah@rhrk.uni-kl.de

Abstract

Signal Transition Graphs (STGs) are a version of Petri nets for the specification of asynchronous circuit behaviour. It has been suggested to decompose such a specification as a first step; this leads to a modular implementation, which can support circuit synthesis by possibly avoiding state explosion or allowing the use of library elements.

In a previous paper, the original method was extended and shown to be much more generally applicable than known before. But further extensions are necessary, and some are presented here, e.g.: to avoid dynamic auto-conflicts, the previous paper insisted on avoiding structural auto-conflicts, which is too restrictive; we show how to work with the latter type of auto-conflicts. This and another simple extension makes it necessary to restructure presentation and correctness proof of the decomposition algorithm.

1 Introduction

Decomposition of Signal Transition Graphs (STGs) has been suggested as a method to alleviate the problem of state space explosion. As a significant improvement compared to previous efforts, it was shown in [10] with a comparatively simple correctness proof that decomposition can be applied quite generally. Yet, the study of benchmark examples has revealed that further improvements of the method are needed. In the present paper, we give several improvements, also restructuring the correctness proof in a way that should ease the incorporation of further improvements.

STGs are a version of Petri nets for the specification of asynchronous circuit behaviour and supported by the tools petrify (e.g. [6]) and CASCADE [2], which in many cases can synthesize a circuit from an STG. Transitions are labelled with rising and falling edges of input and output signals; the latter are considered to be controlled by the circuit, the former by its environment. If the occurrence of an input signal is not specified in some state, this formulates the assumption on the environment not to produce this signal.

Being Petri nets, STGs allow a causality-based specification style, and they give a compact representation of the desired behaviour since they represent concurrency explicitly. As a first step in the synthesis of a circuit corresponding to a given STG \( N \), one usually constructs the reachability graph, where one might encounter the state explosion problem; i.e. the number \( r \) of reachable states (markings) might be too large to be handled. To avoid this, one can try to decompose the STG into components \( C_i \); their reachability graphs taken together can be much smaller than \( r \) since \( r \) might be the product of their sizes. Even if this is not achieved, it might already be interesting enough if each component has a smaller reachability graph: the reachability graph of \( N \) might be too large for the available memory space; even if memory is no limiting factor, further steps of the circuit synthesis might easily take time quadratic in the number of states. Decomposition can also be useful independently of size considerations: there are examples where \( N \) cannot be handled by a specific synthesis method, while the \( C_i \) can; also, one may be able to split off a library element, and this is valuable in particular for arbiters, which are complicated to synthesize; see [10] for an example.

Thus, instead of synthesizing one large circuit from \( N \), we decompose \( N \) into components \( C_i \), synthesize a circuit from each (using tools or library look-ups) and compose these circuits into one system of communicating circuits.

[5, 7] suggest decomposition methods for STGs, but these approaches can only deal with very restricted net classes. [10] deals with STGs of arbitrary graph-theoretic structure, but with some limitations on the labelling. While there is not even a correctness definition in [7], Chu proves that the parallel composition of the components generated by decomposition of \( N \) is language equivalent to \( N \). In [10], it is argued that instead of language equivalence a new bisimulation-type relation is more adequate, and this correctness definition is also used here.
The method in [5] constructs for each output signal $s$ a component $C_i$ that generates this signal and chooses some other signals as inputs. The component is obtained from the STG $N$ by contracting all transitions which do not belong to these input or output signals; these transitions are internal in intermediate stages of the decomposition. So-called secure transition contraction of internal transitions is also the main operation in [10], and components may also generate several outputs as in [7]. An additional operation is the deletion of redundant places (see e.g. [3] for an early reference), which is already essential for the decomposition of the very simple marked graphs. For a more detailed discussion of the literature, see [10].

To study the fundamental ideas of decomposition, STGs were only labelled with signals instead of signal edges in [10]. Here, we consider signal edges; in practical STGs, rising and falling edges of a signal have to alternate, i.e. the STGs have to be consistent. As one improvement we show in the present paper that decomposition of a consistent STG results in consistent components.

Other improvements of the decomposition method from [10] are motivated by our study of examples from a collection of benchmark examples that circulate in the STG-community. For example, the initial STG $N$ was required to be deterministic in [10], but some benchmark examples have internal (or dummy) transitions; it is a simple observation that we can try to remove them with secure transition contractions, which works in many cases. Further, it was required that $N$ has no structural conflicts between input and output signals; this ensures that there are no dynamic conflicts between inputs and outputs, which in principle make it impossible to turn $N$ into a reliable (i.e. hazard-free) digital circuit. The whole correctness proof in [10] makes this assumption, and ensures that there are no such structural conflicts in the constructed components. Motivated by an example, we have untangled the proof to demonstrate: correctness of decomposition does not depend on this assumption; additionally, the constructed components only have structural or dynamic conflicts between input and output signals, if $N$ has such conflicts between the same signals.

The notion of an admissible operation was introduced in [10] in order to structure the correctness proof, also with an eye on possible extensions of the method. From our study of examples, we found that we should add another simple operation: the deletion of internal transitions that are connected to places only by loops. This operation is quite trivial, but we could not insert it directly into the correctness proof of [10]. Therefore, we had to restructure this proof including a change to the notion of admissible operation, and we present decomposition in this paper in a slightly different way. This should also make further possible additions to the list of operations easier.

The rewritten proof also supports our main (and most difficult) contribution concerning auto-conflicts, i.e. conflicts between transitions labelled with the same signal edge. If such a conflict is dynamic, the STG is not deterministic and cannot be implemented directly as a circuit; a simple condition to ensure the absence of dynamic auto-conflicts without generating the reachability graph is to require the absence of structural auto-conflicts. Therefore, it is required in [10] that the initial STG is free of structural auto-conflicts; if a transition contraction introduces such a conflict, some form of backtracking is applied, and this backtracking is essential for finding the appropriate input signals for the respective component. Thus, a proper treatment of auto-conflicts is vital for our method, and great care must be taken when the treatment is modified.

