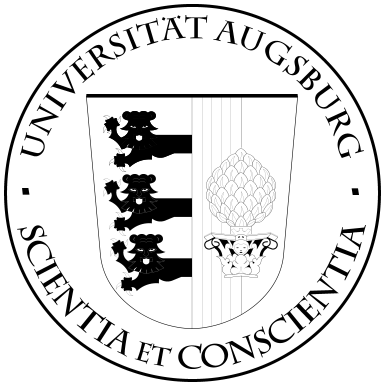
Output-determinacy and asynchronous circuit synthesis

Victor Khomenko, Mark Schaefer, Walter Vogler

Angaben zur Veröffentlichung / Publication details:

Khomenko, Victor, Mark Schaefer, and Walter Vogler. 2007. “Output-determinacy and asynchronous circuit synthesis.” Augsburg: Universität Augsburg.

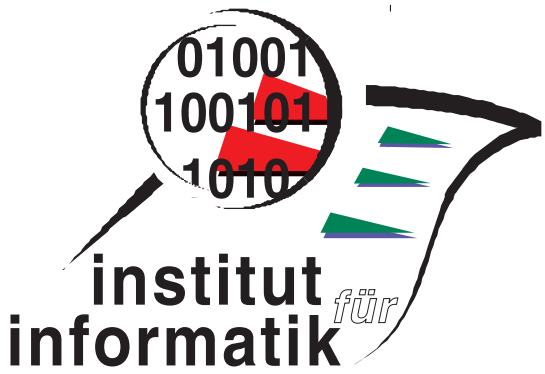
Universität Augsburg



Output-Determinacy and Asynchronous Circuit Synthesis

Victor Khomenko
Mark Schaefer
Walter Vogler

Report 2007-02 January 2007



Institut für Informatik
D-86135 Augsburg

Output-Determinacy and Asynchronous Circuit Synthesis

Victor Khomenko[1], Mark Schaefer[2] and Walter Vogler[2]
[1]School of Computing Science, Newcastle University, UK
victor.khomenko@ncl.ac.uk

[2]Institute of Computer Science, University of Augsburg, Germany
mark.schaefer@informatik.uni-augsburg.de
walter.vogler@informatik.uni-augsburg.de

Abstract

Signal Transition Graphs (STG) are a formalism for the description of asynchronous circuit behaviour. In this paper we propose (and justify) a formal semantics of non-deterministic STGs with dummies and OR-causality. For this, we introduce the concept of *output-determinacy*, which is a relaxation of determinism, and argue that it is reasonable and useful in the speed-independent context.

We apply the developed theory to improve an STG decomposition algorithm used to tackle the state explosion problem during circuit synthesis, and applied this improved algorithm to some benchmark examples.
Keywords: output-determinacy, decomposition, STG, asynchronous circuits, OR-causality.

1 Introduction

Asynchronous circuits are a promising type of digital circuits. They have lower power consumption and electromagnetic emission, no problems with clock skew and related subtle issues, and are fundamentally more tolerant of voltage, temperature and manufacturing process variations. The International Technology Roadmap for Semiconductors report on Design [ITR05] predicts that 22% of the designs will be driven by handshake clocking (i.e., asynchronous) in 2013, and this percentage will raise up to 40% in 2020.

In this paper we are concerned with an important subclass of asynchronous circuits, called *speed-independent* circuits, i.e., circuits which work correctly regardless of their gates' delays (the wires are assumed to have no delays). *Signal Transition Graphs (STGs)* [Chu87] are a formalism for the specification of such circuits. They are a class of interpreted Petri nets in which transitions are labelled with the rising and falling edges of circuit signals.

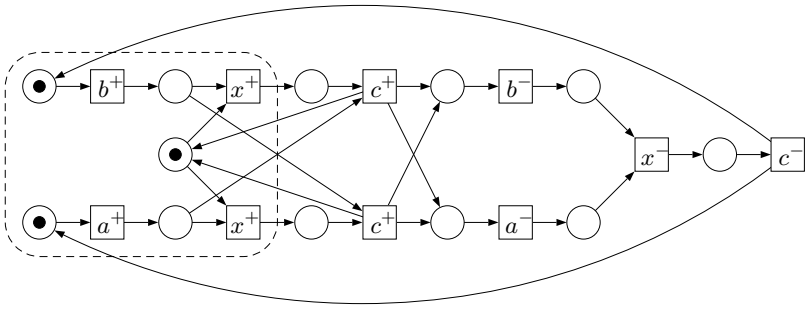
When a circuit is synthesised from an STG, it is often assumed that the specification is deterministic (in the sense of automata theory), and its semantics is the set of its possible traces, i.e., its language. As the final implementation must be deterministic, it may seem reasonable to confine oneself to deterministic specifications only. However, sometimes this turns out to be too restrictive in practice. There are several situations which naturally give rise to non-deterministic specifications which still can be synthesised:

Dummy transitions For convenience of modelling, the designers often use *dummy* transitions in STGs, which are 'silent' transitions not corresponding to any signal change. Such transitions make the STG non-deterministic.

OR-causality When modelling a situation with a safe Petri net, where the system has to respond to any of several possible stimuli in the same way, non-determinism naturally arises,[1] as shown in Fig. 1. OR-causality has been studied in [YKKL94, YKK+96].

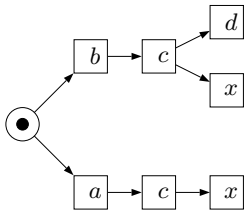
Hiding of signals Non-determinism naturally arises when in a deterministic specification some of the signals are hidden, as illustrated in Fig. 2. In fact, hiding of signals is an essential part of the decomposition algorithm of [VW02, VK05], which we will improve in the present paper.

[1]OR-causality can also be modelled as an non-safe Petri net without non-determinism [YKKL94, YKK+96], but in practice safe Petri nets are preferable as they are much easier to analyse.



inputs: a, b, c; **outputs:** x

Figure 1: OR-causality (the 'interesting' part of the STG is highlighted): a^+ and b^+ are concurrent inputs, and the output x^+ can be produced upon arrival of either of them. Note that the two transitions labelled x^+ are in dynamic auto-conflict, i.e., the specification is non-deterministic. However, it still can be implemented by the deterministic circuit $[x] = a \vee b$.



inputs: a, b, c, d; **outputs:** x

Figure 2: Non-determinism due to hiding. After hiding signals a and b, the STG becomes non-deterministic, but it can be implemented (the system can simply wait for c, and produce x upon receiving it; input d can be ignored). Note that the two branches after the non-deterministic choice are not entirely symmetric, as the upper one has an input d which is not present in the lower one.

To the best of our knowledge, no satisfactory formal semantics of non-deterministic STGs and in particular for dummy transitions[2] has been given so far (we will show below that the language is not a satisfactory semantics in the non-deterministic case). In this paper we propose (and justify) a formal semantics of non-deterministic STGs. For this, we introduce the concept of *output-determinacy*, which is a relaxation of determinism, and argue that it is reasonable and useful in the speed-independent context; c.f. for example [Mil89] for the concept of determinacy.

As an important application of the developed theory of output-determinacy, we will generalise the decomposition algorithm from [VW02, VK05] and prove its correctness with our theory. We will now discuss how decomposition fits into the design flow for synthesising asynchronous circuits from STGs.

Petrify [CKK+97, CKK+02] is one of the commonly used tools for synthesis of asynchronous circuits from STGs. For synthesis, it employs the state space of the STG, and so suffers from the combinatorial *state space explosion* problem. That is, even a relatively small STG may (and often does) yield a very large state space. This puts practical bounds on the size of control circuits that can be synthesised using such techniques, which are often restrictive, especially if the specification is not constructed manually by a designer but rather generated automatically from high-level hardware descriptions. (For example, designing a control circuit with more than 20–30 signals with Petrify is often impossible.) Hence, this approach does not scale.

To cope with the state space explosion problem, Chu suggested a nondeterministic method for decomposing an STG into several smaller ones [Chu87], see also [KKT93]. The idea is that all components together can be synthesised faster than the original STG while the corresponding circuits perform together in the same way as the circuit directly synthesised from the specification. While there are strong restrictions on the structure and labelling of STGs in [Chu87], the improved decomposition algorithm of Vogler, Wollowski and Kangsah [VW02, VK05] works under – comparatively moderate – restrictions on the labelling only. In these previous papers, the specifications had to be deterministic; here, we generalise this to output-determinate specifications.

[2] In practical STGs, the designers intuitively avoid using dummy transitions in situations where their semantics would be ambiguous. However, such situations do exist, in particular when firing a dummy transition can disable other transitions.

Our approach also allows to make the decomposition algorithm more efficient. Each component is obtained from the original STG by hiding some of the signals in it, and then contracting the corresponding transitions. The success of this algorithm depends on the ability to *securely* (i.e., in a behaviour-preserving way) contract all such transitions. If this is not possible, the algorithm of [VK05] has to *backtrack* and re-introduce some of the signals into the component, even if they are not really needed for implementation. In our new version of the algorithm, one can leave such non-contracted hidden transitions in the component and proceed with synthesis for a component with fewer signals, which was obtained in a shorter time. While previously the components were deterministic and correct by construction, our components can be non-deterministic; to guarantee correctness, they have to be checked for output-determinacy in the end. The correctness proof for our version is essentially just language-based, and might be easier to grasp than the proofs in [VW02, VK05]. Furthermore, it is easier now to prove the validity of the STG-transformations (like transition contraction) forming the heart of the decomposition algorithm; it should now also be easier to find further valid transformations.

Another way to cope with the state space explosion problem is to use *syntax-directed* translation of the specification to a circuit, thus avoiding to build the state space. This is essentially the idea behind Balsa [EB02] and Tangram [Ber93]. This technique, although computationally efficient, often yields circuits with large area and performance overheads compared with synchronous counterparts. This is because the resulting circuits are highly over-encoded, i.e., they contain many unnecessary state-holding elements.

For asynchronous circuits to be competitive, one has somehow to combine the advantages of logic synthesis (high quality of circuits) and syntax-directed translation (guarantee of a solution, efficiency) while compensating for their disadvantages. A natural way of doing this is to apply logic synthesis to the control path extracted from a Balsa specification. This control path can be partitioned into smaller 'lumps' which can be handled by logic synthesis, and the 'lumps' on which it fails (because of either inability to find a solution in the given gate library or exceeding memory or time constraints) are implemented using the syntax-directed translation. The initial experiments conducted in [CC06] showed that this combined approach can half the area devoted to control flow and improve its latency, compared with the traditional syntax-directed translation, as long as the size of 'lumps' which can be confidently handled by logic syntax is sufficiently large.

The design flow advocated in [CC06] is as follows. Given a (potentially large) specification STG, the encoding conflicts are resolved using an integer linear programming (ILP) technique to approximate the state space of an STG. Then the resulting STG (free from encoding conflicts) is decomposed into smaller components in such a way that they are also free from encoding conflicts, as described in [CC03]. (Typically, each component is responsible for producing a single signal.) Then these components are synthesised one-by-one using Petrify. This approach can handle much larger specifications than Petrify alone, but its scalability is still limited since ILP is an NP-complete problem.

With our decomposition algorithm, we follow a more scalable approach, which tries to avoid performing expensive operations (such as resolving encoding conflicts) on the original specification. Observe that our check for output-determinacy is also computationally hard, but it is performed on small components; in contrast, in [CC06] the NP-complete ILP-problems are solved for the full specification. The resulting components in our approach, unlike those in the technique described above, are generally not free from encoding conflicts. If a component has an encoding conflict, it can happen due to one of the following two reasons: (i) this conflict was present already in the original STG; or (ii) this conflict was introduced because some of the signals preventing it in the original STG are not present in the component. The technique described in [Sch06] allows one to check which of these two reasons applies, and in case (ii) to find signals which need to be added to the component to prevent such encoding conflicts. Finally, the remaining encoding conflicts are resolved in each component, and they are synthesised.

