Fundamenta Informaticae XX (2011) 1–39
IOS Press

Output-Determinacy and Asynchronous Circuit Synthesis

Victor Khomenko
School of Computing Science
Newcastle University, UK
victor.khomenko@ncl.ac.uk

Mark Schaefer [C] **Walter Vogler**
Institute of Computer Science
University of Augsburg, Germany
{*mark.schaefer, 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 present some experimental data for this improved algorithm and 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 electro-magnetic emission, no problems with clock skew and related subtle issues, and are fundamentally more tolerant of voltage, temperature and manufacturing process variations [CKK+02]. 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.

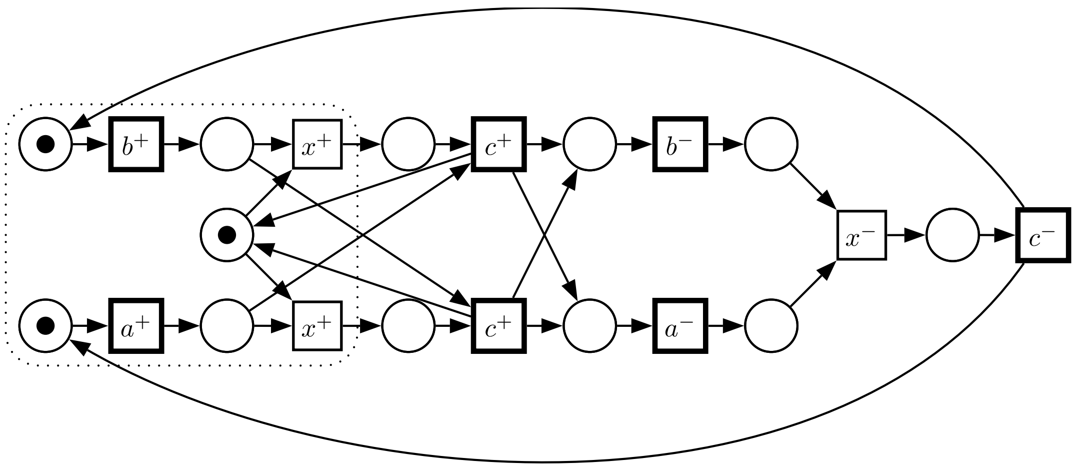


Figure 1. OR-causality (the ‘interesting’ part of the STG is highlighted): a^+ and b^+ are concurrent inputs, and the output x^+ can be produced upon arrival of either of them. Note that the two transitions labelled x^+ are in dynamic auto-conflict, i.e. the specification is non-deterministic. However,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

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 negligible delays, or, alternatively, wire forks are assumed to be isochronic). *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 and formal language 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 a safe Petri net is used for modelling a situation 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 [YKK$^+$96].

Address for correspondence: M. Schaefer, Institut für Informatik, 86135 Augsburg, Germany
[C]Corresponding author

[1]OR-causality can also be modelled as a non-safe Petri net without non-determinism [KKTV94, YKK$^+$96], but in practice safe Petri nets are preferable as they are much easier to analyse.

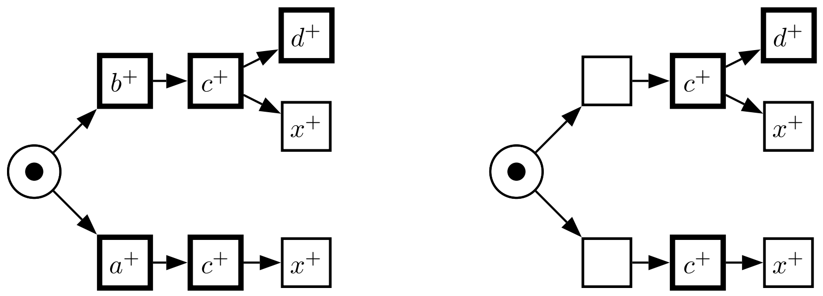


Figure 2. Non-determinism due to hiding. After hiding signals a and b, the STG becomes non-deterministic, but it can be implemented (the system can simply wait for c, and produce x upon receiving it; input d can be ignored). Note that the two branches after the non-deterministic choice are not entirely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

Hiding of signals Non-determinism naturally arises when in a deterministic specification some of the signals are hidden (i.e. the respective transitions are labelled with the empty word λ), as illustrated in Fig. 2. In fact, hiding of signals is an essential part of the decomposition algorithm of [VW02, VK06], which we will improve in the present paper.

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; cf. for example [Mil89] for the concept of determinacy. In particular, it is shown that for output-determinate STGs the language *is a sufficient semantics* (see Section 3), and this also holds in the case of distributed STGs (see Section 4). We also prove that an STG cannot be implemented as an speed-independent circuit if it is not output-determinate (see Section 6). A large part of the paper is devoted to an important application of the developed theory of output-determinacy: we will generalise the decomposition algorithm of [VW02, VK06] 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 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 circuit with more than 20–30 signals with PETRIFY is often impossible.) Hence, this approach does not scale.

[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.

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. There are strong restrictions on the structure and labelling of STGs in [Chu87]; the improved decomposition algorithm of Vogler, Wollowski and Kangsah [VW02, VK06] works for arbitrary deterministic specifications; here, we generalise this to output-determinate specifications.

Based on our theory, we develop a more efficient variant of the decomposition algorithm. In the decomposition algorithm, each component is obtained from the original STG by hiding some of the signals in it, and then removing the corresponding transitions by applying so-called reduction operations; the most important of these transformations is the contraction of λ-labelled transitions (cf. Definition 2.1). 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 [VK06] 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 synthesising 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 language-based, and might be easier to grasp than the bisimulation-based proofs in [VW02, VK06]. 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 extend the set of valid transformations.

An alternative 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 *re-synthesis*, i.e. one applies logic synthesis to the control path extracted from a BALSA specification. This control path can be partitioned into smaller clusters which can be handled by logic synthesis, and the clusters 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 halve the area devoted to control flow and improve its latency, compared with the traditional syntax-directed translation, as long as the size of clusters which can be confidently handled by logic synthesis 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 [KS07] 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. Our decomposition algorithm and all its variants are implemented in the tool DESIJ [Sch07].

The paper is organised as follows: in the next section we introduce the basic concepts of Petri nets and STGs, including the reduction operations from [VK06]. In Section 3, the new notion of output-determinacy is introduced and justified by showing that for output-determinate STGs (and only for them) an implementation relation can purely be based on the language; we also analyse the complexity of checking output-determinacy. In the following section, we give an analogous result for the case where the implementation is a parallel composition. Furthermore, we present the new variant of the STG decomposition algorithm together with a list of semantics-preserving transformations and prove its correctness. We then give some first experimental results in Section 5, and generalise our theory to STGs with internal signals in Section 6.