There are examples where there are structural auto-conflicts initially which are not dynamic; decomposition as in [10] cannot handle such examples. We show how decomposition can be applied in such cases; if the contractions do not introduce new structural auto-conflicts (something that can be checked locally), the resulting components will not have a dynamic auto-conflict. Similarly, we can carry on without backtracking if a new structural auto-conflict is not a dynamic one; this has the potential for better results, since backtracking leads to components with more signals and, thus, larger reachability graphs.

For the time being, we assume that the user has to ensure that a structural auto-conflict is not a dynamic one. Such user intervention is of course error-prone, and in the presence of dynamic auto-conflicts the correctness proof fails, so there is the danger that the result of the decomposition will exhibit incorrect behaviour. A nice final touch to our presentation is the following result, which is not so hard to prove but important for applicability: if due to an error, a dynamic auto-conflict is present in an intermediate stage of some component, then this is essentially preserved and therefore also present in the final component. Since for circuit synthesis the reachability graph of this component will be built, the conflict will be discovered and the utilization of a faulty component can be avoided.

After presenting basic definitions of STGs in Section 2, we have a closer look at contractions in Section 3 refining the results of [10]. In Section 4, we give the new description of our method in detail. The new features of the method have been integrated in our tool DESI; we present some first results of our improvements in Section 5. Topics for future research are discussed in the conclusion in Section 6.

We thank J. Cortadella, M. Schäfer and R. Wollowski for very helpful discussions, and J. Carmona and I. Blunno for providing us with the benchmark examples. Due to lack of space, proofs are omitted; these proofs and further explanations can be found in the technical report 2004-8 at http://www.informatik.uni-augsburg.de/skripts/techreports/
2 Basic notions of Signal Transition Graphs

A Signal Transition Graph or STG is a net that models the desired behaviour of an asynchronous circuit. Its transitions are labelled with edges of signals from some alphabet \( \Sigma \) or with the empty word \( \lambda \), and we distinguish between input and output signals. A transition labelled with \( \lambda \) represents an internal, unobservable signal; in this paper, we use \( \lambda \)-labelled transitions only in the initial STG, where we call them dummy transitions which have to be removed in a first phase, or in intermediate phases of our algorithm, where we call them dividing transitions.

Thus, an STG \( N = (P, T, W, l, M_N, I_n, O_u) \) is a labelled net consisting of finite disjoint sets \( P \) of places and \( T \) of transitions, the arc weight \( W : P \times T \cup T \times P \to \mathcal{N}_0 \), the labelling \( l : T \to I_n(\{+, -\}) \cup O_u(\{+, -\}) \cup \{\lambda\} \), the initial marking \( M_N : P \to \mathcal{N}_0 \) and the disjoint sets \( I_n \subseteq \Sigma \) and \( O_u \subseteq \Sigma \) of input and output signals. Usually, an STG is required to be consistent (i.e. signal edges are required to alternate), see below.

We usually use \( a, b, c \) for input and \( x, y, z \) for output signals; \( I_n(\{+, -\}) = \{a, a^- \mid a \in I_n\} \) is the set of input signal edges, where \( a^+ \) is the rising and \( a^- \) the falling edge of signal \( a \), and the meaning of \( O_u(\{+, -\}) \) or \( \Sigma(\{+, -\}) \) is analogous. For signal, we write \( s \pm \) for any one of its edges if the direction does not matter; writing \( s \pm \) several times in some context refers to the same edge of \( s \). If \( l(t) = s \pm \), then \( s \) is the signal of \( t \) and \( t \) is observable; if \( s \in I_n \) (\( s \in O_u \) resp.), then \( t \) is an input (an output resp.) transition, drawn as a black (a white resp.) box; if \( l(t) = \lambda \), then \( t \) is an internal transition, drawn as a line.

We assume familiarity with the notions: arc, pre- and postset \( *x \) and \( x^* \), loop, tokens and marking; when a transition \( t \) or a transition sequence \( w \) is enabled under a marking \( M \) (notation \( M[t], M[w] \)) and yields \( M' = (M[t], M[w])M' \) when firing; reachable marking, safeness and boundedness. Often, STGs are assumed to be safe and to have only arcs with weight 1. In the first place, we are interested in such STGs; but we also deal with others.

We can extend the labelling to transition sequences as usual, i.e. \( l(t_1 . . . t_n) = l(t_1) . . . l(t_n) \); note that internal signals are automatically deleted. A sequence \( v \) of signal edges from \( \Sigma(\{+, -\}) \) is enabled under a marking \( M \), denoted by \( M[v] \), if there is some transition sequence \( w \) with \( M[w] \) and \( l(w) = v \); \( M[w]M' \) is defined analogously. If \( M = M_N \), then \( v \) is called a trace. The language \( L(N) \) is the set of all traces. We call two STGs language equivalent if they have the same traces.

An STG is consistent (which is usually required) if, for all signals \( s \), in every trace of the STG the edges \( s^+ \) and \( s^- \) alternate and there are no two traces where \( s^+ \) comes first in the one and \( s^- \) in the other.

An STG requires that certain outputs (or, more precisely, output signal edges) are produced provided certain inputs have occurred, namely those outputs that are enabled under the marking reached by the signal occurrences so far. At the same time, the STG describes assumptions about the environment that controls the input signals: if some input signal is not enabled, the environment is supposed not to produce this input at this stage; if it does, the specified system may show arbitrary behaviour, and it might even malfunction.