The paper is organised as follows: in the next section we introduce the basic concepts of Petri nets and STGs. In Section 3, the new notion of output-determinacy is introduced and justified; we give a list of semantics-preserving transformations, and we analyse the complexity of checking output-determinacy. In the following section, we present our STG-decomposition algorithm and prove its correctness. We close with some first experimental results and a conclusion.

2 Basic Definitions

This section provides the basic notions for Petri nets and STGs, for a more detailed explanation cf. e.g. [CKK+02].

2.1 Petri Nets and STGs

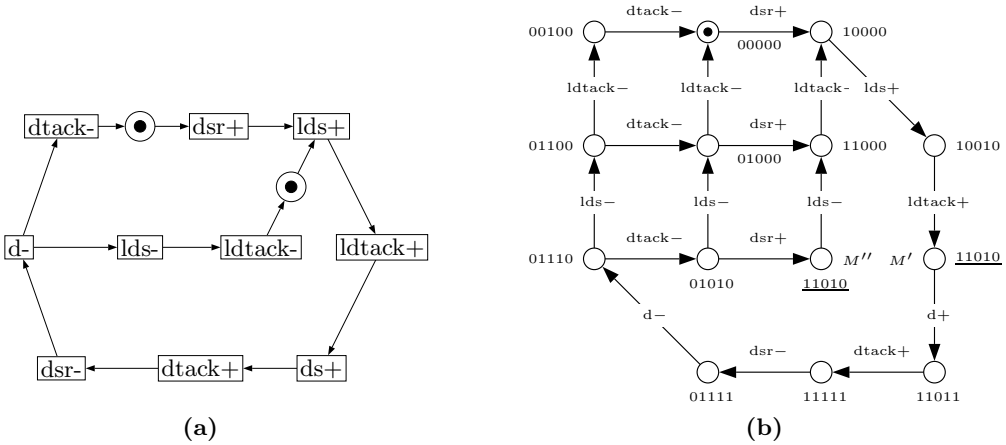
A *Petri net* is a 4-tuple $N = (P, T, W, M_N)$ where P is a finite set of *places* and T a finite set of *transitions* with $P \cap T = \emptyset$. $W : P \times T \cup T \times P \to \mathbb{N}_0$ is the *weight function* and M_N the *initial marking*, where a *marking* is a function $P \to \mathbb{N}_0$ 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. If necessary, we write P_N etc. for the components of N or P' (P_i) etc. for the net N' (N_i) etc. Analogous conventions apply later on.

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 for all $p \in P$ $M'(p) = M(p) - W(p, t) + W(t, p)$.

A transition sequence $v = t_1 \dots t_n$ is *enabled under a marking* M (yielding M') if $M[t_1\rangle M_1[t_2\rangle \dots M_{n-1}[t_n\rangle M_n = M'$, 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.

N is called *bounded* if for every reachable marking M and every place p $M(p) \leq k$ for some constant $k \in \mathbb{N}$; if $k = 1$, N is called *safe*. N is bounded if and only if the set $[M_N\rangle$ of reachable markings is finite. In this paper we only consider bounded Petri nets, STGs resp.



inputs: *dsr*, *ldtack*; **outputs:** *dtack*, *lds*, *d*

Figure 3: An STG modelling a simplified VME bus controller **(a)** and its state graph with a CSC conflict between the underlined states **(b)**. The order of signals in the binary encodings is: *dsr*, *ldtack*, *dtack*, *lds*, *d*.

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 s^+, s^- resp. instead of $(s, +)$, $(s, -)$ 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 $s^\pm$ 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, which do not correspond to any signal change.

An example of an STG is shown in Figure 3(a) (cf. [CKK+02]). Places are drawn as circles containing a number of tokens corresponding to their marking. Unmarked places which have only one transition in their preset, postset resp. are not drawn if the corresponding arcs are weighted with 1; they are implicitly given by an arc between these two transitions. Transitions are drawn as rectangles together with their labelling, the weight function as directed arcs xy (labelled with $W(x, y)$ if $W(x, y) > 1$).

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 – deleting λ-labels automatically since λ is the empty word; i.e. $M[s^\pm\rangle\rangle M'$ means that a sequence of transitions fires, where one of them is labelled $s^\pm$ while the others (if any) are λ-labelled. 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 and denoted by $L(N)$.

An STG is called *consistent* if for each signal s the edges s^+ and s^- alternate in all traces, always beginning with the same signal edge. Only from consistent STGs a circuit can be synthesised.

An STG has a *dynamic conflict* if there are different transitions t_1 and t_2 such that for some reachable marking M: $M[t_1\rangle$ and $M[t_2\rangle$, but $\exists p \in P : M(p) < W(p,t_1) + W(p,t_2)$. A dynamic conflict implies a *structural conflict*, i.e. ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$. The conflict is called an *auto-conflict* if $l(t_1) = l(t_2) \neq \lambda$.

Simulations are a well-known important device for proving language inclusion or equivalence. A *simulation from N_1 to N_2* is a relation $\mathcal{S}$ between markings of N_1 and N_2 such that $(M_{N_1}, M_{N_2}) \in \mathcal{S}$ and for all $(M_1, M_2) \in \mathcal{S}$ and $M_1[t\rangle M_1'$ there is some M_2' with $M_2[l_1(t)\rangle\rangle M_2'$ and $(M_1', M_2') \in \mathcal{S}$. If such a simulation exists, then N_2 can go on simulating all signals of N_1 forever.

Often, nets are considered to have the same behaviour if they are language equivalent. Another, more detailed behaviour equivalence is bisimulation. A relation $\mathcal{B}$ is a *bisimulation* between N_1 and N_2 if it is a simulation from N_1 to N_2 and $\mathcal{B}^{-1}$ is a simulation from N_2 to N_1. If such a bisimulation exists, we call the STGs *bisimilar*; intuitively, the STGs can work side by side such that in each stage each STG can simulate the signals of the other. For deterministic STGs, language equivalence and bisimulation coincide.

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 accepting), and $L(N)$ is the language of this automaton. For an example consider Figure 3(b). 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 $M[s^\pm\rangle\rangle M'$.

If RG_N is not deterministic, one can turn it into a deterministic automaton with accepting states only by well-known methods. (Note: this version of a deterministic automaton is in general not complete.) Thus, the λ-edges of the reachability graph resulting from the λ-transitions are removed by automata-theoretic methods. We call this operation *determinisation* and denote the resulting deterministic finite automaton by $DA(N)$. Observe that automata with accepting states only can be regarded as STGs (with the states as places, the initial state being the only marked place etc.); hence, all definitions for STGs also apply to automata.

In the following definition of *parallel composition* $\|$, we will have to consider the distinction between input and output signals. The idea of parallel composition is that the composed systems run in parallel and synchronise on common signals – corresponding to circuits that are connected on signals with the same name. Since a system controls its outputs, we cannot allow a signal to be an output of more than one component; input signals, on the other hand, can be shared. 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. A composition can also be ill-defined due to what e.g. Ebergen [Ebe92] calls *computation interference*; this is a semantic problem, and we will not consider it here, but later in the definition of correctness.

The parallel composition of STGs N_1 and N_2 is defined if $Out_1 \cap Out_2 = \emptyset$. If we drop this requirement, the definition gives the *synchronous product* $N_1 \times N_2$, which will be technically useful. Let $A = (In_1 \cup Out_1) \cap (In_2 \cup Out_2)$ be the set of common signals. If e.g. s is an output of N_1 and an input of N_2, then an occurrence of an edge $s^\pm$ in N_1 is 'seen' by N_2, i.e. it must be accompanied by an occurrence of $s^\pm$ in N_2. Since we do not know a priori which $s^\pm$-labelled transition of N_2 will occur together with some $s^\pm$-labelled transition of N_1, we have to allow for each possible pairing. Thus, the *parallel composition* $N = N_1 \parallel N_2$ is obtained from the disjoint union of N_1 and N_2 by combining each $s^\pm$-labelled transition t_1 of N_1 with each $s^\pm$-labelled transition t_2 from N_2 if $s \in A$. In the formal definition of parallel composition, $*$ is used as a dummy element, which is formally combined e.g. with those transitions that do not have their label in the synchronisation set A. (We assume that $*$ is not a transition or a place of any net.) Thus, N is defined by

$$
\begin{aligned}
P &= P_1 \times \{*\} \cup \{*\} \times P_2 \\
T &= \{(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, l_1(t_1) = l_2(t_2) \in A\{+,-\}\} \\
&\quad \cup \{(t_1, *) \mid t_1 \in T_1, l_1(t_1) \notin A\{+,-\}\} \\
&\quad \cup \{(*, t_2) \mid t_2 \in T_2, l_2(t_2) \notin A\{+,-\}\}
\end{aligned}
$$

$$
W((p_1,p_2),(t_1,t_2)) = \begin{cases} W_1(p_1,t_1) \text{ if } p_1 \in P_1,\ t_1 \in T_1 \\ \text{or} \\ W_2(p_2,t_2) \text{ if } p_2 \in P_2,\ t_2 \in T_2 \end{cases}
$$

$$
W((t_1,t_2),(p_1,p_2)) = \begin{cases} W_1(t_1,p_1) \text{ if } p_1 \in P_1,\ t_1 \in T_1 \\ \text{or} \\ W_2(t_2,p_2) \text{ if } p_2 \in P_2,\ t_2 \in T_2 \end{cases}
$$

$$
l((t_1,t_2)) = \begin{cases} l_1(t_1) & \text{if } t_1 \in T_1 \\ l_2(t_2) & \text{if } t_2 \in T_2 \end{cases}
$$

$$
M_N = M_{N_1} \dot{\cup} M_{N_2}, \text{ i.e. } M_N((p_1,p_2)) = \begin{cases} M_{N_1}(p_1) & \text{if } p_1 \in P_1 \\ M_{N_2}(p_2) & \text{if } p_2 \in P_2 \end{cases}
$$

$$
\begin{aligned}
In &= (In_1 \cup In_2) - (Out_1 \cup Out_2) \\
Out &= Out_1 \cup Out_2
\end{aligned}
$$

Clearly, one can consider the place set of the composition as the disjoint union of the place sets of the components. Therefore, we can consider markings of the composition (regarded as multisets) as the disjoint union of markings of the components – as exemplified above for M_N, and we will also write a marking $M_1 \dot{\cup} M_2$ of the composition as (M_1, M_2).

Figure 4 shows an example of a parallel composition. For simplicity, a place of the composition is denoted as p instead of $(p, *)$ or $(*, p)$, and the same applies to unsynchronised transitions. For synchronised transitions, we write t_{i_j} instead of (t_i, t_j). To keep the example small, we use only signals as labels instead of signal edges.