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 is 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 is the *initial marking,* where a *marking* is a multiset of places, i.e. a function $P \to \mathbb{N}_0$ (also written as $\mathbb{N}_0^P$) which assigns a number of *tokens* to each place. A Petri net can be considered as a bipartite graph with weighted arcs 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 t can *fire* or *occur* yielding a new marking M', written as $M[t\rangle M'$, where $M'(p) = M(p) - W(p, t) + W(t, p)$, for all $p \in P$. A transition sequence $v = t_1 \ldots t_n$ is *enabled under*

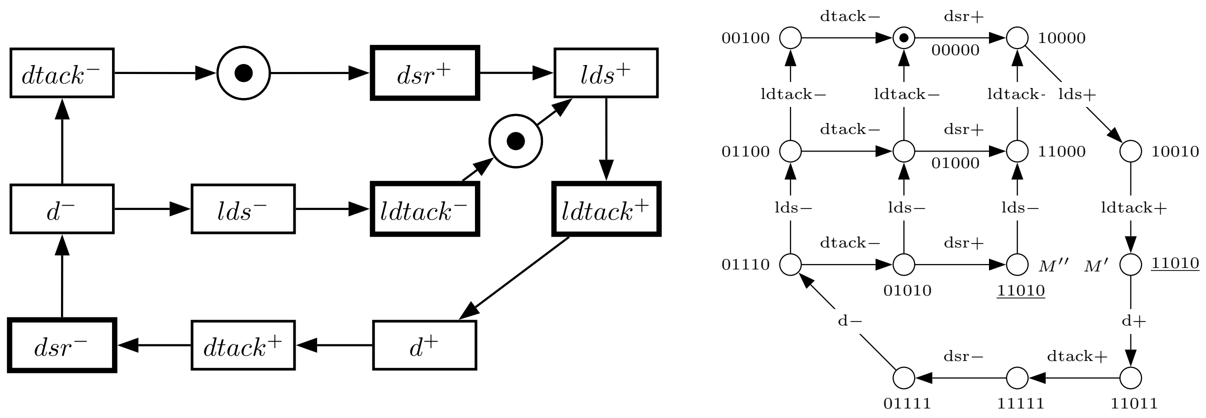


Figure 3. An STG modelling a simplified VME bus controller (left) and its state graph with a CSC conflict between the underlined states (right). The signal order in the binary encodings is: *dsr, ldtack, dtack, lds, d*.

a marking M (yielding M') if $M[t_1\rangle\ M_1[t_2\rangle\ \ldots\ M_{n-1}[t_n\rangle M_n = M'$, and we write $M[v\rangle$, $M[v\rangle M'$ resp.; v is called *firing sequence* if $M_N[v\rangle$. The empty transition sequence λ is enabled under every marking. M is called *reachable* if a transition sequence v with $M_N[v\rangle M$ exists.

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 are mostly concerned with bounded Petri nets and STGs.

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 opposite 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. Usually, the letters close to the beginning of the alphabet ($a, b, c, \ldots$) denote input signals and those close to the end of the alphabet (x, y, z) denote output signals.

An STG may initially contain transitions labelled with λ, called *dummy* transitions, which do not correspond to any signal change. *Hiding a signal* s means to change the label of all transitions labelled with $s^{\pm}$ to λ, and *unhiding* means to change the labels back to the initial values.

An example of an STG is shown in Fig. 3(left) (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 presets and postsets are not drawn if the corresponding arcs have the weight 1; they are implicitly given by an arc between these two transitions. Transitions are drawn as rectangles together with their labelling (input transitions with a thick border), and the weight function is drawn as directed arcs xy whenever $W(x, y) \neq 0$ (and 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; it is 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.

The *reachability graph* RG_N of an STG N is an arc-labelled directed graph on the reachable markings with M_N as root; there is an arc 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 Fig. 3(right). N is *deterministic* if its reachability graph is a deterministic automaton: it contains no λ-labelled transitions and there are no dynamic auto-conflicts, i.e. for each reachable marking M and each signal edge $s^{\pm}$ there is at most one M' with $M[s^{\pm}\rangle\rangle M'$. Note that a deterministic STG can have choices between different outputs. For example, an STG modelling a standard arbiter is deterministic; see the discussion of *output-persistency* at the end of this section. For deterministic STGs, language equivalence and bisimulation coincide.

If RG_N is not deterministic, one can turn it (using well-known automata-theoretic methods) into a language equivalent deterministic automaton with accepting states only; in particular, the resulting automaton will have no λ-arcs. (Note that this version of a deterministic automaton is in general not complete.) We call this transformation *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 the wires corresponding to the signals. 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 a component may be an input of other components, and in any case it is an output of the composition.

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.

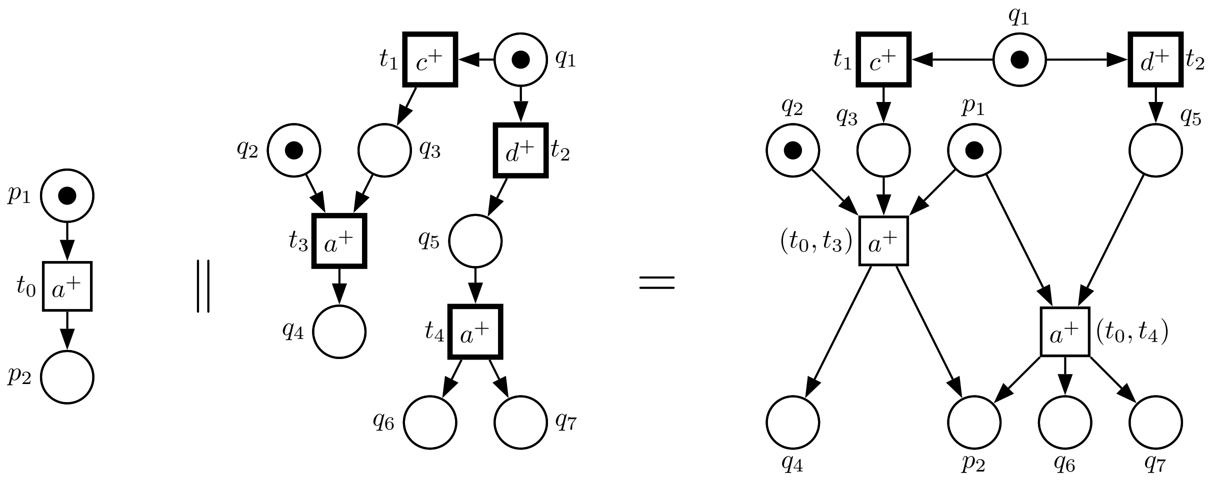


Figure 4. Parallel composition example. In the net fragment on the left hand side, signal a is an output, and in the fragment in the middle it is an input. Hence, in their parallel composition (right) it is an output. In this example, there is *computation interference*: the left component activates an a^+ but the middle one is not ready to receive it. This problem is not visible in the parallel composition; see Definition 4.1.