In this paper, a specification is a deterministic STG, i.e. it does not have internal transitions (but we shortly discuss these in Section 4.1) and, for each signal edge \( s \pm \), a reachable marking enables at most one \( s \pm \)-labelled transition.

Two different transitions \( t_1 \) and \( t_2 \) with \( M[t_1] \) and \( M[t_2] \) (\( M \) a reachable marking) are enabled concurrently if \( W(p, t_1) + W(p, t_2) \leq M(p) \) for all \( p \in P \), and in (dynamic) conflict otherwise. If they have the same label, the STG has auto-concurrency, a (dynamic) auto-conflict resp. (Note that the former cannot occur in a consistent STG; we try to make little use of this in our proofs in order to allow application of our results outside the area of circuit design.) If the signal of one of the transitions is an input signal, while the signal of the other is an output signal, then in the latter case the STG has a (dynamic) input/output-conflict.

Two different transitions \( t_1 \) and \( t_2 \) and their signals – are in structural conflict if \( *t_1 \cap *t_2 \neq \emptyset \). If both transitions are labelled with the same signal edge \( s \pm \), then \( s \) is in structural auto-conflict and the STG has such a conflict. If \( t_1 \) is an input (or a \( \lambda \)-labelled) and \( t_2 \) an output transition, then they form a structural input/output conflict (or a structural non-output conflict) and the STG has such a conflict.

Clearly, an STG without internal transitions is deterministic if and only if it is without auto-concurrency and without dynamic auto-conflict; the latter is ensured if there are no structural auto-conflicts. Note that internal transitions enabled concurrently or being in conflict do not introduce auto-concurrency or -conflict.

Often, having the same behaviour is understood as language equivalence, but just as often one has to consider the more detailed behaviour equivalence bisimilarity. A bisimulation between \( N_1 \) and \( N_2 \) is a relation \( B \) between markings of \( N_1 \) and \( N_2 \) such that \( (M_{N_1}, M_{N_2}) \in B \) and for all \( (M_1, M_2) \in B \) we have: if \( M_1[t]M'_1 \), then there is some \( M_2[t']M'_2 \) with \( M_2[t][l(t)]M'_2 \) and \( (M'_1, M'_2) \in B \) and vice versa. If such a bisimulation exists, we call the STGs bisimilar. For deterministic STGs, language equivalence and bisimulation coincide.

In a parallel composition, the composed systems run in parallel synchronizing on common signals. Since a system controls its outputs, we cannot allow a signal to be an output of more than one component. An output signal of one component can be an input of one or several others, and in any case it is an output of the composition.
The parallel composition of STGs $N_1$ and $N_2$ is defined if $\text{Out}_1 \cap \text{Out}_2 = \emptyset$. Then, let $A = (\text{In}_1 \cup \text{Out}_1) \cap (\text{In}_2 \cup \text{Out}_2)$ be the set of common signals. To get the parallel composition $N = N_1 \parallel N_2$, take the disjoint union of $N_1$ and $N_2$ and then combine each $s$±-labelled transition of $N_1$ with each $s$±-labelled transition of $N_2$ if $s \in A$. Finally, put $\text{In} = (\text{In}_1 \cup \text{In}_2) - (\text{Out}_1 \cup \text{Out}_2)$ and $\text{Out} = \text{Out}_1 \cup \text{Out}_2$.

It should be clear that, up to isomorphism, composition is associative and commutative. Therefore, we can define the parallel composition of a family (or collection) $(C_i)_{i \in I}$ of STGs as $(\bigcup_{i \in I} C_i)$, provided that no signal is an output signal of more than one of the $C_i$. We can consider a marking $M$ of a composition as the disjoint union of markings of the components, which we denote by $M \parallel p_i$ (in Definitions 4.1.2d) and 4.2.2c).

### 3 Transition contraction and redundant places

Transition contraction (see e.g. [1]) is most important in our decomposition procedure. In the following definition, we add the new notion of a new conflict pair.

**Definition 3.1** Let $N$ be an STG and $t \in T$ with $\{t\} = \emptyset$, $\forall p \in P : W(p, t, W(t, p)) \leq 1$ and $l(t) = \lambda$. We define the $t$-contraction $\overline{N}$ of $N$ by

$$T' = \{ (p, \star) | p \in P - \{t \cup \star \} \}$$

$$\overline{T} = T - \{t\} \quad \overline{\mathcal{T}} = \text{In} \quad \overline{\text{Out}} = \text{Out}$$

$$W((p, p'), t_1) = W(p, t_1) + W(p', t_1)$$

$$W(t_1, (p, p')) = W(t_1, p) + W(t_1, p')$$

$$M_\overline{N}((p, p')) = M_N(p) + M_N(p')$$

Here, $\star \notin P \cup T$ is a dummy element with $W(\star, t_1) = W(t_1, \star) = M_N(\star) = 0$.

For two different transitions $t_1, t_2$ with $t_1 \neq t_2$, we call $\{t_1, t_2\}$ a new conflict pair whenever $\{t \cap t_1 \neq \emptyset \}$ and $\{t \cap t_2 \neq \emptyset \}$ in $N$ (or vice versa).

![Figure 1](image)

Note that $\overline{N}$ might fail to be e.g. consistent; we will have to study for which cases it is consistent, since this is usually required. Figure 1 shows a simple example of a contraction. Here, the $b+$ and the $c+$-labelled transition form a new conflict pair; note that this is also true, if they already had a common place (not drawn) in their presets in $N$ – they now have a new such place.