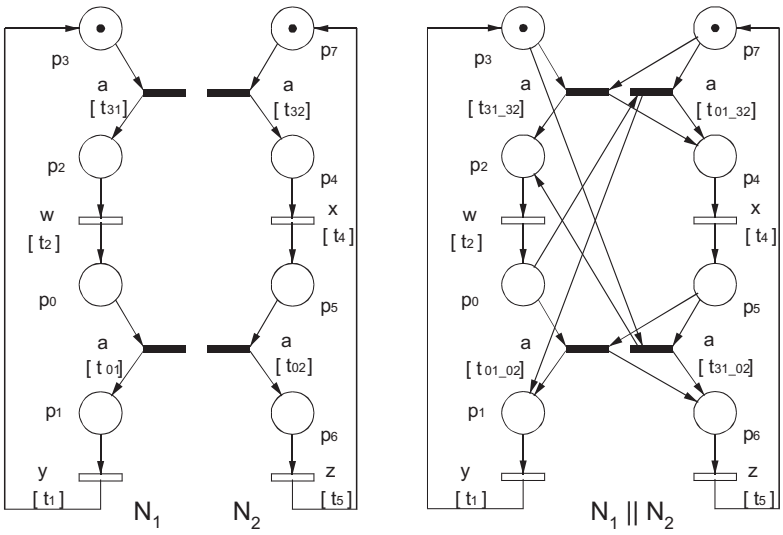


Figure 4: Parallel composition example

In the example, there are two transitions with label a in N_1 – t_{31} and t_{01} – and two in N_2 – t_{32} and t_{02}. Each transition with label a in N_1 should be synchronised with each transition with label a in N_2. Therefore, in $N_1 \parallel N_2$, there are four transitions with label a. Note that though there is a synchronisation between t_{01} and t_{32} (and between t_{31} and t_{02}), the synchronised transition t_{01_32} (and t_{31_02}) in $N_1 \parallel N_2$ will never fire.

By definition of $\parallel$, the firing $(M_1, M_2)[(t_1, t_2)\rangle(M_1', M_2')$ of N corresponds to the firings $M_i[t_i\rangle M_i'$ in N_i, $i = 1, 2$; here, the firing of $*$ means that the empty transition sequence fires. Therefore, all reachable markings of N have the form (M_1, M_2), where M_i is a reachable marking of N_i, $i = 1, 2$. This can easily be checked for the example in Figure 4.

If the components do not have internal transitions, then also their composition has none; it is easy to see that N is deterministic if N_1 and N_2 are. But note that N might have structural auto-conflicts even if none of the N_i has; cf. Figure 4.

It should be clear that, up to isomorphism, composition is associative and commutative. Therefore, we can define the parallel composition of a finite family (or collection) $(C_i)_{i\in I}$ of STGs as $\|_{i\in I}\ C_i$, provided that no signal is an output signal of more than one of the C_i. We will also denote the markings of such a composition by $(M_1, \ldots, M_n)$ if M_i is a marking of C_i for $i \in I = \{1, ..., n\}$.

We now introduce transition contraction (see e.g. [And83] for an early reference), which will be most important in our decomposition procedure. We essentially repeat from [VK05], where further discussions can be found.

Definition 2.1 (Transition Contraction)
Let N be an STG and $t \in T$ with $l(t) = \lambda$, ${}^\bullet t \cap t^\bullet = \emptyset$ and $W(p,t), W(t,p) \leq 1$ for all $p \in P$. We define the *t-contraction* $\overline{N}$ of N by

$$\begin{aligned}
\overline{P} &= \{(p,*) \mid p \in P - ({}^\bullet t \cup t^\bullet)\} \\
&\quad \cup\ \{(p,p') \mid p \in {}^\bullet t, p' \in t^\bullet\} \\
\overline{T} &= T - \{t\}
\end{aligned}$$

$$\begin{aligned}
\overline{W}((p,p'),t_1) &= W(p,t_1) + W(p',t_1) \\
\overline{W}(t_1,(p,p')) &= W(t_1,p) + W(t_1,p') \\
\overline{l} &= l\big|_{\overline{T}} \\
M_{\overline{N}}((p,p')) &= M_N(p) + M_N(p') \\
\overline{In} = In & \qquad \overline{Out} = Out
\end{aligned}$$

In this definition, $* \notin P \cup T$ is a dummy element; we assume $W(*,t_1) = W(t_1,*) = M_N(*) = 0$.
We say that the markings M of N and $\overline{M}$ of $\overline{N}$ satisfy the *marking equality* if for all $(p,p') \in \overline{P}$

$$\overline{M}((p,p')) = M(p) + M(p').$$

For two different transitions t_1, t_2 with $t_1 \neq t \neq t_2$, we call the unordered pair $\{t_1, t_2\}$ a *new conflict pair* whenever ${}^\bullet t \cap {}^\bullet t_1 \neq \emptyset$ and $t^\bullet \cap {}^\bullet t_2 \neq \emptyset$ in N (or vice versa); if $l(t_1) = l(t_2) \neq \lambda$, we speak of a *new structural auto-conflict*.

A transition contraction is called *secure* if either $({}^\bullet t)^\bullet \subseteq \{t\}$ (*type-1 secure*) or ${}^\bullet(t^\bullet) = \{t\}$ and $M_0(p) = 0$ for some $p \in t^\bullet$ (*type-2 secure*).

Note that, in general, $\overline{N}$ might fail to be consistent (see below), even if N is; but secure contractions preserve consistency [VK05].

Figure 5 (a) shows a part of a net and the result when the λ-transition is contracted. In many cases, the preset or the postset of the contracted transition has only one element, and then the result of the contraction looks much easier as e.g. in Figure 5 (b). 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.

The following theorem of [VK05] and the succeeding corollary of its second part are relevant in the following:

Theorem 2.2
Let $\overline{N}$ be a secure contraction of N.

1. *If the contraction is of type 1, then N and $\overline{N}$ are bisimilar.*
2. *If the contraction is of type 2, then there is a simulation $\mathcal{S}$ from $\overline{N}$ to N such that $\mathcal{S}^{-1}$ is contained in the simulation $\mathcal{B} = \{(M, \overline{M}) \mid M$ and $\overline{M}$ satisfy the marking equality$\}$ from N to $\overline{N}$.*
3. *N and $\overline{N}$ are language equivalent, hence consistency-preserving.*

Corollary 2.3
If $\overline{N}$ is a type-2 secure contraction of N, then the simulation of $\mathcal{S}$ in Theorem 2.2.2 is a ready simulation *from $\overline{N}$ to N, i.e. a simulation where $(\overline{M}, M) \in \mathcal{S}$ implies $\overline{M}[s^\pm\rangle\rangle$ if and only if $M[s^\pm\rangle\rangle$ for all signals s.*

We conclude this section by defining redundant transitions and places; the deletion of such a transition, place resp., (including the incident arcs) is another operation that can be used in our decomposition algorithm.

A *redundant transition* is a λ-transition t, where either each place $p \in {}^\bullet t \cup t^\bullet$ forms a loop with t with two arcs of the same weight (t is a *loop-only transition*) or some other λ-transition has arcs to and from the same

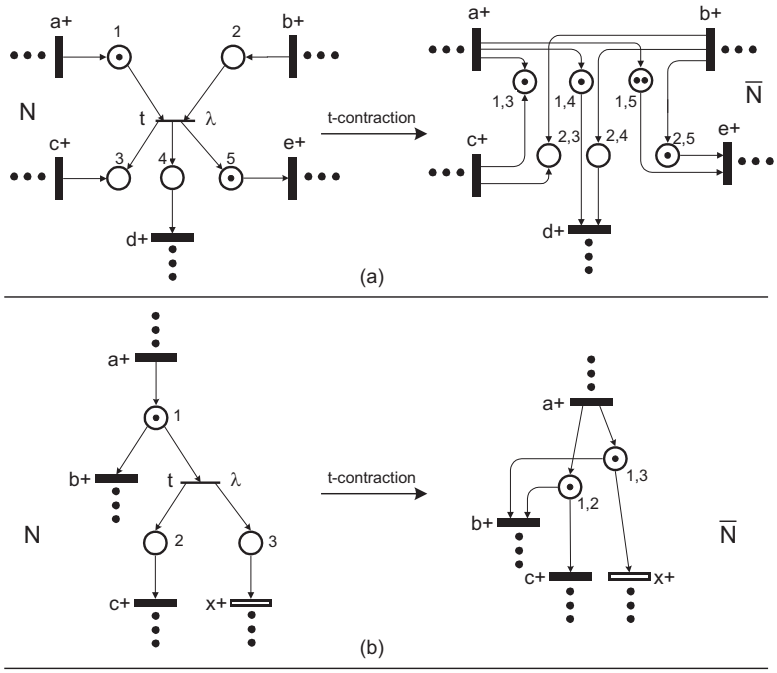


Figure 5:

places with the same weight as t (which is a *duplicate transition*). In this paper, we also speak of a duplicate and redundant transition if the two transitions in question are labelled with the same signal edge.

A place p of an STG N is (structurally) *redundant* (see e.g. [Ber87]) if there is a set of places Q with $p \notin Q$, a valuation $V : Q \cup \{p\} \to \mathbb{N}$ and some $c \in \mathbb{N}_0$ which satisfy the following properties for all transitions t:

- $V(p)M_N(p) - \sum_{q \in Q} V(q)M_N(q) = c$
- $V(p)(W(t,p) - W(p,t)) - \sum_{q \in Q} V(q)(W(t,q) - W(q,t)) \geq 0$
- $V(p)W(p,t) - \sum_{q \in Q} V(q)W(q,t) \leq c$

The first two items ensure that p is something like a linear combination of the places in Q with factors $V(q)/V(p)$. Indeed, for the case $c = 0$, the first item says that p is such a combination initially; the second item, in the case of equality, says that this relationship is preserved when firing any transition. The proof that p is indeed redundant argues that the valuated token number of p is at least c larger than the valuated token sum on Q for all reachable markings, while the third item says that each transition or at least each output transition needs at most c 'valuated tokens' more from p than from the places in Q; this shows that for the enabling of a transition the presence or absence of p does not matter.

Proposition 2.4
If N' is obtained from an STG N by deleting a redundant transition or place, then N and N' are bisimilar.

2.2 STGs and Asynchronous Circuits

STGs are widely used for specifying the behaviour of *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. We now explain the important concept of *complete state coding (CSC)*.

For an STG N, a *state vector* is a function $sv : Sig \to \{0, 1\}$ where '0' means logical low and '1' logical high. A *state assignment* assigns a state vector to each marking M of RG_N denoted by sv_M.

A state assignment must satisfy for every signal $x \in Sig$ and every pair of markings $M, M' \in [M_N\rangle$:

$$M[x+\rangle\rangle M' \text{ implies } sv_M(x) = 0, sv_{M'}(x) = 1$$
$$M[x-\rangle\rangle M' \text{ implies } sv_M(x) = 1, sv_{M'}(x) = 0$$
$$M[y^{\pm}\rangle\rangle M' \text{ for } y \neq x \text{ implies } sv_M(x) = sv_{M'}(x)$$
$$M[\lambda\rangle\rangle M' \text{ implies } sv_M = sv_{M'}$$

If such an assignment exists, it is uniquely defined by these properties[3], and the reachability graph and the underlying STG are consistent. From an *inconsistent* STG, one cannot synthesise a circuit. Figure 3(b) shows the reachability graph of the STG in Figure 3(a); every marking is annotated with its state vector.

If there is a state assignment, N has *Complete State Coding (CSC)* if any two reachable markings M_1 and M_2 with the same state vector (i.e. $sv_{M_1} = sv_{M_2}$,) enable the same output signals. Otherwise, N has a *CSC conflict*, cf. e.g. Figure 3(b), and no circuit can be synthesised directly.

If CSC is violated, one tries to achieve it by the insertion of *internal signals*, i.e. outputs which are considered to be unknown to the environment, without changing the external behaviour of the STG.

3 Output-Determinacy

In this section, we define in a natural way when a deterministic STG can be regarded as a correct implementation of a specification STG N; we only consider deterministic implementations here, since the final implementation of N will be a circuit, which is deterministic by nature. Considering the case that N is non-deterministic, we introduce the concept of *output-determinacy*, which is a relaxation of determinism. It turns out that output-determinate STGs are exactly the STGs which have correct implementations according to our notion. Hence, non-output-determinate STGs are ill-formed (in particular, they cannot be correctly implemented by a circuit). This shows that the language is not a satisfactory semantics of non-deterministic STGs in general; in particular, synthesising the determinised state graph of a non-output-determinate STG may either fail or result in an incorrect circuit.