The place set of the composition is 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, and we will also write such a marking $M_1 \dot\cup M_2$ of the composition as (M_1, M_2). To define the transitions, 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 fusing each $s^\pm$-labelled transition t_1 of N_1 with each $s^\pm$-labelled transition t_2 from N_2 if $s \in A$. Such transitions are pairs and 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, $i = 1, 2$; for an example of a parallel composition, see Fig. 4. More generally, we have $(M_1, M_2)[w\rangle\rangle(M_1', M_2')$ iff $M_i[w|_{N_i}\rangle\rangle M_i'$ for $i \in \{1, 2\}$, where $w|_{N_i}$ denotes the projection of the trace w onto the signals of the STG N_i. Hence, all reachable markings of N have the form (M_1, M_2), where M_i is a reachable marking of N_i, $i = 1, 2$.

A composition can also be ill-defined due to what e.g. Ebergen [Ebe92] calls *computation interference* (see Fig. 4); this is a semantic problem, and we will not consider it here, but later in the definition of correctness.

It is easy to see that N is deterministic if N_1 and N_2 are. However, as illustrated in Fig. 4, N might have structural auto-conflicts even if none of the N_i has them.

Obviously, 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 $i \in I = \{1, ..., n\}$. As above, $(M_1, M_2, \ldots, M_n)[w\rangle\rangle(M_1', M_2', \ldots, M_n')$ iff $M_i[w|_{C_i}\rangle\rangle M_i'$ for all $i \in \{1, \ldots, n\}$.

We now introduce transition contraction (see e.g. [And83] for an early reference), which is the most important *reduction operation* of our decomposition procedure. We essentially repeat from [VK06], where further discussions can be found. Intuitively, a transition contraction removes the respective transition from the net and combines each place of the preset with each place of the postset to 'simulate' the firing of the deleted transition. In this definition, $\star \notin P \cup T$ is a pseudo element used for notational convenience to have only pairs of places. We assume $W(\star, t_1) = W(t_1, \star) = M_N(\star) = 0$.

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

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

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

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 $M_N(p) = 0$ for some $p \in t^\bullet$ (*type-2 secure*). $\diamond$

Intuitively, for a type-1 secure contraction, there is no conflict in the preset of t, and for a type-2 secure contraction, essentially there is no merging in the postset of t. Note that, in general, $\overline{N}$ might fail to be consistent, even if N is; but secure contractions preserve consistency [VK06].

Fig. 5 (top) shows a part of a net and the result of contracting the λ-transition. 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 Fig. 5 (bottom). 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 [VK06] and the succeeding corollary of its second part show in which sense secure transition contractions are behaviour-preserving. These results are used in Section 4.

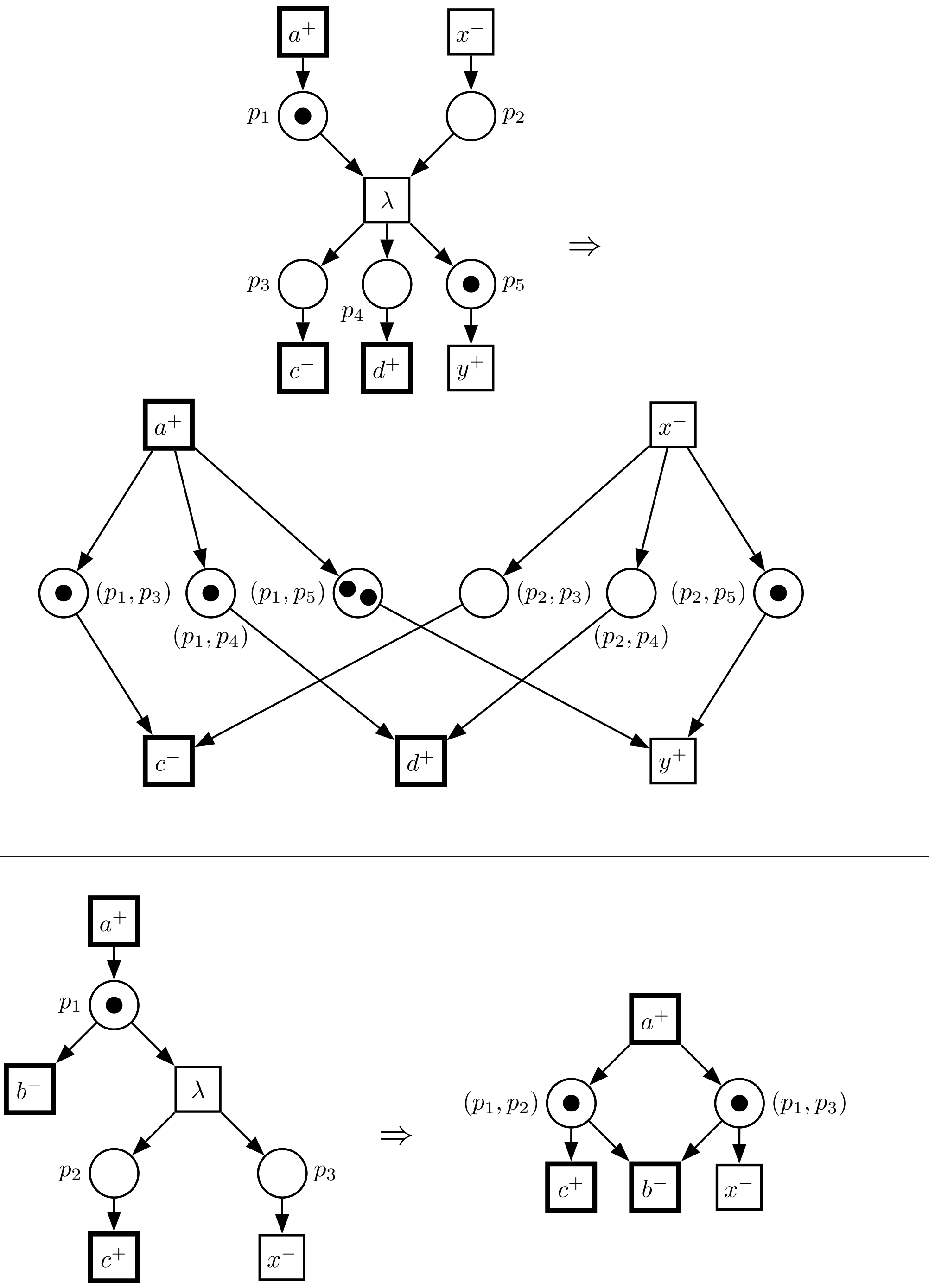


Figure 5. Two examples of a transition contraction.

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 $\mathcal{S} = \{(M, \overline{M}) \mid M$ and $\overline{M}$ satisfy the marking equality$\}$ is a simulation from N to $\overline{N}$, and there is a simulation $\mathcal{S}' \subseteq S^{-1}$ from $\overline{N}$ to N.