For the rest of this section, we fix an STG $N$ with a transition $t$ satisfying the requirements of Definition 3.1 and denote its $t$-contraction by $\overline{N}$. The first theorem shows that contraction preserves behaviour in a weak sense; the refined method of the present paper needs its second part, for which we introduce a new notion:

**Definition 3.2** An operation transforming an STG $S$ into an STG $S'$ is auto-cc-preserving whenever the following holds: if there are two transitions labelled with the same signal edge and enabled under the same reachable marking of $S$, then the same is true for $S'$. (In other words, if $S$ has auto-concurrency or an auto-conflict, then so has $S'$.)

**Theorem 3.3** $L(N) \subseteq L(\overline{N})$. Transition contraction is auto-cc-preserving.

The next two results show that under additional assumptions contraction preserves behaviour in a stronger sense. In both theorems, the last but one and the new last sentence are what is really needed for the remainder of the paper.

**Theorem 3.4** Assume that $(\{t\}) \subseteq \{t\}$. Then $N$ and $\overline{N}$ are bisimilar. The contraction preserves boundedness and freedom from auto-concurrency. If $N$ is free of dynamic auto-conflicts, then so is $\overline{N}$.

**Theorem 3.5** Assume that $\{t\} = \{t\}$; in particular, $t^* \neq \emptyset$. Further, assume that $\exists p_0 \in r^* : M_N(p_0) = 0$. Then $N$ and $\overline{N}$ are language equivalent. The contraction preserves boundedness and freedom from auto-concurrency. If $N$ is free of dynamic auto-conflicts, then $\overline{N}$ is not due to some $t_1$ and $t_2$, then $\{t_1, t_2\}$ is a new conflict pair.

If the preconditions of Definition 3.1 and Theorem 3.4 (3.5 resp.) are satisfied, then we call the contraction of $t$ secure of type 1 (of type 2 resp.).

As to our results about the introduction of auto-conflicts, observe that this would be rather trivial for structural auto-conflicts: If $\overline{N}$ has a structural auto-conflict, while $N$ has not, then certainly this is due to a new conflict pair – and this holds for an arbitrary contraction, secure or not. For dynamic auto-conflicts as treated in the above theorems, the situation is different. Figure 2 shows an STG $N$ and the STG $\overline{N}$ obtained by a non-secure contraction: while $N$ is actually dead (i.e. no transition can fire), $\overline{N}$ has a dynamic auto-conflict. Violation of soundness allows a behaviour that intuitively corresponds to a backfiring of $t$.

We call an operation on STGs consistency-preserving, if it turns a consistent STG into one that is consistent again.
Corollary 3.6 Secure transition contractions are consistency-preserving.

We conclude this section with a definition: A place \( p \) of an STG \( S \) is (structurally) redundant (see e.g. [3]) if there is a set of places \( Q \) with \( p \notin Q \), a valuation \( V : Q \cup \{ p \} \rightarrow N \) and some \( c \in N_0 \) such that for all transitions \( t \):

1. \( V(p)M_s(p) = \sum_{q \in Q} V(q)M_s(q) = c \)
2. \( V(p)(W(t,p) - W(p,t)) = \sum_{q \in Q} V(q)(W(t,q) - W(q,t)) \leq 0 \)
3. \( V(p)W(p,t) = \sum_{q \in Q} V(q)W(q,t) \leq c \)

The full version explains why deletion of a redundant place gives a bisimilar STG. A special case of a redundant place is a loop-only place, i.e. a marked place \( p \) such that \( p \) and \( t \) form a loop with arcs of weight 1 for all \( t \in p \cup p^* \). Another simple case is that of a duplicate: place \( p \) is an (extended) duplicate of place \( q \), if for all transitions \( t \) \( W(t,p) = W(t,q) \), \( W(p,t) = W(q,t) \) and \( M_N(p) \geq M_N(q) \).

4 Decomposing a Signal Transition Graph

4.1 Correctness Definition

The first new observation of the present paper concerns the following: though an STG should be deterministic for the application in circuit design, it seems to be convenient to allow \( \lambda \)-transitions in \( N \) where these so-called dummy transitions do not represent state changes, i.e. what really matters about \( N \) is just its language; hence, when synthesizing a circuit via the reachability graph of \( N \), one can remove the resulting \( \lambda \)-arcs in the reachability graph by well-known automata-theoretic methods – but we want to avoid the construction of this graph. With the results of the previous section, we can instead contract the dummy transitions provided these contractions are secure. If this succeeds, we can just as well work with the resulting STG if it is free of dynamic auto-conflicts.

Henceforth, we assume that we are given a fixed deterministic STG \( N \) as a specification. In contrast to [10], we do not require \( N \) to be free of structural auto-conflicts. As in [10], we will construct components that are also deterministic.

Besides determinism, [10] assumes further that \( N \) is free of structural input/output conflicts; this ensures that there are no dynamic input/output conflicts, which are very hard to implement. But in the literature, we have found an example with a structural input/output conflict, which is even also a dynamic one; cf. stg_burleso in Section 6. We take this as motivation to generalize the approach of [10] to STGs with input/output conflicts; but given the problems such conflicts can create, we still regard it as important that our method does not introduce any new input/output conflicts.

We first repeat the definition when a collection of components is a correct implementation of \( N \); for a detailed explanation and a discussion of related work, we refer to [10]. We will additionally show that the components our algorithm generates are consistent if \( N \) is.

Definition 4.1 A collection of deterministic components \((C_i)_{i \in \Pi}\) is a correct decomposition or a correct implementation of a deterministic STG \( N \), if the parallel composition \( C \) of the \( C_i \) is defined, \( In_C \subseteq In_N, Out_C \subseteq Out_N \) and there is a relation \( B \) between the markings of \( N \) and those of \( C \) with the following properties.