For the class of output-determinate STGs we show that their language is an adequate semantics, and re-formulate the notion of correct implementation purely in terms of the language; this notion will play an important role as part of the invariant in the proof of correctness of our STG decomposition algorithm described in Section 4, which we view as an important application of the developed theory. Moreover, we introduce a set of semantics-preserving STG transformations, which are, in particular, used in our decomposition algorithm. This set can easily be extended since the definition of semantics-preserving is simple.

Finally, we analyse the computational complexity of checking whether a given STG is output-determinate for several classes of STGs, and describe a practical way of checking it in the case of a divergence-free safe or bounded STG.

3.1 Correct implementations

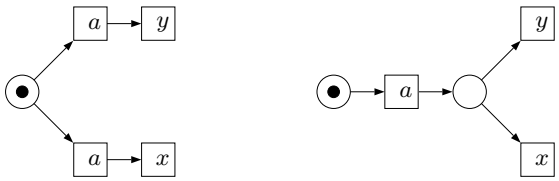
An STG N specifies the behaviour of a system in the sense that the system must provide *all and only* the specified outputs and that it must allow *at least* the specified inputs. As a consequence, the system must be able to perform at least all traces of N. In fact, N also describes assumptions about the environment the system will interact with; namely, the environment will only produce the inputs specified by N. A correct implementation of N may allow additional inputs, but these inputs and subsequent behaviour will never occur in the envisaged environment. In other words, when the system is running in a proper environment, only traces of N can occur.

The implementation may actually have fewer input signals than N, keeping only those that are relevant for producing the required outputs. In this case, the environment may provide irrelevant inputs, but the implementation simply ignores them — and in this sense, they are always allowed (e.g., in the STG in Fig. 2, inputs a, b and d are irrelevant for producing x and can be ignored).

The following definition assumes a deterministic implementation (as it is the case in circuit design), but the specification can be non-deterministic. The projection of a trace w of N onto the signals of C, obtained by deleting all signal edges where the signal belongs to $In_N \setminus In_C$, is denoted by $w\big|_C$.

Definition 3.1 (Correct Implementation)
A deterministic STG C is a *correct implementation* of an STG N if $In_C \subseteq In_N$, $Out_C = Out_N$, and for all w and all M such that $M_N[w\rangle\rangle M$ the following hold:

[3]At least for every signal $s \in Sig$ which actually occurs, i.e. $M[s^{\pm}\rangle\rangle$ for some reachable marking M.



(a) **(b)**

inputs: a; **outputs:** x, y

Figure 6: Non-semi-modularity due to determinisation. A semi-modular but not output-determinate STG **(a)** and the non-semi-modular STG (due to the choice between the outputs x and y) obtained from it by determinisation **(b)**. Note that determinisation can also result in a choice between an input and an output (this would be the case if y were an input).

(C1) $w\big|_C$ is a trace of C, i.e., $M_C[w\big|_C\rangle\rangle M'$ for some marking M' of C (note that M' is unique as C is deterministic);

(C2) If $a \in In_N$ and $M[a^{\pm}\rangle\rangle$, then either $M'[a^{\pm}\rangle\rangle$ or $a \notin In_C$;

(C3) If $x \in Out_N$, then $M[x^{\pm}\rangle\rangle$ iff $M'[x^{\pm}\rangle\rangle$. ◇

This definition is a formalisation of the considerations above: the implementation must be able to perform all traces of the specification, maybe dropping some irrelevant input signals (C1); all the inputs allowed by the specification must be allowed (or ignored) by the implementation (C2); and the implementation must produce exactly the specified outputs (C3). In particular, every deterministic STG N is a correct implementation of itself.

3.2 The notion of output-determinacy

A non-deterministic specification can perform the same trace in two different ways, reaching different states M_1 and M_2. In the speed-independent context the only information available to the circuit is the execution history, i.e., the trace performed,[4] and so an implementation cannot know whether its current state corresponds to M_1 or M_2. Hence, a deterministic implementation must behave consistently with the specification *no matter in which of these markings it is*.

Our definition of correctness requires that the implementation must provide *exactly* the outputs enabled by M_1 and exactly the outputs enabled by M_2. This is only possible if M_1 and M_2 enable the same outputs. In contrast, the implementation must allow *at least* the inputs enabled under M_1 and the inputs enabled under M_2; this is very well possible, even if these sets of inputs differ – i.e. the implementation may allow the union of these sets or any of its supersets. This observation leads to our central notion of output-determinacy.

Definition 3.2 (Output-Determinacy)
An STG N is called *output-determinate* if $M_N[w\rangle\rangle M_1$ and $M_N[w\rangle\rangle M_2$ implies for every $x \in Out_N$ that $M_1[x^{\pm}\rangle\rangle$ iff $M_2[x^{\pm}\rangle\rangle$. ◇

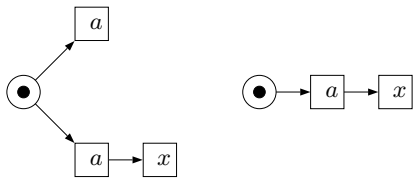
For example, the STG in Fig. 2 is output-determinate after hiding a and b. Clearly, a deterministic STG is also output-determinate; note also that – in contrast to a deterministic STG – an output-determinate STG may contain λ-transitions.

3.3 Semantics of non-deterministic specifications

Now we demonstrate that the notion of output-determinacy is useful for defining a semantics of non-deterministic specifications (in particular, allowing λ-transitions), and we also justify this semantics.

First of all, the naïve approach consisting in determinisation of a non-deterministic specification using the usual procedure for finite state automata and then proceeding with the synthesis is not always correct. In the context of STGs and circuit synthesis, the result of determinisation can manifest some problems, e.g., non-semi-modularity, as illustrated in Fig. 6; Fig. 7 illustrates a much more dangerous scenario, where the determinised STG contains no apparent problems but the resulting circuit is incorrect according to Definition 3.1. In both

[4] In a non-speed-independent context some additional information such as timing of events may help to resolve non-determinism.



(a) **(b)**

inputs: a; **outputs:** x

Figure 7: Incorrect determinisation: a non-output-determinate STG before **(a)** and after **(b)** determinisation. The latter STG, though implementable, is not a correct implementation of the original specification, since it can cause a failure in the environment by producing x when the environment does not expect it.

cases, it is wiser to inform the designer of an error than to determinise and synthesise such a specification. Below we show that determinisation can be safe only for output-determinate specifications.

> **Semantic Rule 1.** A non-output-determinate specification of a speed-independent system cannot be implemented deterministically and thus is ill-formed.

This rule can be justified by the following result.

Proposition 3.3
Let C be a correct implementation of N; in particular, C is deterministic. Then N is output-determinate.

Proof. For the sake of contradiction, suppose that N has a trace w and two reachable markings, M_1 and M_2, such that for some $x \in Out_N$, $M_N[w\rangle\rangle M_1[x^{\pm}\rangle\rangle$, and $M_N[w\rangle\rangle M_2$ and $\neg M_2[x^{\pm}\rangle\rangle$. Then, by (C1) of Definition 3.1, $w|_C$ is a trace of C; moreover, since C is deterministic, it has a unique reachable marking M' such that $M_C[w|_C\rangle\rangle M'$. Now, by (C3) of Definition 3.1, $M'[x^{\pm}\rangle\rangle$ due to $M_1[x^{\pm}\rangle\rangle$, and, on the other hand, $\neg M'[x^{\pm}\rangle\rangle$ due to $\neg M_2[x^{\pm}\rangle\rangle$, a contradiction. □

Observe that a non-output-determinate STG always has CSC conflicts, as, according to Definition 3.2, any violation of output-determinacy implies the presence of two states which can be reached by the same trace (and thus have the same encoding) and enable different sets of outputs. It can be shown that such a CSC conflict is *irreducible*, i.e. it cannot be resolved by the insertion of *internal signals* into the STG (as performed e.g., by Petrify or Mpsat) in such a way that its 'external' behaviour does not change. These new *internal* signals can be treated as outputs which are ignored by the environment. The STG resulting from such an insertion will always have a violation of output-determinacy (and thus CSC conflicts) again. Further explanations and a proof can be found in Appendix A.

On the other hand, output-determinate specifications can safely be determinised, and so there is no reason to distinguish between the specification itself and its determinised form:

> **Semantic Rule 2.** The semantics of an output-determinate specification of a speed-independent system is its (prefix-closed) language.

This rule can be justified by the following result.

Proposition 3.4
Let N be output-determinate and C be the deterministic automaton $DA(N)$ obtained by determinisation of the reachability graph of N. Then C is a correct implementation of N.

Proof. The determinisation does not change the language; hence, $M_N[w\rangle\rangle M[s^{\pm}\rangle\rangle$ ($w \in (Sig_N^{\pm})^*, s \in Sig_N$) implies directly $M_C[w\rangle\rangle M'[s^{\pm}\rangle\rangle$. This proves (C1), (C2) and the '$\Rightarrow$' part of (C3).

To show the '$\Leftarrow$' part, assume $M'[x^{\pm}\rangle\rangle$ ($x \in Out_N$). This implies $M_N[w\rangle\rangle M''[x^{\pm}\rangle\rangle$ for some marking M'', otherwise the language is not preserved. Since N is output-determinate, also $M[x^{\pm}\rangle\rangle$. □

The proposed semantics has interesting consequences, in particular, a deadlock-free specification can be equivalent to one with deadlocks, as illustrated in Fig. 8. Hence, arbitrary language-preserving transformations of output-determinate specifications are allowed, as long as the resulting STG is still output-determinate. That is, there is no need to preserve stronger equivalences such as bisimulation. We discuss valid transformations in Section 3.4.

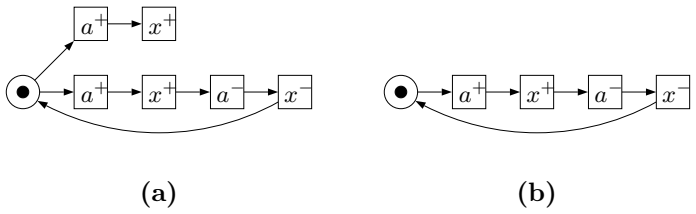


Figure 8: Determinisation: an output-determinate STG N with a deadlock **(a)** and the deadlock-free STG obtained from N by determinisation **(b)**. The latter STG is a correct implementation of N; intuitively, the execution of x^- is correct, since it only occurs when the environment signalled with a^- that the system is in the 'lower' branch of N. The circuit $[x] = a$ implements either of these two STGs.

In view of Semantic Rule 2, one would expect that the notion of correct implementation given in Definition 3.1 can be re-formulated purely in terms of the language if the specification and the implementation are known to be output-determinate. In fact, we generalise the definition to allow a non-deterministic implementation, as long as it is output-determinate.

Definition 3.5 (Trace-Correct Implementation)
An output-determinate STG C is a *trace-correct implementation* of an output-determinate STG N if $In_C \subseteq In_N$, $Out_C = Out_N$, and for every trace w of N the following hold:

(TC1) $w\big|_C$ is a trace of C;

(TC2) If $w\big|_C x^{\pm}$ is a trace of C for some $x \in Out_C$, then $wx^{\pm}$ is a trace of N. ◇

This definition can be viewed as a *denotational* notion of correctness, as opposed to the *operational* one given in Definition 3.1. However, it should be emphasised that this notion explicitly requires the specification to be output-determinate (i.e., this purely trace-based view is unable to distinguish between output-determinate and non-output-determinate specifications). The result below shows that this notion is equivalent to Definition 3.1 if the implementation is deterministic and the specification is output-determinate.

Proposition 3.6 (Justification of the notion of trace-correct implementation)
Let N be an output-determinate STG and C be a deterministic STG such that $In_C \subseteq In_N$ and $Out_C = Out_N$. Then C is a correct implementation of N iff it is a trace-correct implementation of N.