3. N and $\overline{N}$ are language equivalent. ◇

Corollary 2.3. If $\overline{N}$ is a type-2 secure contraction of N, then the simulation $\mathcal{S}'$ in Theorem 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 introducing redundant transitions and places; the deletion of such a transition, place resp., (including the incident arcs) is another transformation that is used in our decomposition algorithm.

A transition t is *redundant* if either it is a λ-transition with $W(p,t) = W(t,p)$ for each place p (i.e. t is a *loop-only* transition), or there is another transition t' with the same label such that $W(p,t) = W(p,t')$ and $W(t,p) = W(t',p)$ for each place p (i.e. t is a *duplicate* transition).

A place p is *implicit* if it can be deleted from the net without changing the set of firing sequences. However, detecting implicit places is $\mathcal{PSPACE}$-complete already for safe nets, and during decomposition only *redundant places* [Ber87] are deleted. Redundant places are a subset of implicit ones, and they are defined on the structure of the net; there are efficient linear programming techniques to find them. The details of the definition are not relevant for this paper, see e.g. [VK06].

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 and which input signals the environment is allowed to produce; cf. Subsection 3.1. 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 sv_M to each marking M of RG_N. A state assignment must satisfy for every signal $x \in Sig$ and every pair of markings $M, M' \in [M_N\rangle$ the following properties:

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

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. Fig. 3(right) shows the reachability graph of the STG in Fig. 3(left); every reachable 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, i.e. $M_1[x^{\pm}\rangle\rangle$ iff $M_2[x^{\pm}\rangle\rangle$ for each output signal x. Otherwise, N has a *CSC conflict*, cf. e.g. Fig. 3(right), and no circuit can be synthesised directly. In such a case, 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; cf. Section 6 for a detailed discussion of internal signals.

Another important properties of STGs and asynchronous circuits which has to be fulfilled to permit synthesis is *output-persistency*. An STG is output-persistent if every activated output edge will eventually happen, i.e. enabled outputs cannot be disabled. However, inputs may be in conflict with other inputs. It should be noted that some circuits, like arbiters, can handle a choice between outputs in a speed-independent way with the help of analog circuitry allowing to resolve the arising meta-stability. In practice, to synthesise such circuits, the arbitration is ‘factored out’ to the environment, so that the choice between outputs is transformed into the choice between inputs, making the STG output-persistent; then a standard arbiter is used in the final implementation.

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 the specification may be 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, if an STG is not output-determinate, it is ill-formed and 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, if an STG is *not output-determinate, then synthesising its determinised state graph will 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 reformulate the notion of correct implementation purely in terms of the language. This notion is a pre-congruence for parallel composition, and this plays 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. To illustrate this, in Section 4 we introduce new transformations, which were not used in decomposition algorithms so far.

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

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 input events (and traces), but these events 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 and b 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 $w|_C$.

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:

(C1) $w|_C$ is a trace of C, i.e. $M_C[w|_C\rangle\rangle M'$ 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 $M'[x^\pm\rangle\rangle$. $\diamond$

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 *at*

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

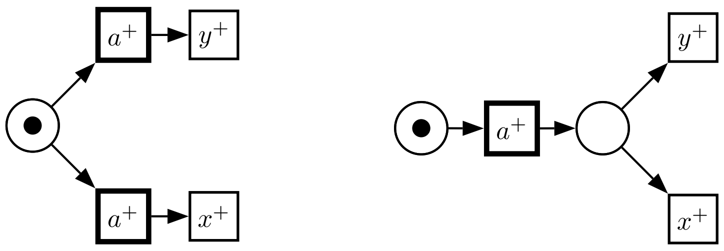


Figure 6. Non-output-persistency due to determinisation. An output-persistent but not output-determinate STG (left) and the non-output-persistent STG (due to the choice between the outputs x and y) obtained from it by determinisation (right). Note that determinisation can also result in a choice between an input and an output (this would be the case if y were an input).

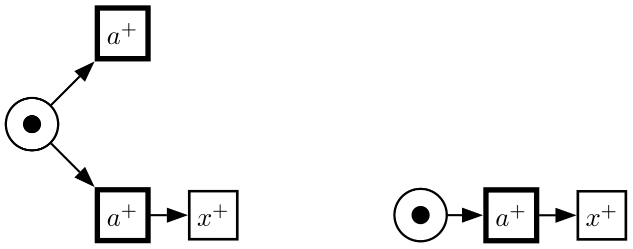


Figure 7. Incorrect determinisation: the STG on the left is not output-determinate; the result of determinisation is shown on the right. The latter STG, though implementable, is not a correct implementation of the original specification: it can cause a failure in the environment by producing x when the environment does not expect it.

least 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 $M_2[x^{\pm}\rangle\rangle$. $\diamond$

For example, the non-deterministic STG in Fig. 2(right) is output-determinate. Clearly, a deterministic STG is always 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 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-output-persistency, 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 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. If a specification of a speed-independent system is not output-determinate, it 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. $\diamond$

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. $\Box$

Observe also that an STG always has CSC conflicts if it is not output-determinate, since, 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. In Section 6 it is shown that such a CSC conflict is *irreducible* [KKTV94], 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. The STG resulting from such an insertion will always have a violation of output-determinacy (and thus CSC conflicts) again.

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 an output-determinate STG 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. $\diamond$

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 of (C3), 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'', as otherwise the language is not preserved. Since N is output-determinate, also $M[x^\pm\rangle\rangle$. $\Box$

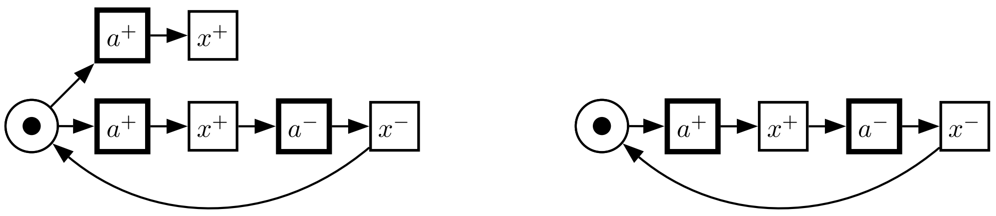


Figure 8. Determinisation: an output-determinate STG N with a deadlock (left) and the deadlock-free STG obtained from N by determinisation (right). The latter STG is a correct implementation of N; intuitively, the execution of x^- is correct, since it only occurs when the environment signalled with a^-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

The proposed semantics has interesting consequences, in particular, a specification with deadlocks can have a deadlock-free implementation, 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.* In fact, in Section 4.2 we relax these requirements even further.

In view of Semantic Rule 2, one would expect that the notion of correct implementation given in Definition 3.1 can be reformulated *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) $w|_C$ is a trace of C;

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

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 whether a specification is output-determinate or not). The result below shows that this notion is equivalent to Definition 3.1 if the implementation is deterministic.