1. \( (M_N, M_C) \in B \)
2. For all \((M, M') \in B\), we have:
   a) If \( a \in In_N \) and \( M'[a] \neq M_1 \), then either \( a \not\in In_C \) and \( M'[a] \neq M_1' \) or \( M_1 \not\in B \) for some \( M_1' \) or \( a \not\in In_C \) and \( M_1 \not\in B \).
   b) If \( x \in Out_N \) and \( M'[x] \neq M_1 \), then \( M'[x] \neq M_1' \) and \( (M_1, M_1') \in B \) for some \( M_1' \).
   c) If \( x \in Out_C \) and \( M'[x] \neq M_1 \), then \( M'[x] \neq M_1 \) and \( (M_1, M_1') \in B \) for some \( M_1' \).
   d) If \( x \in Out_i \) for some \( i \in I \) and \( M'[x] \neq M_1 \), then \( M'[x] \neq M_1 \) and \( (M_1, M_1') \in B \) for some \( M_1' \).

Here, and whenever we have a collection \((C_i)_{i \in I}\) in the following, \( P_i \) stands for \( P_{C_i}, Out_i \) for \( Out_{C_i} \), etc.

4.1 requires that the behaviours on \( N \) and \( C \) match as for bisimilarity, except for the following points. We allow \( C \) to have fewer input and output signals than \( N \); this is possible for input signals which are irrelevant for producing the right outputs and for outputs that actually never have to be produced; our method may find such irrelevant inputs, but it always preserves all outputs. There is no clause requiring a match for inputs of \( C \); this implies that \( N \) and \( C \) are not necessarily language equivalent, as e.g. required in [5]; see [10] for a discussion why this is adequate and useful. Clause (d) is important for a correct functioning; if it is violated, then component \( C_i \) could produce some output.
that some other component is not ready to accept, which could lead to malfunction of this other component. Finally, it should be pointed out that the chosen style of this definition is technically useful in the correctness proof.

4.2 The Decomposition Algorithm

We start with a rough description of our algorithm, where the unknown notions will be described later in this section: Given \(N\) and a feasible partition of its signals, the algorithm constructs an initial decomposition \((C_i)_{i \in I}\). Then it repeatedly applies a totally admissible operation (from some list of such operations) or backtracking to one of the \(C_i\) after the other until it does not contain any \(\lambda\)-transitions anymore.

To initialize the algorithm, one has to choose a feasible partition of the signals of \(N\); we still have to develop heuristics how to find good ones. Our condition (C1) for such a partition is different from [10], since we allow input/output-conflicts in this paper. A feasible partition is a family \((I_i, \text{Out}_i)_{i \in I}\) for some set \(I\) such that the sets \(\text{Out}_i\), \(i \in I\), are a partition of \(\text{Out}_N\) and for each \(i \in I\) we have \(I_i \subseteq \text{In}_N \cup \text{Out}_N \setminus \text{Out}_i\), and furthermore:

(C1) If signal \(s\) and output signal \(x\) of \(N\) are in structural conflict, then \(x \in \text{Out}_i\) implies \(s \in \text{In}_i\) if \(s \in \text{In}_N\) and \(s \in \text{Out}_i\) if \(s \in \text{Out}_N\) for each \(i \in I\).

The rationale for this is that a component responsible for output signal \(x\) must at least ‘see’ any signal that could be in dynamic conflict with \(x\) in \(N\); if such a signal is an output as well, the component should also produce it, because otherwise we would have a new input/output conflict.

(C2) If \(t \cap t' \neq \emptyset\) for \(t, t' \in T_N\) and the signal of \(t'\) is in some \(\text{Out}_i\), then the signal of \(t\) is in \(\text{In}_i \cup \text{Out}_i\). (The latter might be in \(\text{In}_i\) even if it belongs to \(\text{Out}_i\); then it will be produced by some other component, and the \(i\)th component just listens to it.)

For a feasible partition, the initial decomposition is \((C_i)_{i \in I}\), where each initial component \(C_i = (P, T, W, l_i, M_N, I_i, \text{Out}_i)\) is a copy of \(N\) except for the labelling and the signal classification: \(l_i(t) = l(t)\) if the signal of \(t\) is in \(\text{In}_i \cup \text{Out}_i\), and \(l_i(t) = \lambda\) otherwise — in which case we speak of a divining transitions.

The main idea of the algorithm is now to remove the \(\lambda\)-transitions using secure transition contractions. Unfortunately, the algorithm could get stuck, e.g. when no secure contraction is applicable although there are still \(\lambda\)-transitions left; therefore, it can also do the following:

- **Backtracking**: backtracking applied to some \(C_i\) and some signal \(s \notin \text{In}_i \cup \text{Out}_i\) adds \(s\) to \(\text{In}_i\) and replaces \(C_i\) by the respective new initial component.

Observe that this modifies the feasible partition such that the resulting partition is feasible again; in particular, \(C_i\) already has all signals that are in structural conflict to an output signal of \(C_i\). Backtracking undoes all the totally admissible operations that have already been performed on \(C_i\). In many cases, it will be possible to perform some of these also on the new initial component; hence, we will study in the future how to implement backtracking such that not always all the operations are undone.

Here is the list of totally admissible operations we use in this paper. In the context of STG-decomposition, RedPD was suggested for the first time in [10], while RedTD is new; further operations may turn up in the future.

- **SecTC**: Perform a secure transition contraction to some \(t\) of some \(C_i\), provided this gives an STG without dynamic auto-conflicts.

If there is such a conflict, the algorithm will perform backtracking on \(C_i\) and the signal of \(t\) next — but this should not be seen as part of SecTC. The rationale for this is: the dynamic auto-conflict shows that \(C_i\) should better know about the signal edge labelling \(t\) in \(N\) in order to decide which of the two equally labelled transitions to fire. It is not obvious how to detect dynamic auto-conflicts without building the reachability graph; we will discuss this at the end of Subsection 4.4.