We postpone the proof of this result until the next section, where it is formulated and proven for the more general case of a distributed implementation $C = \|_{i \in I} C_i$. (Note that C in the above result can be seen as being a distributed implementation comprised of a single component.)

3.4 Valid STG transformations

Due to Semantic Rule 2, any language-preserving STG transformation of an output-determinate specification is valid, as long as the resulting STG is output-determinate. However, it is desirable for a transformation to preserve non-output-determinacy as well, so that an ill-formed STG does not become well-formed after its application; that is, *a transformation should propagate errors rather than eliminate them, so that they can eventually be detected.* This motivates the following notion.

Definition 3.7 (LOD-equivalence and LOD-transformations)
Two STGs N and N' are *LOD-equivalent*, denoted $N \approx_{lod} N'$, if

- N and N' are both non-output-determinate, or
- N and N' are language-equivalent and both output-determinate.

An STG transformation is an *LOD-transformation* if the original and the transformed STG are LOD-equivalent. ◇

One can observe that any transformation yielding a bisimilar STG is a LOD-transformation, but there are LOD-transformations which yield a non-bisimilar STG, e.g., determinisation of an output-determinate STG, as illustrated in Fig. 9. Moreover, any transformation preserving the language and output-determinacy can be

inputs: a, b, c

Figure 9: Two LOD-equivalent STGs which are not bisimilar.

made into an LOD-transformation if its domain is restricted to output-determinate systems. Below we list some LOD-transformations which will be useful for our decomposition algorithm.

For one of the transformations and for further use, we first introduce some notions.

Definition 3.8
For transitions t, t' of some STG, t is a (syntactic) *trigger of* t' or *triggers* t' if $t^\bullet \cap {}^\bullet t' \neq \emptyset$. A λ-transition t is a *weak trigger* of t', if it triggers t' or another weak trigger of t'. A transition t with $l(t) \neq \lambda$ is a *signal trigger* of t', if it triggers t' or a weak trigger of t'.

A transition t is in a *weak syntactic conflict* with t', if it is in syntactic conflict with t' or with a weak trigger of t'.

List of LOD-transformations

RedPD Deletion of a redundant place.

RedTD Deletion of a redundant transition.

SecTC1 Type-1 secure contraction of a λ-transition.

LOD-SecTC2 Type-2 secure contractions of λ-transitions restricted to output-determinate STGs.

SecTC2' Type-2 secure contractions of λ-transitions which are not in weak syntactic conflict with an output transition.

The first three transformations in this list always yield a bisimilar STG and thus are LOD-transformations. Below we prove that the remaining two transformations are also LOD-transformations.

Theorem 3.9
If $\overline{N}$ is obtained from some STG N by LOD-SecTC2 or SecTC2', then N and $\overline{N}$ are LOD-equivalent.

Proof. A secure contraction gives a language-equivalent result in any case by Proposition 2.2.

Now we consider an output-determinate N and show that $\overline{N}$ is also output-determinate. If $M_{\overline{N}}[w\rangle\rangle\overline{M}_1[x^\pm\rangle\rangle$ and $M_{\overline{N}}[w\rangle\rangle\overline{M}_2$ ($w \in (Sig^\pm)^*, x \in Out$), then $M_N[w\rangle\rangle M_1$ and $M_N[w\rangle\rangle M_2$ with $(\overline{M}_1, M_1), (\overline{M}_2, M_2) \in \mathcal{S}$ for the ready simulation $\mathcal{S}$ of Corollary 2.3. Furthermore, $M_1[x^\pm\rangle\rangle$ due to simulation, $M_2[x^\pm\rangle\rangle$ due to output-determinacy, and $\overline{M}_2[x^\pm\rangle\rangle$ due to ready simulation.

This settles the case of LOD-SecTC2, and for SecTC2' (applied to transition t) it only remains to show that N is output-determinate if $\overline{N}$ is; so assume the latter.

Consider firing sequences u, v of N such that $l(u) = l(v)$, $M_N[u\rangle[x^\pm\rangle\rangle$ and $M_N[v\rangle M_1$. We will now apply the simulation $\mathcal{B}$ of Theorem 2.2.2; to get a result on the level of transitions, observe that this relation also is a simulation if the labelling of N is λ for t and the identity otherwise. This consideration implies that e.g. u is simulated by $u\big|_{-t}$, obtained by deleting all occurrences of t in u. Thus, we get $M_{\overline{N}}[u\big|_{-t}\rangle[x^\pm\rangle\rangle$ and $M_{\overline{N}}[v\big|_{-t}\rangle\overline{M}_1$.

Since $\overline{N}$ is output-determinate and $\overline{l}(u\big|_{-t}) = \overline{l}(v\big|_{-t})$, we have $\overline{M}_1[x^\pm\rangle\rangle$. Therefore, we can take some $t' \in \overline{T}$ and a minimal $w \in \overline{T}^*$ such that $\overline{M}_1[wt'\rangle\overline{M}_2$, $\overline{l}(t') = x^\pm$ and $\overline{l}(w) = \lambda$. By minimality, each transition in w triggers a transition in wt'; hence each transition in w is a weak trigger of the output transition t' and $(*)$ does not share a preset-place with t by assumption of SectTC2'; neither does t'.

We conclude the proof by showing inductively that $M_1[w'\rangle M_2$ with $w'\big|_{-t} = wt'$ for some suitable M_2. As induction base, we have $M_1[\lambda\rangle M_1$ and $\overline{M}_1[\lambda\rangle\overline{M}_1$. So assume $M_1[w''\rangle M$ and $\overline{M}_1[w''\big|_{-t}\rangle\overline{M}$ for some prefix $w''\big|_{-t}$ of wt', and let t_1 be the next transition of wt'. If $M[t_1\rangle M'$ for some M', then $\overline{M}[t_1\rangle\overline{M'}$ due to the simulation $\mathcal{B}$.

It remains to consider the case that $\neg M[t_1\rangle$. We observe that $\overline{M}[t_1\rangle$ (i.e. in particular $\overline{M}((p,p')) \geq \overline{W}((p,p'),t_1)$ for all $p \in {}^\bullet t$ and $p' \in t^\bullet$), that M and $\overline{M}$ coincide on the places not adjacent to t, and that

t_1 and t do not share a preset-place by $(*)$. Thus, the only reason for $\neg M[t_1\rangle$ is that for some $p_0 \in t^\bullet$ we have $W(p_0, t_1) > M(p_0)$.

We choose $p_1 \in t^\bullet$ such that $m_1 = W(p_1, t_1) - M(p_1)$ is maximal; m_1 is not negative due to p_0. We check that t can fire m_1 times under M: for all $p \in {}^\bullet t$, we have $M(p) + M(p_1) = \overline{M}((p, p_1)) \geq \overline{W}((p, p_1), t_1) = W(p, t_1) + W(p_1, t_1)$, and thus $M(p) \geq W(p_1, t_1) - M(p_1) + W(p, t_1) \geq m_1$; recall that t has only arcs of weight 1. Firing t under M m_1 times gives a marking M'', which satisfies the marking equality with $\overline{M}$ by Theorem 2.2.2. By our above considerations and choice of p_1, M'' enables t_1; recall that t_1 is only disabled because of some missing tokens in $t^\bullet$ – and even the largest of these deficits has been compensated in M''. Thus, $M[t^{m_1} t_1\rangle M'$ and again $\overline{M}[t_1\rangle\overline{M'}$ due to the simulation $\mathcal{B}$. □

Finally, we note that also the determinisation of an *output-determinate* STG N can be seen as an LOD-transformation. If N is output-determinate, then constructing $DA(N)$ gives a language equivalent STG, which is not only output-determinate, but even deterministic. The same is true if one additionally minimises the deterministic automaton.

3.5 Checking output-determinacy

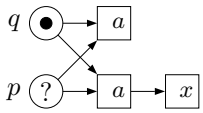
In this section, we analyse the complexity of checking output-determinacy for several classes of STGs and propose a practical test for the cases of safe or bounded divergence-free STGs.

The *coverability* problem is the problem of deciding whether a given Petri net has a reachable marking M *covering* a given marking M' (i.e., $M \geq M'$). The complementary problem will be called the *uncoverability* problem. A special case of the (un)coverability problem is the *single-place (un)coverability,* where $|M'| = 1$. The computational complexity of these problems for various Petri net classes is well-understood [Esp98].

The following reduction from the single-place uncoverability problem to checking output-determinacy forms the basis of our lower complexity bounds analysis.

Proposition 3.10
Let N be a safe/bounded/unbounded Petri net and p be one of its places. Then one can build a, respectively, safe/bounded/unbounded and consistent STG N' whose size is linear in the size of N and which is output-determinate iff p is uncoverable.

Proof. The STG N' is obtained from N by treating all its transitions as λ-labelled and attaching the following net fragment to p, where a is a new input and x is a new output:



Note that the obtained STG is consistent (since the new place q prevents the new transitions from firing more than once) and belongs to the same class (safe/bounded/unbounded) as N. Moreover, N' is output-determinate iff p is uncoverable. □

Corollary 3.11
The problem of checking output-determinacy is $\mathcal{PSPACE}$-hard for safe and bounded consistent STGs and $\mathcal{EXPSPACE}$-hard for unbounded consistent STGs.

Proof. Follows from Proposition 3.10, the corresponding complexity results for the single-place coverability problem in [Esp98] and the fact that the space classes are closed w.r.t. complementation. □

We complete our analysis by giving tight upper bounds for the problem of checking the output-determinacy for the cases of safe and bounded STGs. The basis for this is the following non-deterministic algorithm for checking whether the net is non-output-determinate.

/* execute a sequence σ of transitions without remembering it, */
/* non-deterministically choosing each of its steps */
choose σ **such that** $M_{N\times N}[\sigma\rangle(M_1, M_2)$

/* non-deterministically choose an output transition */
if $\forall t \in T_{N\times N} : l(t) \notin Out$ **then loop forever**
choose $t \in T_{N\times N}$ **such that** $l(t) \in Out$

/* check the non-output-determinacy enabledness condition */
if $\neg M_1[t\rangle$ **then loop forever**
if $\neg M_2[l(t)\rangle\rangle$ **then accept else loop forever**

Given an STG N, the algorithm builds the synchronous product $N \times N$ and analyses its reachable markings. One can observe that in order to show that N is non-output-determinate it is enough to demonstrate the existence of a reachable marking (M_1, M_2) of $N \times N$ such that $M_1[x^{\pm}\rangle\rangle \wedge \neg M_2[x^{\pm}\rangle\rangle$, for some output x. In fact, this condition can be simplified, without loss of generality, replacing $M_1[x^{\pm}\rangle\rangle$ by $\exists t:\ M_1[t\rangle \wedge l(t) = x^{\pm}$. Using this observation, one can easily show the correctness of the above algorithm. Indeed, if it accepts N then N is non-output-determinate; moreover, every non-output-determinate STG N can be accepted if the algorithm makes the proper sequence of choices (exploiting the power of non-determinism).

Note that for safe and bounded STGs, the memory requirement of this algorithm is only polynomial in the size of N; in particular, one can decide whether $M_2[l(t)\rangle\rangle$ at the last step of the algorithm by performing a number of marking coverability tests linear in the size of N (one for each $l(t)$-labelled transition), where each test can be decided in $\mathcal{PSPACE}$ for safe and bounded STGs [Esp98]). Since the deterministic and non-deterministic versions of $\mathcal{PSPACE}$ coincide and the space classes are closed w.r.t. complementation, the following holds.

Proposition 3.12
Output-determinacy can be decided in $\mathcal{PSPACE}$ for safe and bounded STGs.