Proposition 3.6. (Justification of the notion of trace-correct implementation)
Let N be an 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, and N is output-determinate in this case. $\diamond$

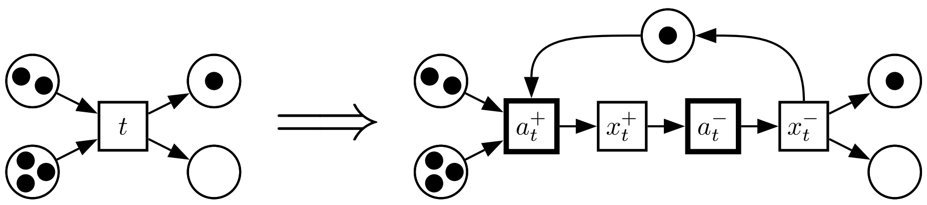
We postpone the proof of this result until the next section, where it is formulated and proven for the more general case of a distributed implementation $C = \|_{i \in I} C_i$. (Note that C in the above result can be seen as being a distributed implementation comprised of a single component.)

3.4. 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.

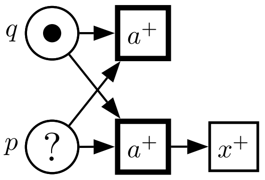
The *coverability* problem is the problem of deciding whether a given Petri net or an STG has a reachable marking M *covering* a given marking M' (i.e. $M \geq M'$). The complementary problem will be called the *uncoverability* problem. A special case of the (un)coverability problem is the *single-place (un)coverability,* where $|M'| = 1$. The computational complexity of this problem for various Petri net classes is well-understood [Esp98], namely it is $\mathcal{PSPACE}$-complete for safe and bounded, and $\mathcal{EXPSPACE}$-complete for unbounded nets.

We now show that the (un)coverability problem on a Petri net can be reduced to the (un)coverability problem on a consistent output-persistent and output-determinate STG without dummies, which belongs to the same class (safe/bounded/unbounded) as the original Petri net. The main idea of the reduction is to split each transition t of the Petri net into four STG transitions, a_t^+, x_t^+, a_t^- and x_t^-, where a_t is a new input and x_t is a new output, as illustrated below:



One can easily show that the size of the resulting STG is linear in the size of the original Petri net, and that this STG contains no dummies, is output-determinate (in fact, deterministic), output-persistent (since no two different output transitions have overlapping presets) and consistent (the place in the feedback with a token on it prevents self-concurrent execution of transitions; note that this arc is not necessary if the original Petri net is safe).[5] Moreover, all the places of the original net are also places of this STG, and they are not in the preset of an output transition; any marking M of the original Petri net is (un)coverable iff M is (un)coverable in the STG, i.e. the (un)coverability problem on a Petri net is reducible to the corresponding problem on this STG.

We proceed by showing that the single-place uncoverability problem on such an STG can be reduced to checking output-determinacy (of a modified STG); this reduction forms the basis of our lower complexity bounds analysis. Given an STG with a place p that is not in the preset of an output transition, we attach the following net fragment to p, where a is a new input and x is a new output:



Note that the obtained STG contains no dummies and is still consistent (since the new place q prevents the new transitions from firing more than once), output-persistent and belongs to the same class (safe/bounded/unbounded) as the original Petri net. Moreover, this STG is output-determinate iff p is uncoverable. In effect, we have proved the following result.

[5]In fact, this STG is implementable by the circuit $[x_t] = a_t$, for all t.

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

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

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

Figure 9. Algorithm checking for violation of output-determinacy.

Proposition 3.7. 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, consistent and output-persistent STG N' which contains no dummy transitions and whose size is linear in the size of N, such that N' is output-determinate iff p is uncoverable in N. $\diamond$

Corollary 3.8. The problem of checking output-determinacy is $\mathcal{PSPACE}$-hard for safe and for bounded STGs and $\mathcal{EXPSPACE}$-hard for unbounded STGs. Moreover, these complexity bounds remain the same even if the STG contains no dummy transitions and is known to be consistent and output-persistent. $\diamond$

Proof:
Follows from Proposition 3.7, 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. $\square$

We complete our analysis by giving tight upper bounds for the problem of checking output-determinacy for the cases of safe and bounded STGs. The basis for this is the non-deterministic algorithm in Fig. 9 for checking whether a net is not output-determinate.

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 not 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, by 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 this algorithm. Indeed, if it accepts N then N is not output-determinate; moreover, every non-output-determinate STG N can be accepted if the algorithm makes a 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 3.9.
Output-determinacy can be decided in $\mathcal{PSPACE}$ for safe and for bounded STGs. $\diamond$

Combined with Corollary 3.8, 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 contains no dummies and is known to be consistent and output-persistent. 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 safe or bounded *divergence-free* STG, i.e. an STG which cannot execute an infinite sequence of λ-transitions from any of its reachable markings.[6] 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 complementary places) that is polynomial in the size of N, or checked directly using, e.g. the unfolding-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 the 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 generalise Definition 3.1 to families of components, additionally taking care of *computation interference* as explained below.

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

Definition 4.1. (Correct Decomposition)
Let N be an STG and $C = \|_{i \in I} C_i$ be a parallel composition of deterministic components. Then $(C_i)_{i \in I}$ is a *correct distributed implementation* (or a *correct decomposition*) 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, $M_C[w|_C\rangle\rangle(M_i)_{i \in I}$ 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. $\diamond$

Thus, computation interference occurs 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, i.e. the problem becomes hidden, as illustrated in Figure 4. Since our decomposition algorithm is correct, it ensures (C4), and such unexpected outputs do not occur in the components produced by it.

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. In fact, it is not possible to treat computation interference in the definition of parallel composition for the following subtle reason: (C4) only forbids computation interference in states that can really occur in appropriate environments, i.e. when performing a trace of N (modulo the irrelevant inputs). In fact, our decomposition algorithm frequently produces components which show computation interference in other reachable markings. 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) trivially holds. Furthermore, in [VW02, VK06] 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 reformulated purely in terms of the language, if the specification and the distributed implementation are known to be output-determinate.

Definition 4.2. (Trace-Correct Distributed Implementation)
Let N and $(C_i)_{i \in I}$ be output-determinate STGs. Then $(C_i)_{i \in I}$ is a *trace-correct distributed implementation* (or *trace-correct decomposition*) of N, if for $C = \|_{i \in I} C_i$ (TC1) and (TC2) of Definition 3.5 hold and for every trace w of N the following holds:

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

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. Clearly, Definition 3.5 is a special case of this definition, and Proposition 3.6 is obtained as a special case of this theorem by considering $I = \{1\}$ and $C = C_1$.