- **RedPD**: Delete a redundant place in some \(C_i\).

- **RedTD**: Delete a divining transition \(t\) in some \(C_i\), where either each place \(p \in t \cup t'\) forms a loop with \(t\) with two arcs of the same weight (\(t\) is a loop-only transition) or some other divining transition has arcs to and from the same places with the same weight as \(t\) (which is a duplicate transition).

The latter two operations seem rather trivial since they clearly do not change the behaviour of the respective \(C_i\). Still, both of them have turned out to be essential for good results in some cases; furthermore, both operations change the net structure, and since our notion of admissible operation refers to the net structure, some care should be taken when adding them to the list. In fact, we structure the presentation of the algorithm and of the proof differently from [10], partly because it was not possible to add RedTD directly to the algorithm as described in [10].

There, operations are restricted to SecTC and RedPD, while backtracking was explicitly described in the algorithm as exception handling in case of auto-conflicts. Our new presentation treats this as an implementation detail, as a strategy for choosing the next operation; this seems much better suited for adding further operations.

Note that the algorithm is nondeterministic, and already [10] shows an example where also its result is not unique. This issue deserves further investigations, in particular since different results might give reachability graphs of different sizes; see also [8]. In some cases the algorithm might even
fail to produce something useful, but this is actually not surprising since one cannot expect to fight state explosion successfully in all cases. All one can hope for are reasonable results reasonably often; in the examples we have checked, the algorithm most often performed quite well.

### 4.3 Correctness

The correctness proof shows that the use of totally admissible operations guarantees a correct result; it then remains to check that our operations from above are totally admissible. Only this check has to be performed when adding further operations to the algorithm, while the first part of the proof can be reused. We will first discuss the issue of termination, where our presentation deviates from [10]. For the definition of a totally admissible operation, one has to fix a function (possibly referring to N) from STGs with signals in In ∪ Out into some set with a well-founded ordering as a termination function; we choose here the function that gives for such an STG C the triple \((sc, tc, pe)\), where sc is \(|In \cup Out| - |In_C \cup Out_C|\) (the number of signals missing in C), tc is the number of transitions, and pe is the number of places of C. We order such triples lexicographically according to the standard ≤ on natural numbers.

When extending the above list of operations in future work, it might be necessary to modify this termination function; in this sense, our approach should be seen as parametric with the termination function as parameter.

A totally admissible operation is an admissible operation that applied to an STG with signals in In ∪ Out (and we assume that it is only applied to such STG) does not change the signals and decreases the termination function.

As long as there is still some divining transition in some \(C_i\), an operation can be applied – we can always choose backtracking. When backtracking is applied, the number of signals missing in \(C_i\) goes down and so the value of the termination function decreases for all operations applied in the algorithm; hence, the algorithm terminates.

Backtracking changes the feasible partition; had we started out with this new partition, the operations performed on some other \(C_j\) before the backtracking would give the same result. Further, we have already concluded that backtracking can only be applied finitely often. Thus, the result of the algorithm can be seen as being obtained (from a modified feasible partition) by totally admissible operations alone (without any backtracking), and it suffices to show correctness for such a case.

Before we define admissible operations, we have to introduce a central notion in our correctness proof; it is a variant of a bisimulation with an angelic treatment of internal transitions (which are seen as divining), and it is (like a loop invariant) needed to describe in what sense the intermediate stages of our algorithm are correct. Note that angelic correctness is just a mathematical tool for our proof.

**Definition 4.2** A collection of components \((C_i)_{i \in I}\) is an angelically correct decomposition or implementation of a deterministic STG \(N\), if the parallel composition \(C\) of the \(C_i\) is defined, \(In_C \subseteq In_N\), \(Out_C \subseteq Out_N\) and there is an angelic bisimulation relation \(B\) between the markings of \(N\) and those of \(C\), i.e. \(B\) satisfies the following properties.

1. \((MN, MC) \in B\)
2. For all \((M, M') \in B\), we have:
   a) If \(a \in In_N\) and \(M[a \pm]\), then either \(a \in In_C\) and \(M'[a \pm]\) and \((M_1, M_1') \in B\) for some \(M_1\) or \(a \notin In_C\) and \(M'[a\lambda]\) and \((M_1, M_1') \in B\) for some \(M_1\).
   b) If \(x \in Out_N\) and \(M[x \pm]\), then \(M'[x \pm]\) and \((M_1, M_1') \in B\) for some \(M_1\).
   c) If \(x \in Out_i\) for some \(i \in I\) and \(M'[x \pm]\), then some \(M_i'\) and \(M_1\) satisfy \(M'[x \pm]M_i'\) and \((M_1, M_1') \in B\).

The differences to Definition 4.1 are that here \([x \pm]\) in \(C\) might involve additional λ-transitions besides an \(x\)-labelled transition, that in 2(a) internal transitions are allowed to match an input of \(N\) that is not one of \(C\), and that 2(c) is a combination of 4.1.2(c) and (d) and guarantees a matching only for some \(M'_i\) – this is an angelic part of the definition. It is also angelic that we do not require a match for the firing of only internal transitions in \(C\).

**Definition 4.3** An operation is pre-admissible if, whenever applied to an STG without auto-concurrency and dynamic auto-conflicts and satisfying a) and b) below, it gives an STG satisfying these four properties again:

a) There is no structural λ/output conflict.
   b) If \(t_2\) is an output transition and \(t_1^* \cap t_2^* \neq \emptyset\), then \(t_1\) is not an internal transition.

We call a pre-admissible operation applied to some member of a family \((C_i)_{i \in I}\) that satisfies a) and b) above admissible if it preserves angelic correctness w.r.t. \(N\).