Combined with Corollary 3.11, this result means that the problem of checking output-determinacy is $\mathcal{PSPACE}$-complete for safe and bounded STGs, and the complexity remains the same if the STG is known to be consistent. However, the algorithm above cannot be used to claim that output-determinacy can be decided in $\mathcal{EXPSPACE}$ for unbounded STGs, even though the property $\neg M_2[l(t)\rangle\rangle$ at the last step of the algorithm can be decided in $\mathcal{EXPSPACE}$ in this case. The reason is that the amount of memory consumed by the algorithm can become arbitrarily large due to the need to keep the current marking of $N \times N$, whose size is unbounded. Hence, in this paper we leave the question about the upper complexity bound for the case of unbounded STGs open. (This case is not very interesting from the practical point of view anyway.)

Though the above algorithm is adequate for proving the theoretical upper complexity bounds, it may be non-trivial to efficiently implement it in practice. Therefore, we propose a much simpler approach for the practically important case of a *divergence-free* STG, i.e., an STG which cannot execute an infinite sequence of λ-transitions from any of its reachable markings.[5] One can observe that in such a case the condition $M_1[x^{\pm}\rangle\rangle \wedge \neg M_2[x^{\pm}\rangle\rangle$ can be simplified further to $(\exists t:\ M_1[t\rangle \wedge l(t) = x^{\pm}) \wedge \neg(\exists t:\ M_2[t\rangle \wedge l(t) \in \{x^{\pm}, \lambda\})$. The latter can be reduced to a number of coverability tests (by introducing complimentary places) that is polynomial in the size of N, or checked directly using, e.g., the unfoldings-based theory developed in [Kho03, Mel98].

4 Decomposition into Output-Determinate Components

In this section, we describe how the developed theory of output-determinacy can be applied to derive an algorithm for decomposition of STGs into smaller components. First, we consider *distributed implementations,* i.e., implementations which can be represented as a parallel composition of STGs, and derive a correctness condition for such implementations, which is consistent with the ones developed in the previous section. Then we describe our decomposition algorithm and formally prove its correctness.

4.1 Correct Decompositions

In this section, implementations consisting of a family of *components* $(C_i)_{i\in I}$ are considered. Recall that we assume all STGs to be bounded; this is preserved by all LOD-transformations described in this paper. For each of the C_i, synthesis is performed separately and the resulting circuits are simply connected with wires for their common signals. Clearly, an output must be produced by only one component. On the other hand, several components can listen to the same signal, produced by the environment or another component. On the level of STGs, this is captured by the *parallel composition* of the $(C_i)_{i\in I}$. We first specialise Definition 3.1 to families of components, additionally taking care of *computation interference* as explained below.

Definition 4.1 (Correct Decomposition)
Let N be an STG and $C \stackrel{\text{df}}{=} \|_{i\in I} C_i$ be a parallel composition of deterministic components. Then $(C_i)_{i\in I}$ is a *correct distributed implementation* of N, if C is a correct implementation of N (cf. Definition 3.1) and the following holds:

(C4) If w is a trace of N, $M_C[w\big|_C\rangle\rangle(M_i)_{i\in I}$ for some marking $(M_i)_{i\in I}$ of C, and $M_j[x^{\pm}\rangle\rangle$ for some $j \in I$ and $x \in Out_j$, then $(M_i)_{i\in I}[x^{\pm}\rangle\rangle$ (no *computation interference*).

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

[5] A practical sufficient condition for divergence-freeness can be obtained using T-invariants [Mur89].

If some component produces an output which is not expected by the other components, in reality this output is produced anyway – leading to a malfunction of the system. But on the level of STGs, in the parallel composition of the components, this output will be disabled instead; hence, (C4) forbids such unexpected outputs.

Since computation interference is a semantical notion, we have not considered it in the definition of parallel composition, where we only required the syntactic condition that the output sets are disjoint. More precisely, computation interference is only forbidden in states that can really occur in appropriate environments, i.e., when performing a trace of N (modulo the irrelevant inputs) according to (C1). And in fact, our decomposition algorithm frequently produces components which show computation interference in other states. This is another reason, why we not disallowed computation interference in the definition of parallel composition. In short, (C4) is violated if and only if malfunction on the physical level can occur while the components work in an appropriate environment.

Observe that Definition 4.1 is a generalisation of Definition 3.1: if $(C_i)_{i\in I}$ consists of only one component C_1 then $C = C_1$, no computation interference can occur, and (C4) can be dropped. Furthermore, in [VW02, VK05], a correctness definition in a bisimulation style was presented for deterministic N and applied in the context of decomposition; this definition is easily seen to be equivalent to Definition 4.1 for general N.

Analogously to the notion of correct implementation, the notion of correct distributed implementation can be re-formulated purely in terms of the language, if the specification and the implementation are known to be output-determinate.

Definition 4.2 (Trace-Correct Distributed Implementation)
Let N and $C \stackrel{\text{df}}{=} \|_{i\in I} C_i$ be output-determinate STGs. Then $(C_i)_{i\in I}$ is a *trace-correct distributed implementation* of N, if C is a trace-correct implementation of N and for every trace w of N the following holds:

(TC3) If $w\big|_{C_j} x^{\pm}$ is a trace of C_j for some $x \in Out_j$, then $w\big|_C x^{\pm}$ is a trace of C (no computational interference).
◇

This definition can be viewed as a *denotational* notion of correctness, as opposed to the *operational* one given in Definition 4.1. The result below shows that this notion is equivalent to Definition 4.1, if the implementation is deterministic and the specification is output-determinate. Proposition 3.6 is obtained as a special case of this theorem by considering $I = \{1\}$ and $C = C_1$.

Theorem 4.3 (Justification of the notion of trace-correct distributed implementation)
Let N be an output-determinate STG and $C \stackrel{\text{df}}{=} \|_{i\in I} C_i$ be a parallel composition of deterministic STGs such that $In_C \subseteq In_N$ and $Out_C = Out_N$. Then $(C_i)_{i\in I}$ is a correct distributed implementation of N iff it is a trace-correct distributed implementation of N.

Proof. First we prove that, if $(C_i)_{i\in I}$ is a correct distributed implementation of N, then it is also a trace-correct distributed implementation of N. We consider each requirement of Definition 4.2 in turn, and show that they follow from Definition 4.1.

(TC1) Coincides with (C1).

(TC2) Let w be a trace of N, i.e., $M_N[w\rangle\rangle M$ for some reachable marking M of N. Then, by (C1), $M_C[w\big|_C\rangle\rangle(M_i)_{i\in I}$ for some marking $(M_i)_{i\in I}$ of C. Since all the components are deterministic, the marking $(M_i)_{i\in I}$ is uniquely determined by $w\big|_C$, and thus for any $x \in Out_N$: if $w\big|_C x^{\pm}$ is a trace of C, then $(M_i)_{i\in I}[x^{\pm}\rangle\rangle$ and $wx^{\pm}$ is a trace of N by (C3).

(TC3) Let w be a trace of N, i.e., $M_N[w\rangle\rangle M$ for some reachable marking M of N. Then, by (C1), $M_C[w\big|_C\rangle\rangle(M_i)_{i\in I}$ for some marking $(M_i)_{i\in I}$ of C. Suppose now that $x \in Out_j$ and $w\big|_{C_j} x^{\pm}$ is a trace of C_j for some $j \in I$. Since all the components are deterministic, M_j is uniquely determined by $w\big|_{C_j}$, and thus $M_j[x^{\pm}\rangle\rangle$. Therefore, by (C4), $(M_i)_{i\in I}[x^{\pm}\rangle\rangle$ and $w\big|_C x^{\pm}$ is a trace of C.

Now we show that if C is a trace-correct distributed implementation of N then it is also a correct implementation of N. We consider each requirement in Definition 4.1 in turn, and show that it follows from Definition 4.2.

(C1) Coincides with (TC1).

(C2) Let w be a trace of N, i.e., $M_N[w\rangle\rangle M$ for some reachable marking M of N. Then, by (TC1), $M_C[w\big|_C\rangle\rangle M'$ for some reachable marking M' of C. Let $a \in In_N$ be such that $M[a^{\pm}\rangle\rangle$. Since $wa^{\pm}$ is a trace of N, $wa^{\pm}\big|_C$ is a trace of C by (TC1), i.e., either $a \notin In_C$ or $M'[a^{\pm}\rangle\rangle$, as M' is uniquely defined by $w\big|_C$ due to the determinism of C.

(C3) $\Rightarrow$ Similar to the case for (C2).

$\Leftarrow$ Let w be a trace of N. Then for some marking M of N, $M_N[w\rangle\rangle M$. Suppose $M_C[w\big|_C\rangle\rangle(M_i)_{i\in I}[x^\pm\rangle\rangle$ for some marking $(M_i)_{i\in I}$ of C. Then $wx^\pm$ is a trace of N by (TC2), i.e., $M_N[w\rangle\rangle M'[x^\pm\rangle\rangle$ for some reachable marking M' of N, and so $M[x^\pm\rangle\rangle$ due to the output-determinacy of N.

(C4) Let w be a trace of N. Then $M_N[w\rangle\rangle M$ for some marking M of N, and $M_C[w\big|_C\rangle\rangle(M_i)_{i\in I}$ by (TC1). Suppose $x \in Out_j$ and $M_j[x^\pm\rangle\rangle$ for some $j \in I$. Then $w\big|_C x^\pm$ is a trace of C by (TC3). Since C is deterministic, the marking $(M_i)_{i\in I}$ is uniquely determined, and thus $(M_i)_{i\in I}[x^\pm\rangle\rangle$.

□

4.2 The Decomposition Algorithm

Given a specification STG N, the algorithm works as follows:

- Choose a *feasible partition* $(In_i, Out_i)_{i\in I}$ for some set I with $Out_i \subseteq Out_N$ and $In_i \subseteq In_N \cup Out_N$ for each $i \in I$ (as explained in greater detail below). For each $i \in I$, a component C_i will be constructed, which produces the signals in Out_i by taking into account only the signals in In_i.
- Construct an *initial decomposition* $(C_i)_{i\in I}$ as follows. For each $i \in I$, the *initial component* $C_i = (P, T, W, l_i, M_N, In_i, Out_i)$ is a copy of N except for the labelling and the signal classification. If $l(t) \in (In_N \cup Out_N) \setminus (In_i \cup Out_i)$, then the label is changed to $l_i(t) = \lambda$; such t and its original signal are *hidden*. In contrast, transitions which have label λ in N already are called *specification dummies* or *spec-dummies* for short.

Then perform the following steps to one of the C_i after the other:

- Repeatedly apply LOD-transformations or backtracking:

  Backtracking: For some hidden signal $s \notin In_i \cup Out_i$, add s to In_i and replace C_i by the respective new initial component.
- Eventually, check C_i for output-determinacy. If the check fails, perform backtracking for some hidden signal or, if no hidden signal is left, report that N is not output-determinate. Otherwise, component C_i is constructed.

We now give some more detailed explanations for the steps of our algorithm. A *feasible partition* is a family $(In_i, Out_i)_{i\in I}$ for some set I such that the sets Out_i, $i \in I$, are a partition of Out_N and for each $i \in I$ we have $In_i \subseteq In_N \cup Out_N \setminus Out_i$, and furthermore:

(F1) If signal s and output signal x of N are in structural conflict, then $x \in Out_i$ implies $s \in In_i$ for $s \in In$ and $s \in Out_i$ for $s \in Out_N$ for each $i \in I$.

The rationale for this is: clearly, 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 two conflicting outputs cannot be produced by two different components in a speed-independent way.

(F2) If there are $t, t' \in T_N$ such that $l(t') \in Out_i$ and t is a signal trigger of t', then the signal of t is in $In_i \cup Out_i$.