Note that we require the C_i to be output-determinate, but we do not have to require this for C to get a sensible trace-based definition. This is also adequate, because circuits will be synthesised from the C_i (and these *circuits* are composed while C does not have to be built).

Theorem 4.3. (Justification of the notion of trace-correct distributed implementation)
Let N be an STG and $C = \|_{i\in I} C_i$ 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, and in this case N is output-determinate. $\diamond$

Proof:
N is output-determinate by Proposition 3.3. 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). So, let w be a trace of N, i.e. $M_N[w\rangle\rangle M$ for some reachable marking M of N, and $M_C[w|_C\rangle\rangle(M_i)_{i\in I}$ for some marking $(M_i)_{i\in I}$ of C.

(TC2) Since all the components are deterministic, the marking $(M_i)_{i\in I}$ is uniquely determined by $w|_C$, and thus for any $x \in Out_N$: if $w|_C x^\pm$ 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) Suppose now that $x \in Out_j$ and $w|_{C_j} x^\pm$ is a trace of C_j for some $j \in I$. Since C_j is deterministic, M_j is uniquely determined by $w|_{C_j}$, and thus $M_j[x^\pm\rangle\rangle$. Therefore, by (C4), $(M_i)_{i\in I}[x^\pm\rangle\rangle$ and $w|_C x^\pm$ 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 one. We consider each requirement in Definition 4.1 in turn, and show that it follows from Definition 4.2.

(C1) Coincides with (TC1). So, let $M_N[w\rangle\rangle M$ for a marking M of N, and $M_C[w|_C\rangle\rangle(M_i)_{i\in I}$.

(C2) Let $a \in In_N$ be such that $M[a^\pm\rangle\rangle$. Since $wa^\pm$ is a trace of N, $wa^\pm|_C$ is a trace of C by (TC1), i.e. either $a \notin In_C$ or $(M_i)_{i\in I}[a^\pm\rangle\rangle$, as $(M_i)_{i\in I}$ is uniquely defined by $w|_C$ due to the determinism of C.

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

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

$\square$

The next theorem shows that trace-correctness can be applied *hierarchically*, i.e. given a trace-correct distributed implementation, any of its components can in turn be replaced with its own trace-correct distributed implementation. A similar theorem for the bisimulation style correctness of arbitrary STGs can be found in [SV05].

Theorem 4.4. (Hierarchical Trace-Correct Decomposition)
Let $(C_i)_{i \in I}$ be a trace-correct decomposition of N, and for some $i' \in I$ let $(C_j)_{j \in J}$ be a trace-correct distributed implementation of $C_{i'}$, where $I \cap J = \emptyset$. Then $(C_k)_{k \in K}$ is a trace-correct decomposition of N, where $K = (I \setminus \{i'\}) \cup J$. $\diamond$

Proof:
Clearly, the components $(C_k)_{k \in K}$ are all output-determinate. We define $C = ||_{i \in I} C_i$, $C' = ||_{j \in J} C_j$ and $C'' = ||_{k \in K} C_k$. Proving $In_{C''} \subseteq In_N$ and $Out_{C''} = Out_N$ is comparatively simple but a bit tedious; a proof can be found in [SV05]. To simplify the notion, e.g. (TC1/C) denotes applying (TC1) for the parallel composition C, and instead of $w|_{C_i}$ we will just write $w|_i$. Also, we treat (TC3) before (TC2). Now let $w \in L(N)$.

(TC1) (TC1/C) implies $w|_C \in L(C)$ and therefore $\forall i \in I : w|_i \in L(C_i)$. In particular, $w|_{i'} \in L(C_{i'})$. Then, by (TC1/C'), $w|_{C'} \in L(C')$ and therefore $\forall j \in J : w|_j \in L(C_j)$. Together, $\forall k \in K : w|_k \in L(C_k)$, and hence $w|_{C''} \in L(C'')$.

(TC3) $w|_k x^{\pm} \in L(C_k)$ for $k \in K$ and $x \in Out_k$. We consider the following two cases.

k $\notin$ J Then, (TC3/C) implies $w|_C x^{\pm} \in L(C)$, and by (TC2/C) $wx^{\pm} \in L(N)$.

k $\in$ J Then, (TC3/C') implies $w|_{C'} x^{\pm} \in L(C')$. By (TC2/$C'$), $w|_{i'} x^{\pm} \in L(C_{i'})$. Applying (TC3) and (TC2) for C in the same way implies $wx^{\pm} \in L(N)$. In both cases, $w|_{C''} x^{\pm} \in L(C'')$ follows with (TC1/C').

(TC2) $w|_{C''} x^{\pm} \in L(C'')$ for $x \in Out_{C''}$. Obviously, $x \in Out_k$ for some $k \in K$ and $w|_k x^{\pm} \in L(C_k)$. Then, $wx^{\pm} \in L(N)$ as just shown in case (TC3).

□

Corollary 4.5. The relation 'trace-correct implementation' (i.e. $\{(N, N') \mid N'$ is a trace-correct implementation of $N\}$) is a pre-congruence for parallel composition. $\diamond$

Proof:
Reflexivity is trivial. Transitivity follows from the above theorem if I and J are singletons. Now pre-congruence follows when considering that in a decomposition a component is replaced by another single component. □

We will use this result for the new correctness proof at the end of this section.

4.2. 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. Actually, as Theorem 4.4 (for $|J| = 1$) suggests, it is sufficient to preserve trace-correctness. However, it is also desirable for a transformation to preserve violation of 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 notions.

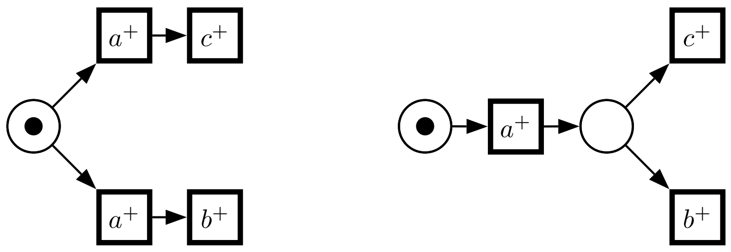


Figure 10. Two LOD-equivalent STGs which are not bisimilar.

Definition 4.6. ($\approx_{lod}$, $\preceq_{tcod}$ and LOD/TCOD-transformations)
Two STGs N and N' are *LOD-equivalent*, denoted $N \approx_{lod} N'$, if

- N and N' are both not output-determinate, or
- N and N' are **l**anguage-equivalent and both **o**utput-**d**eterminate.

They are in the *TCOD relation*, denoted $N \preceq_{tcod} N'$, if

- N and N' are both not output-determinate, or
- N' is a **t**race-**c**orrect implementation of N and both are **o**utput-**d**eterminate.