Note that a) above differs from its version in [10], since we allow input/output conflicts. Due to the definition of pre-admissible, each intermediate \(C_i\) satisfies a) and b), and the resulting \(C_i\) are deterministic. These give a correct decomposition of \(N\) due to admissibility. Conditions a) and b) are needed to show that SecTC preserves angelic correctness.

It remains to show that each of our operations is totally admissible. All in all, we will then have the following correctness results, for which we need one further notion.

**Definition 4.4** An operation transforming some STG \(S\) to an STG \(S'\) does not introduce io-conflicts if, for any structural, dynamic resp., input/output-conflict in \(S'\), \(S\) already has one for the same signals.
Theorem 4.5 1. The decomposition algorithm terminates for each deterministic STG \( N \); the resulting components \((C_i)_{i \in I}\) are deterministic and form a correct implementation of \( N \).

2. If the decomposition algorithm uses only the operations presented in this paper, we have additionally: if \( N \) is consistent, then so are the components \((C_i)_{i \in I}\); if some of the components has a structural, dynamic resp., input/output-conflict then \( N \) has one for the same signals.

4.4 Admissible Operations

We now prove that the operations on the list above are indeed admissible. It is easy to see that then they are also totally admissible: the number of missing signals is unchanged by all three operations; SecTC reduces the number of transitions, which also holds for RedTD; RedPD leaves this number unchanged but reduces the number of places. We also check that each operation is consistency- and auto-cc-preserving and that none of them introduces io-conflicts. The first theorem is based on the following easy lemma (but note the condition on signals):

Lemma 4.6 Let a pre-admissible operation be given that, applied to some member of a family \((C_i)_{i \in I}\) satisfying a) and b) of Definition 4.3, transforms some \( C_j \) to a bisimilar \( \overline{C_j} \) with the same input and output signals. Then the operation is admissible.

Theorem 4.7 RedPD and RedTD are consistency- and auto-cc-preserving, they are admissible operations and do not introduce io-conflicts.

One shows the corresponding result for SecTC in several steps.

Theorem 4.8

1. SecTC is an admissible operation, and it is consistency- and auto-cc-preserving.

2. When applied in the algorithm, SecTC does not introduce io-conflicts.

It remains to discuss one vital issue about SecTC: how can we check whether a secure transition contraction has introduced some auto-conflict? There are a number of strategies to deal with this question:

Conservative strategy: In [10], it was suggested to restrict attention to STGs without structural auto-conflicts, i.e.: the input specification \( N \) has to be free of structural auto-conflicts, and a transition contraction is forbidden if it would create a structural auto-conflict. (Then, instead, backtracking is performed.)

Clearly, there is no dynamic auto-conflict if there is not a structural one. Also, it is easy to check for structural auto-conflicts, since they can only appear in form of new conflict pairs. But this strategy is over-cautious in some cases.

Specifier-dependent strategy: Accept inputs with structural auto-conflicts, if the specifier guarantees that they are not dynamic ones. A secure transition contraction is only forbidden, if it is of type 2 and a new conflict pair forms a structural auto-conflict. For example, this strategy works well for vmecon, discussed in the next section.

This strategy relies on our new results in Thms. 3.4 and 3.5. Without them, the specifier guaranteeing absence of dynamic auto-conflicts would not ensure this absence after some contraction, even if no new conflict pairs are created.

This is still a conservative strategy, since not each new structural auto-conflict based on a new conflict pair has to be a dynamic one. It will certainly be more difficult to deal with these intermediate results of our algorithm, but with human insight the user still might be helpful, e.g. by using place invariants:

Interactive strategy: Accept inputs with structural auto-conflicts, as long as they are not dynamic ones. If a secure transition contraction of type 2 creates a new conflict pair forming a structural auto-conflict, ask for human intervention; if the user can ensure that there is no dynamic auto-conflict, the contraction can be performed, otherwise it is forbidden – and backtracking is performed instead.

One might worry that human intervention is error-prone: if the specifier’s or the user’s guarantee are wrong, the result of the algorithm might be wrong without anyone noticing. Using the concept of auto-cc-preservation, we have provided results to avoid this problem: Theorems 3.3. and 4.7 show that a dynamic auto-conflict arising during a run will survive in the form of an auto-conflict or auto-concurrency to the respective final component. When synthesizing a circuit from this component, one will generate its reachability graph, and during this generation the problem will show up.

As an extreme application, we can run the algorithm simply hoping that new conflict pairs are no problem.

Risky strategy: Accept inputs with structural auto-conflicts, as long as they are not dynamic ones. All secure transition contractions are allowed. In the end, build the reachability graphs of the final components and check that no reachable marking enables two equally labelled transitions; if the check fails, discard the result.

The risk is clearly that all time spent on the algorithm might be wasted. But if our hopes are justified – as e.g. for locked2 mentioned in the next section –, we get a result with fairly small reachability graphs comparatively cheap.
5 Examples

We will now demonstrate the new features of our improved algorithm for some realistic examples, discussing two of them in some detail; for even more details see http://www.eit.uni-kl.de/beister/eng/projects/deco_examples/main_examples.html

The decomposition algorithm with the new features presented here has been implemented in our tool DESI; DESI can follow the conservative, the specifier-dependent and the risky strategy. (At the time of writing, the tool only checks for loop-only and duplicate places, but not for redundant places in general.) The table below gives a numbered list of some examples we treated with the improvements of this paper, where NEI is the NEI-arbiter from e.g. [11] (see below), while the others are from the benchmark examples. (Results on some other examples are listed in [10].) For each example, we give the number \( r \) of reachable markings together with the area \( a \) of a circuit synthesized by petrify and the CPU-time \( c \) (in sec.) for the synthesis on an Intel Xeon 2.2Ghz with 1 GB memory; the last column gives the respective numbers \( r_i \), \( a_i \) and \( c_i \) for the components of the best decomposition we have found so far using the specifier-dependent strategy. (For locked2 we also show a better result obtained with the risky strategy; but see below.) The last but one column gives the sums \( R \) and \( A \) of the \( r_i \) and the \( a_i \) and the sum \( C \) of all the \( c_i \), plus the time DESI took. (The area-values are listed for completeness; we do not expect improved values for \( A \).) Observe the time savings e.g. in examples 2 (even though \( r < R \)) and 4.