The latter signal might be in In_i even if it belongs to Out_N; in this case, it will be produced by some other component, and the ith component just listens to it.

As yet, it is not clear how to choose a feasible partition that gives an optimal decomposition in some sense, e.g. one with the least overall size of the reachability graphs of its components. But there is a canonical candidate: according to (F1), output signals in structural conflict must be in the same Out_i, and there is a finest partition of Out_N satisfying this; for each of the resulting Out_i, there is a least set In_i such that (F1) and (F2) are satisfied. In many cases, this canonical feasible partition will have one (In_i, Out_i) for each output signal.

The main idea of the algorithm is now to remove the λ-transitions using appropriate secure transition contractions and other LOD-transformations. This way, we hopefully make the component STGs small enough that the check for output-determinacy and further synthesis can be performed fast. In an *optimistic strategy*, one performs LOD-transformations as long as possible – with our list of LOD-transformations, this will terminate eventually, see below, – and only backtracks if forced to in the last step.

Observe that backtracking modifies the feasible partition in such a way that the resulting partition is feasible again; in particular, C_i already has all signals that are in structural conflict with some output signal of C_i.

Backtracking undoes all the LOD-transformations 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 have studied in [SVWK06] how to implement backtracking in such a way that not always all the LOD-transformations are undone.

The algorithm of this paper is a generalisation of the decomposition algorithm in [VK05], where the latter only dealt with deterministic specifications; for these, the latter algorithm considered the same partitions, transformations, and backtracking. Since the concept of output-determinacy was not available, it was required to remove all λ-transitions; thus, backtracking had also to be performed for a hidden signal if a respective transition could not be contracted, e.g. because it was on a loop or had an arc with weight greater one. Since backtracking applies to all transitions of a signal, one had to un-hide a number of transitions just for technical reasons, although they had already been removed successfully. This can make the reachability graph much larger, while from the perspective of circuit design the additional signal might not be needed. We have the chance to avoid this in the present paper, and this is an important contribution.

If a transition contraction generates a new dynamic auto-conflict, then – as explained in [VW02, VK05] – this is an indication that the original signal of the contracted transition might be important for producing the proper outputs; here we can add that the latter corresponds to a violation of output-determinacy. Thus, to be sure to get a correct result, it was recommended to backtrack in case of a new dynamic auto-conflict; to make this strategy efficient, one has to avoid the generation of the reachability graph, hence it was recommended to backtrack in case of a new structural auto-conflict. With this strategy, the algorithm of [VK05] is guaranteed to find a correct decomposition without any final check.

In another version discussed in [VK05], the algorithm does not backtrack in case of a new structural auto-conflict. The hope is that the conflict might not indicate a dynamic auto-conflict, and that avoiding backtracking gives a smaller component. The price to pay is a final sanity check as in our present algorithm: in the end, components had to be checked for determinism, which is more restrictive than our check. The experience reported in [SVWK06] is that the hope is most often in vain.

Consequently, we recommend a *conservative strategy*: whenever the contraction of a hidden transition creates a new structural auto-conflict, one should backtrack on the respective signal – unless the conflicting transitions are duplicates and one of them can thus be deleted. In this latter case, the conflict clearly does not indicate a violation of output-determinacy. There is no obvious recommendation if a new structural auto-conflict is created by the contraction of a spec-dummy.

If all components are constructed successfully, circuits are synthesised from them using tools like Petrify or Mpsat. Such tools build the reduced state-vector tables for Boolean minimisation for each C_i, which can be viewed as derived from the respective deterministic finite automaton $DA(C_i)$. Hence, the equations derived from the state graphs give a correct implementation of the specification N, as we will prove in the next subsection.

4.3 Correctness

Before we present the correctness proof, observe that we have partitioned the λ-transitions into hidden transitions and spec-dummies. Observe further that notions like signal trigger and weak syntactic conflict (Definition 3.8) are concerned with λ-transitions; when we speak of signal triggers in condition (F2), we consider N, i.e. the respective λ-transitions are spec-dummies; when we apply SecTC2' to a component and check for a weak syntactic conflict, the respective λ-transitions could be hidden transitions as well as spec-dummies.

We start with two lemmata. The first of them implies that during decomposition we have as an invariant that some C_i is not output-determinate or $(C_i)_{i\in I}$ is a trace-correct distributed implementation of N (also cf. the proof of the correctness theorem below).

Lemma 4.4
Let N be an STG with an initial decomposition $(C_i)_{i\in I}$ where all components are output-determinate. Then $(C_i)_{i\in I}$ is a trace-correct distributed implementation of N.

Proof. (TC1) Let $w \in L(N)$ due to $u \in T^*$. Then, for one transition of u after the other, we can fire all copies of the respective transition in the C_i. In more detail, all copies with label not equal to λ are synchronised in the parallel composition and fire as one transition; the other copies fire one after the other. This shows that $w\big|_C \in L(C)$.

(TC2) & (TC3) Again, let $w \in L(N)$ due to $u \in T^*$. To show (TC3), consider $j \in I$ such that $x \in Out_j$ and $w\big|_{C_j}x^{\pm} \in L(C_j)$ due to the firing sequence vt with $l(t) = x^{\pm}$. Since $l_j(v) = w\big|_{C_j} = l_j(u)$ and C_j is output-determinate, we have a firing sequence $uu't'$ of C_j with $l_j(u') = \lambda$ and $l_j(t') = x^{\pm}$; choose such a u' with minimal length. By minimality, each transition in u' triggers a succeeding transition in $u't'$; thus, if u' contained

a hidden transition, we could consider the last one, which would be a signal trigger of t', a contradiction to (F2) for C_j. We conclude that all transitions in u' are spec-dummies. Thus, firing $uu't'$ in all C_i as in the first part of this proof, we get that $w\big|_C x^\pm$ is a trace of C.

Whenever $w\big|_C x^\pm$ is a trace of C, we have $w\big|_{C_j} x^\pm \in L(C_j)$. So from the above argument, we also see that (TC2) holds since we can fire $uu't'$ in N as well, showing $wx^\pm \in L(N)$. □

Lemma 4.5
Let N be a non-output-determinate STG with an initial decomposition $(C_i)_{i\in I}$. Then some C_i is not output-determinate.

Proof. Suppose that $M_N[wx^\pm\rangle\rangle$ and $M_N[w\rangle\rangle M$ in N with $x \in Out_j$. Then $M_{C_j}[w\big|_{C_j} x^\pm\rangle\rangle$ and $M_{C_j}[w\big|_{C_j}\rangle\rangle M$ in C_j. Assume now that C_i is output-determinate, i.e. $M[x^\pm\rangle\rangle$ and $M[vt\rangle$ with $l(t) = l_j(t) = x^\pm$ and $l_j(v) = \lambda$. As in the previous proof, we choose v to be minimal; then, all transitions in v are weak triggers of t in C_j, none of them can be a signal trigger in N, and thus they all are spec-dummies. This shows that $M[vt\rangle$ also gives rise to $M[x^\pm\rangle\rangle$ in N, hence N is output-determinate contradicting the hypothesis. □

Theorem 4.6
Consider the application of the decomposition algorithm to an STG N.

i) If all components are constructed successfully, then N is output-determinate, $(C_i)_{i\in I}$ is a trace-correct distributed implementation of N and $(DA(C_i))_{i\in I}$ is a correct distributed implementation of N.

ii) If the algorithm reports that N is non-output-determinate, then this is the case.

iii) If only the LOD-transformations from the list in Subsection 3.4 are applied, the algorithm terminates.

Proof. i) Suppose all components are constructed successfully. If N were not output-determinate, we could consider the initial components that arise after the last backtracking. By Lemma 4.5, one of them would not be output-determinate, and this would be preserved by the LOD-transformations, contradicting our hypothesis.

This consideration also implies that all mentioned initial components are output-determinate; hence, by Lemma 4.4 this initial decomposition $(C_i)_{i\in I}$ is a trace-correct distributed implementation of N. Since the final components are language-equivalent to the initial ones, also the final parallel composition is language-equivalent to the initial one (see e.g. [Vog92, Cor. 3.1.8] for the well known congruence result for language equivalence w.r.t. parallel composition). Hence, (TC1) – (TC3) hold just as well for the final components.

Also determinisation of the C_i preserves the language, and the third claim follows from Theorem 4.3.

ii) The algorithm reports that N is non-output-determinate only if there is some component without hidden signals which is not output-determinate. In this case, the respective initial component is also not output-determinate; this initial component is identical to N except that some outputs of N might be inputs. It is easy to see that in this situation the violation of output-determinacy carries over to N.

iii) This is a result from [VK05]: backtracking un-hides a hidden signal, which can only be done finitely often. When no backtracking occurs, contractions and transition deletions reduce the number of transitions, while the deletion of places reduces the number of places without increasing the number of transitions. Hence, the listed LOD-transformations can also be applied only finitely often.

□

It should also be noted that for a consistent N only consistent components are produced, cf. [VK05].

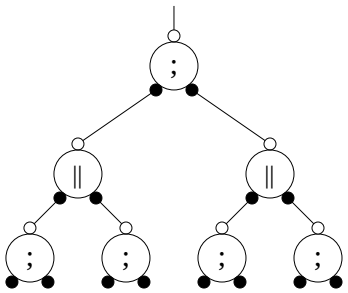
Compared to the approach of [VK05], the above correctness proof is considerably simpler and deals with more general specifications. The price we pay is the check for output-determinacy, which can be avoided in the approach of [VK05]. Additionally, the proof in [VK05] takes care to show that, for deterministic specifications, type-2 secure contractions can be applied without restriction. Since we use the same operations as in [VK05], we can read off from the correctness proof there that the same result applies here if in the specification N there are no weak triggers of or λ-transitions in structural conflict with output transitions; this observation means that we do not have to check for weak syntactic conflicts and this can save a little time.

5 Results

As described in the previous sections, it is now possible to leave λ-transitions in the final components as long as they are output-determinate. Moreover, the new approach can also be used to speed up existing decomposition strategies, in particular *tree decomposition* [SVWK06]. This strategy generates all components together, re-using the intermediate STGs which are generated during decomposition. For efficiency, only some signals are contracted at each stage of the algorithm, resulting in re-usable intermediate STGs. If not all transitions of some signal can be contracted, the contraction of this signal is postponed to later stages of the algorithm, which is detrimental for performance due to the decreased re-use of intermediate STGs. Note that *all* the transitions of this signal are postponed, even if only *one* of them cannot be contracted.

In practice, most of the postponed signals can actually be contracted at later stages of the algorithm. The new approach makes it possible to avoid postponing of signals, leaving the non-contractible transitions as dummies in the intermediate STGs. In most cases, they will be contracted later; otherwise these dummies will remain in the final component. Of course, only transitions which are not contractible due to structural reasons are left in the STG, e.g., if they are incident to an arc with weight ≥ 2. If the contraction would generate a new structural auto-conflict, the contraction of the respective signal is postponed.

We applied this approach to a number of benchmark examples constructed from two basic Balsa handshake components (see [EB02]): the 2-way *sequencer*, which performs two subsequent handshakes on its two 'child' ports when activated on its 'parent' port, and the 2-way *paralleliser*, which performs two parallel handshakes on its two 'child' ports when activated on its 'parent' port; either can be described by a simple STG. The benchmark examples SeqParTree-N are complete binary trees with alternating levels of sequencers and parallelisers, as illustrated in Figure 10 (N is the number of levels), which are generated by the parallel composition of the elementary STGs corresponding to the individual sequencers and parallelisers in the tree. These benchmarks do not have CSC. We also worked with other benchmarks made of handshake components (e.g., trees of parallelisers only); the results did not differ much, so we consider here only SeqParTree-N.