An STG transformation is a *LOD/TCOD-transformation* if the original and the transformed STG are LOD-equivalent / in the TCOD relation. $\diamond$

Recall that trace-correctness is weaker than language equivalence, since it allows to delete input signals or to have additional occurrences of input edges. Hence, every LOD-transformation is also a TCOD-transformation, but not vice versa.

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. 10. Moreover, any transformation preserving the language and output-determinacy can be made into a LOD-transformation if its domain is restricted to output-determinate systems. Below we list some TCOD-transformations which will be useful for our decomposition algorithm.

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

Definition 4.7. 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'. $\diamond$

List of TCOD-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.

IIC Increasing the concurrency of inputs for deterministic and safe STGs, see Definition 4.9.

The first three transformations in this list always yield a bisimilar STG and thus are LOD-transformations. Below we prove that LOD-SecTC2 and SecTC2' are LOD-transformations and that IIC is a TCOD-transformation (it is not a LOD-transformation since it changes the language). IIC is not intended as a reduction operation, but it may be applied to the final *deterministic* components, where it is sometimes useful for converting speed-independent circuits into delay-insensitive ones [SKC+99]. Observe also that the determinisation of an *output-determinate* STG N is a LOD-transformation. Indeed, iff 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.

Theorem 4.8. If $\overline{N}$ is obtained from some STG N by LOD-SecTC2 or SecTC2', then N and $\overline{N}$ are LOD-equivalent. $\diamond$

Proof:
A secure contraction gives a language equivalent result in any case by Theorem 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, while for SecTC2' (applied to transition t) it 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 $\mathcal{S}$ 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 $u|_{-t}$, obtained by deleting all occurrences of t in u. Thus, we get $M_{\overline{N}}[u|_{-t}\rangle[x^{\pm}\rangle\rangle$ and $M_{\overline{N}}[v|_{-t}\rangle\overline{M}_1$.

Since $\overline{N}$ is output-determinate and $\overline{l}(u|_{-t}) = \overline{l}(v|_{-t})$, 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 $\overline{M}_1[wt'\rangle$, $\overline{l}(t') = x^{\pm}$ 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 $(*)$ it does not share a preset-place with t by assumption of SectTC2'; neither does t'.

We conclude the proof by showing inductively that $M_1[w'\rangle$ with $w'|_{-t} = wt'$. As induction base, we have $M_1[\lambda\rangle$. So assume $M_1[w''\rangle M$ and $\overline{M}_1[w''|_{-t}\rangle\overline{M}$, where $w''|_{-t}$ is a proper prefix of wt' satisfying $\overline{M}_1[w''|_{-t}\rangle\overline{M}$ due to simulation $\mathcal{S}$, and let t_1 be the next transition of wt'. If $M[t_1\rangle M'$ we are done.

It remains to consider the case that $\neg M[t_1\rangle$. We observe that $\overline{M}[t_1\rangle$, 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 positive due to p_0. We check that t can fire m_1 times under M: for all $p \in {}^\bullet t$, we have $M(p) + M(p_1) = \overline{M}((p, p_1)) \geq \overline{W}((p, p_1), t_1) = W(p, t_1) + W(p_1, t_1)$ (where the inequality follows from $\overline{M}[t_1\rangle$), 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 also satisfies the marking equality with $\overline{M}$. By our above considerations and choice of p_1, M'' enables t_1; recall that t_1 needs no tokens from ${}^\bullet t$ and is only disabled because of some missing tokens in $t^\bullet$ – and even the largest of these deficits has been compensated in M''. Thus, $M[t^{m_1} t_1\rangle$. $\square$

The following definition of IIC can only be applied to safe and deterministic STGs; these requirements are not too strict, since – as mentioned above – it is intended for the final deterministic components.

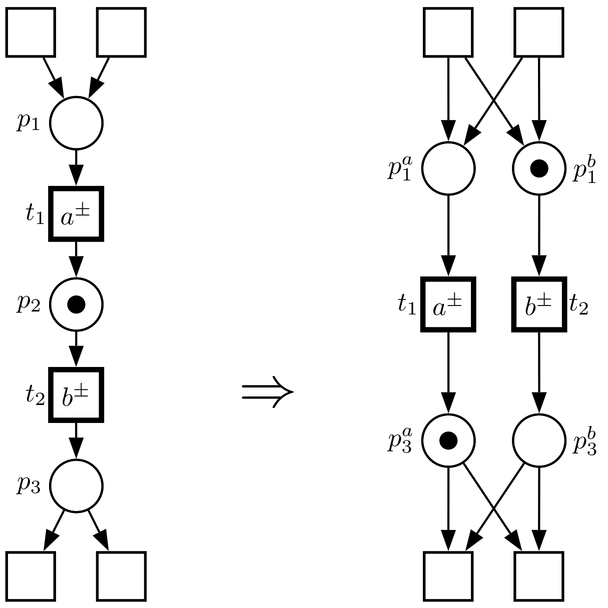


Figure 11. Increasing input concurrency for the case $M(p_2) = 1$.

Definition 4.9. (Increasing Input Concurrency)
Let N be a safe STG and let t_1 and t_2 be two transitions which are labelled with edges of two different input signals a and b. If t_1 is a syntactic trigger of t_2 via a single place p_2 with no other incident arcs, ${}^\bullet t_1 = \{p_1\}$ and $t_2{}^\bullet = \{p_3\}$ with $p_1{}^\bullet = \{t_1\}$ and ${}^\bullet p_3 = \{t_2\}$, *increasing input concurrency (IIC) of* t_1 *and* t_2 results in the net N' defined as follows (cf. also Figure 11):

- $T' = T$ and $l' = l$
- $P' = P \setminus \{p_1, p_2, p_3\} \cup \{p_1^a, p_1^b, p_3^a, p_3^b\}$
- $W'(p,t) = W(p,t)$ and $W'(t,p) = W(t,p)$ for $p \in P \cap P'$ and $t \in T'$,
  $W'(p_1^a, t_1) = W'(p_1^b, t_2) = W(p_1, t_1)$ and $W'(t_1, p_3^a) = W'(t_2, p_3^b) = W(t_2, p_3)$,
  $W'(t, p_1^a) = W'(t, p_1^b) = W(t, p_1)$ and $W'(p_3^a, t) = W'(p_3^b, t) = W(p_3, t)$ for $t \neq t_1, t_2$,
  $W'(x,y) = 0$ otherwise.
- $M_{N'} = iic(M_N)$, where the function $iic : [M_N\rangle \to \mathbb{N}_0^{P'}$ is defined as follows $M' = iic(M)$ if:
  $M'(p) = M(p)$ for $p \in P \cap P'$
  If $M(p_1) = 1$, then $M'(p_1^a) = M'(p_1^b) = 1$ and $M'(p_3^a) = M'(p_3^b) = 0$.
  If $M(p_2) = 1$, then $M'(p_1^a) = M'(p_3^b) = 0$ and $M'(p_3^a) = M'(p_1^b) = 1$.
  If $M(p_3) = 1$, then $M'(p_1^a) = M'(p_1^b) = 0$ and $M'(p_3^a) = M'(p_3^b) = 1$.
  If $M(p_1) = M(p_2) = M(p_3) = 0$ then $M'(p_1^a) = M'(p_1^b) = M'(p_3^a) = M'(p_3^b) = 0$.
  (Note that these four cases are mutually exclusive since N is safe.) $\diamond$