<table>
<thead>
<tr>
<th>name</th>
<th>( r/a/c )</th>
<th>( R/A/C )</th>
<th>( r_i/a_i/c_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>locked2</td>
<td>168/23</td>
<td>82/29</td>
<td>70/19 6/5 6/5</td>
</tr>
<tr>
<td>(risky)</td>
<td>6.91</td>
<td>2.36</td>
<td>1.91 0.03 0.03</td>
</tr>
<tr>
<td>tsend</td>
<td>36/38</td>
<td>41/35</td>
<td>25/17 16/18</td>
</tr>
<tr>
<td>_csn</td>
<td>9.43</td>
<td>1.40</td>
<td>0.81 0.4</td>
</tr>
<tr>
<td>mux2</td>
<td>101/68</td>
<td>93/109</td>
<td>50/58 43/51</td>
</tr>
<tr>
<td>255.42</td>
<td>50.26</td>
<td>42.68</td>
<td>7.07</td>
</tr>
<tr>
<td>stg</td>
<td>1241/54</td>
<td>248/36</td>
<td>102/28 44/20</td>
</tr>
<tr>
<td>_blunno</td>
<td>401.35</td>
<td>2.95</td>
<td>0.59 0.59 0.5</td>
</tr>
<tr>
<td>vmecon</td>
<td>24/19</td>
<td>27/22</td>
<td>19/16 8/6</td>
</tr>
<tr>
<td>1.72</td>
<td>0.57</td>
<td>0.37</td>
<td>0.07</td>
</tr>
<tr>
<td>pe-send</td>
<td>117/50</td>
<td>100/62</td>
<td>68/38 32/24</td>
</tr>
<tr>
<td>_ifc</td>
<td>2.07</td>
<td>4.82</td>
<td>4.18 3.35</td>
</tr>
<tr>
<td>NEI</td>
<td>42/-</td>
<td>33/-</td>
<td>21/12 12/</td>
</tr>
</tbody>
</table>

Examples 1–4 have dummy transitions, which in these examples can now be handled by DESI without any initial human adaption. The two small components of locked2 are significantly smaller due to the new operation RedTD, as we demonstrate in the full version; also for mux2, this gives a small improvement. Examples 4–7 all have structural auto-conflicts; treatment of these with the decomposition method is only possible due to our improvements. Additionally, stg_blunno also has an io-conflict. NEI cannot be synthesized by petrify, but DESI splits it into a standard ME-element, see [10] for further discussion, and a smaller synthesizable STG. The smaller component for locked2 obtained with the risky strategy cannot be synthesized by petrify due to CSC-problems; cf. the discussion in Section 6.

A small example with some structural auto-conflict initially is vmecon as shown in Fig. 3 with a conflict between \( t_3 \) and \( t_{10} \). A component producing signal \( ds \) must see signals \( ltdack, dsr, dsw \), while a component producing signal \( lds \) must see just these four signals. Hence, the only feasible partition of interest contains \( \{ ltdack, dsr, dsw \}, \{ ds, lds \} \) and \( \{ ds, dsw \}, \{ ltdack \} \). For the first component, one has to contract \( t_0, t_7 \) and \( t_9 \). Since no new conflict pair turns up, the specifier-dependent strategy works without any backtracking – although there always is a structural auto-conflict. For the second component, one finds new conflict pairs when contracting \( t_3, t_8, t_{10} \) and \( t_{11} \). But these either involve at least one internal transition or \( t_q \) and \( t_9 \), the latter two have the same signal, but different labels, so for the second component the specifier-dependent strategy works without any backtracking as well.

6 Conclusion and future work

We have presented several improvements for the STG-decomposition algorithm presented in [10]. A number of our results allow us to deal with structural auto-conflicts: the decomposition algorithm remains correct when there are no dynamic auto-conflicts. Hence, we can carry on even if there are structural auto-conflicts initially or at intermediate stages, provided these are not dynamic ones. If we do so, we only have to check new structural auto-conflicts that turn up in a contraction. If such a check is passed by mistake – e.g. made by the user in an interactive strategy –, this mistake was proven to be recognizable during synthesis.

We have added the deletion of loop-only transitions to the list of operations allowed in the algorithm, shown how to deal with io-conflicts, consistency and dummy transitions, and restructured the correctness proof of [10] to cover these additions. The improvements have been integrated in our tool DESI, and we have demonstrated the usefulness of our improvements with some practical examples.

For synthesizing a circuit from an STG, it is important that the STG has what is called CSC; if an STG fails to have CSC, petrify tries to insert new internal signals to achieve it – and where necessary this was done for the components we reported about in the previous section. In a companion paper [9], it is shown that the components produced by our method are still a correct decomposition of the original specification after such insertions; for this, the correctness definition had to be extended to internal signals.

Two related approaches to decomposition only work un-
There are some other open problems we will tackle in the near future. E.g. we will improve the recognition of redundant places in our tool DESI. Often, essentially the same operations have to be performed in different components; DESI should be enabled to reuse the results of some operations. Also, backtracking makes DESI start with a new initial component, while reuse of some operations performed before the backtracking should be possible; [9] also contains the theoretical underpinning for this.

References