	Size	Signals
Benchmark	$\|P\| - \|T\|$	$\|In\| - \|Out\|$
SeqParTree-05	382 – 252	33 – 93
SeqParTree-06	798 – 508	65 – 189
SeqParTree-07	1566 – 1020	129 – 381
SeqParTree-08	3230 – 2044	257 – 765
SeqParTree-09	6302 – 4092	513 – 1533
SeqParTree-10	12958 – 8188	1025 – 3069

Figure 10: **Left**: SeqParTree-03. Filled dots denote active handshake ports (they can start a handshake), blank dotes denote passive ones. Each port is implemented by two signals, an input and an output. If two ports are connected then the parallel composition merges these four signals into two outputs. **Right**: Sizes of the STGs in the SeqParTree series.

In contrast to the decomposition method of [CC03, CC06], we allow components with more than output. This was utilised by choosing the initial partition in such a way that each component of the decomposition corresponds to one handshake component. Other partitions of the outputs might lead to even faster synthesis; there are also approaches for the automatic detection of suitable partitions, see [SVWK06].

We applied four variants of tree decomposition to these benchmarks, as well as stand-alone Petrify and Mpsat. (The tool for CSC conflict resolution and decomposition described in [CC06, Car03] was not available from the authors.) The experiments were conducted on a PC with Pentium 4 HT/3GHz processor and 2GB RAM.

We performed two variants of tree-decomposition: with postponing signals, and with leaving the non-contractible transitions as dummies in the intermediate STGs, as described above. Additionally, we performed some experiments using only *safeness-preserving contractions*, i.e., contractions which do not destroy the safeness of the STG. (This kind of contraction is needed to combine decomposition with unfolding techniques for STG synthesis, see [KS07].) Essentially, the preservation of safeness is another condition which can prevent some contractions and thus increase the runtime. This resulted in four series of experiments, see Table 1. In the end, the final components were synthesised (which included the resolution of CSC conflicts) with Petrify, which was possible for every component. As explained in Section 3, this means that the resulting components were output-determinate, i.e., the decomposition can be considered as successful. Moreover, the resulting components

are equal for all series and the synthesis is hence given only once.

	Safe		Non-Safe		
Benchmark	New	Old	New	Old	Synthesis
SeqParTree-05	1	1	1	2	5
SeqParTree-06	4	4	3	5	16
SeqParTree-07	8	9	8	9	22
SeqParTree-08	17	32	19	21	1:02
SeqParTree-09	1:18	1:27	1:24	1:29	1:30
SeqParTree-10	6:03	42:37	5:49	7:04	4:32

Table 1: Results of the handshake benchmarks. Columns 2 – 5 give the pure decomposition time, the last column gives the Petrify synthesis time for the components. Times are given in seconds or as minutes:seconds. *Safe* means safeness-preserving contractions, the *old* method does not leave λ transitions in the intermediate results, the *new* one does.

The synthesis with stand-alone Petrify or Mpsat has not terminated within 12 hours, even for SeqParTree-05, as the corresponding STGs are very large. We consider it as a notable achievement that the proposed approach could synthesise them so quickly — e.g., SeqParTree-10 with more than 4000 signals was synthesised in about 7–11 minutes. One can see that leaving non-contractible transitions as dummies in the intermediate STGs is useful, especially for the case of safeness-preserving contractions. The reason is that in these benchmarks the algorithm does not encounter many non-contractible λ-transitions. On the other hand, it encounters many λ-transitions whose contraction is non-safeness-preserving, which significantly increases the runtime of the old approach. As one can see, the new approach leaving such transitions as dummies in the intermediate STGs is much faster.

6 Conclusion

In this paper we proposed the concept of output-determinacy, which is a generalisation of determinism. It allowed us to define in a natural way a semantics of non-deterministic STGs, in particular STGs with dummies. We showed that non-output-determinate specifications are ill-formed, whereas the semantics of an output-determinate STG is its language. Moreover, for the class of output-determinate STGs we gave a denotational (language-based) notion of a correct implementation, and showed that it is consistent with the corresponding operational notion. The computational complexity of checking output-determinacy has been investigated for several important net classes, and a practical test for the cases of safe or bounded divergence-free STGs has been developed.

One of the main application of the theory developed in this paper is the new algorithm for decomposition of STGs. This algorithm is much more flexible than the one in [VW02, VK05]. In particular, it no longer requires that all the λ-transitions must be contracted in the final components, and it can use more net reductions; moreover, the list of such reductions can easily be extended by adding new LOD-transformations. The experimental results show that our decomposition algorithm can handle very large STGs efficiently. Combined with tools for logic synthesis [KS07], it can be used in the context of control re-synthesis, as explained in the introduction.

Acknowledgements

We would like to thank Dominic Wist for helping us with generating the benchmarks. This research was supported by DFG-projects 'STG-Dekomposition' Vo615/8-2, and the Royal Academy of Engineering/Epsrc grant EP/C53400X/1 (Davac).

References

[And83] C. André. Structural transformations giving B-equivalent PT-nets. In Pagnoni and Rozenberg, editors, *Applications and Theory of Petri Nets*, Informatik-Fachber. 66, 14–28. Springer, 1983.

[Ber87] G. Berthelot. Transformations and decompositions of nets. In W. Brauer et al., editors, *Petri Nets: Central Models and Their Properties*, Lect. Notes Comp. Sci. 254, 359–376. Springer, 1987.

[Ber93] K. v. Berkel. Handshake Circuits: an Asynchronous Architecture for VLSI Programming. *International Series on Parallel Computation*, 5, 1993.

[Car03] Josep Carmona. *Structural Methods for the Synthesis of Well-Formed Concurrent Specifications*. PhD thesis, Universitat Politècnica de Catalunya, 2003.

[CC03] 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*, pages 818–825, 2003.

[CC06] J. Carmona and J. Cortadella. State Encoding of Large Asynchronous Controllers. In *Proc. DAC'06*, pages 939–944, 2006.

[Chu87] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT, 1987.

[CKK+97] 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*, E80-D, 3:315–325, 1997.

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

[EB02] D. Edwards and A. Bardsley. BALSA: an Asynchronous Hardware Synthesis Language. *The Computer Journal*, 45(1):12–18, 2002.

[Ebe92] J. Ebergen. Arbiters: an exercise in specifying and decomposing asynchronously communicating components. *Sci. of Computer Programming*, 18:223–245, 1992.

[Esp98] J. Esparza. Decidability and complexity of petri net problems — an introduction. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, Lect. Notes Comp. Sci. 1491, pages 374–428. Springer-Verlag, 1998.

[ITR05] International Technology Roadmap for Semiconductors: Design, 2005.
URL: `www.itrs.net/Links/2005ITRS/Design2005.pdf`.

[Kho03] V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, School of Computing Science, Newcastle University, 2003.

[KKT93] A. Kondratyev, M. Kishinevsky, and A. Taubin. Synthesis method in self-timed design. Decompositional approach. In *IEEE Int. Conf. VLSI and CAD*, pages 324–327, 1993.

[KS07] V. Khomenko and M. Schaefer. Combining decomposition and unfolding for STG synthesis. Technical Report 2007-01, University of Augsburg, 2007. URL: `http://www.informatik.uni-augsburg.de/ skripts/techreports/`.

[Mel98] S. Melzer. *Verifikation Verteilter Systeme mit Linearer — und Constraint-Programmierung*. PhD thesis, Technische Universität München, Utz Verlag, 1998.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[Sch06] M. Schäfer. CSC-Aware STG-Decomposition. In *Proc. 18th UK Asynchronous Forum*. Newcastle University, 2006.

[SV05] M. Schaefer and W. Vogler. Component refinement and CSC solving for STG decomposition. In Vladimiro Sassone, editor, *FOSSACS 05*, Lect. Notes Comp. Sci. 3441, pp. 348–363. Springer, 2005.

[SVWK06] M. Schaefer, W. Vogler, R. Wollowski, and V. Khomenko. Strategies for optimised STG decomposition. In *Proceedings of ACSD*, 2006.

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

[Vog92] W. Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*. Lect. Notes Comp. Sci. 625. Springer, 1992.

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

[YKK+96] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with or causality. *Formal Methods in System Design*, 9:189–233, 1996.

[YKKL94] A. Yakovlev, M. Kishinevsky, A. Kondratyev, and L. Lavagno. Or causality: Modelling and hardware implementation. In R. Valette, editor, *Applications and Theory of Petri Nets 1994*, Lect. Notes Comp. Sci. 815, 568–587. Springer, 1994.

A Non-Output-Determinacy cannot be resolved by signal insertion

In this section we deal with STGs with *internal* signals. An STG is now defined as $N = (P, T, W, M_N, In, Out, Int, l)$, where Int is the set of internal signals, such that $Int \cap (In \cup Out) = \emptyset$. We will denote by $Ext_N \stackrel{\text{df}}{=} In \cup Out$ the set of *external* signals of N. Internal signals are considered as outputs of the STG which the environment ignores. However, they still must be produced by the circuit, as they (unlike dummies) occur in traces and are a part of the state encoding. In practice, internal signals are used for resolution of CSC conflicts. For more details on the semantics of internal signals, see [SV05].

In this section we consider specifications without internal signals, and allow them for implementations only. This simplification can be justified as follows. If the original specification contains internal signals then they can be turned into dummies, as they are ignored by the environment. Alternatively, if the internal signals are inserted into the STG to resolve CSC conflicts, they can be turned into outputs, as the circuit has to produce them. We now generalise the notion of correct implementation given in Definition 3.1 to implementations with internal signals. Observe that we require $Int_C \cap Ext_N = \emptyset$ for technical reasons only; this can always be achieved by a suitable renaming.

Definition A.1 (Correct Implementation with Internal Signals)
A deterministic STG C (with internal signals) is a *correct implementation* of an STG N without internal signals if $In_C \subseteq In_N$, $Out_C = Out_N$, $Int_C \cap Ext_N = \emptyset$, and for all w and all M such that $M_N[w\rangle\rangle M$ the following hold:

(I1) There is a trace v of C such that $M_C[v\rangle\rangle$ with $v|_{Ext_C} = w|_{Ext_C}$.

(I2) For every trace v of C such that $M_C[v\rangle\rangle M'$ with $v|_{Ext_C} = w|_{Ext_C}$: If $M[a^{\pm}\rangle\rangle$ then either $M'[a^{\pm}\rangle\rangle$ or $a \notin In_C$.

(I3) For every trace v of C such that $M_C[v\rangle\rangle M'$ with $v|_{Ext_C} = w|_{Ext_C}$: If $x \in Out_N$, then $M[x^{\pm}\rangle\rangle$ iff $M'[v'x^{\pm}\rangle\rangle$ for some $v' \in (Int_C^{\pm})^*$. ◇

Violation of output-determinacy always results in a CSC-conflict. Below we show that this conflict cannot be resolved by insertion of internal signals. In fact, we prove a more general statement: a non-output-determinate specification cannot be implemented by a deterministic STG with internal signals. Thus, behaviour-preserving insertion of internal signals into a non-output-determinate STG always results in a non-output-determinate STG, and the latter still has CSC conflicts. The result below is an extension of Proposition 3.3 to implementations with internal signals.

Proposition A.2
Let N be an STG without internal signals and C (with internal signals) be a correct implementation of N. Then N is output-determinate.

Proof. For $x \in Out_N$, let $M_N[w\rangle\rangle M_1[x^{\pm}\rangle\rangle$ and $M_N[w\rangle\rangle M_2$. Then, by (I1) of Definition A.1, $M_C[v\rangle\rangle M'$ for some v such that $v|_{Ext_C} = w|_{Ext_C}$, and by (I3), $M'[v'x^{\pm}\rangle\rangle$ for some $v' \in (Int_C^{\pm})^*$. Therefore, by (I3) $M_2[x^{\pm}\rangle\rangle$. □