Theorem 4.10. (IIC is a TCOD-transformation)
Let N' be the result of applying IIC for the transitions t_1 and t_2 to a deterministic safe STG N. Then

(1) $\mathcal{S} = \{(M, iic(M)) \mid M \in [M_N\rangle\}$ (i.e. $\mathcal{S}$ and iic coincide) is a transition simulation between N and N'; if $(M, M') \in \mathcal{S}$ and $t \neq t_2$ or $M'(p_1^a) = 0$, then $M'[t\rangle M_1'$ implies $M[t\rangle M_1$ with $(M_1, M_1') \in \mathcal{S}$.

(2) $[M_{N'}\rangle = \mathcal{M} \,\dot\cup\, \mathcal{M}'$, where $\mathcal{M} = \mathcal{S}([M_N\rangle)$ and $\mathcal{M}' = \{M_1' \mid \exists M' \in \mathcal{M} : M'(p_1^a) = 1 \wedge M'[t_2\rangle M_1'$ in $N'\}$.

(3) N' is safe.

(4) N' is deterministic.

(5) N' is a trace-correct implementation of N.

Proof:
Let $l(t_1) = a^{\pm}$ and $l(t_2) = b^{\pm}$ for $a \neq b$.

(1) By definition of $M_{N'}$, $(M_N, M_{N'}) \in \mathcal{S}$, so assume $(M, M') \in \mathcal{S}$ and $M[t\rangle M_1$.

- $t \notin \{t_1, t_2\} \cup {p_3}^\bullet$: then ${}^\bullet t \subseteq P \cap P'$ and, since $M' = iic(M)$, $M'|_{{}^\bullet t} = M|_{{}^\bullet t}$ and $M'[t\rangle M_1'$. Due to definition of W', $M_1'|_{P \cap P'} = M_1|_{P \cap P'}$. If $p_1 \in t^\bullet$, observe that $M(p_1) = 0$ because N is safe, and $M'(p_1^a) = M'(p_1^b) = 0$ by definition of M'; therefore, $M_1(p_1) = 1$ and $M_1'(p_1^a) = M_1'(p_1^b) = 1$. Also, $M'(p_3^a) = M_1'(p_3^a) = M'(p_3^b) = M_1'(p_3^b) = 0$ and then in any case, $M_1' = iic(M_1)$.
- $t \in {p_3}^\bullet$: then $M(p_3) = 1$ (by the safeness of N), $M'(p_3^a) = M'(p_3^b) = 1$ and $M'[t\rangle M_1'$. With similar arguments as above, $(M_1, M_1') \in \mathcal{S}$.

- $t = t_1$: then $M(p_1) = 1$ and $M_1(p_2) = 1$. Therefore, $M'(p_1^a) = M'(p_1^b) = 1$ and $M'(p_3^a) = M'(p_3^b) = 0$. Hence, $M'[t_1\rangle M_1'$ with $M_1'(p_1^a) = M_1'(p_3^b) = 0$ and $M_1'(p_3^a) = M_1'(p_1^b) = 1$. Clearly, $M_1'|_{P\cup P'} = M_1|_{P\cup P'}$ and $(M_1, M_1') \in \mathcal{S}$.
- $t = t_2$: similar to the case $t = t_1$.

Now assume $M'[t\rangle M_1'$. A similar case analysis shows that $t \neq t_2$ or $M'(p_1^a) = 0$ implies $M[t\rangle M_1$ with $(M_1, M_1') \in \mathcal{S}$.

(2) Since $(M, M') \in \mathcal{S}$ implies $M \in [M_N\rangle$, we have $\mathcal{M} \subseteq [M_{N'}\rangle$ by (1); this in turn implies that $\mathcal{M}' \subseteq [M_{N'}\rangle$, too.

Due to (1) and $M_{N'} \in \mathcal{M}$, a marking in $[M_{N'}\rangle \setminus \mathcal{M}$ can only be reached via firing t_2 from a marking which marks p_1^a, i.e. such a marking is in $\mathcal{M}'$. Now take $M_1' \in \mathcal{M}'$, i.e. $\exists (M, M') \in \mathcal{S}$, $M'[t_2\rangle M_1'$ and $M'(p_1^a) = 1$. This implies $M(p_1) = 1$ in N, and $M_1'(p_3^b) = 1$, $M_1'(p_3^a) = M_1'(p_1^b) = 0$ and $M_1'|_{P\cap P'} = M_1|_{P\cap P'}$ in N'.

We now consider all markings reachable directly from M_1'. Clearly, $M_1'[t_1\rangle M_2'$ with $(M_2, M_2') \in \mathcal{S}$ for $M_1[t_1t_2\rangle M_2$, i.e. $M_2' \in \mathcal{M}$. Let now $M_1'[t\rangle M_3'$ with $t \neq t_1$. We have $t \neq t_2$ due to $M_1'(p_1^b) = 0$ and $t \notin p_3{}^\bullet$ due to $M_1'(p_3^a) = 0$. Therefore, ${}^\bullet t \cap ({}^\bullet t_2 \cup t_2{}^\bullet) = \emptyset$ in N', and t is activated concurrently to t_2 under M': $M'[t\rangle M_4'[t_2\rangle M_3'$ and $M_4'(p_1^a) = M'(p_1^a) = 1$. Additionally, (1) implies $M[t\rangle M_4$ and $(M_4, M_4') \in \mathcal{S}$. Thus, $M_3' \in \mathcal{M}'$ implying $[M_{N'}\rangle \subseteq \mathcal{M} \cup \mathcal{M}'$, which proves the claim.

(3) Follows from (2) when considering the properties of $\mathcal{M}$ and $\mathcal{M}'$.

(4) For $M' \in [M_{N'}\rangle$, $M'[t\rangle$, $M'[t'\rangle$ and $t \neq t'$ we will show that $l(t) \neq l(t')$; the claim is obvious for the case $\{t, t'\} = \{t_1, t_2\}$, and in what follows we assume that $\{t, t'\} \neq \{t_1, t_2\}$ $(*)$.

Let $M' \in \mathcal{M}$. Therefore, $(M, M') \in \mathcal{S}$ for some M. If $t_2 \notin \{t, t'\}$ or $M'(p_1^a) = 0$, then (1) implies $M[t\rangle$ and $M[t'\rangle$, and we are done since N is deterministic. Let w.l.o.g. $t = t_2$ and $M'(p_1^a) = 1$; thus, $M(p_1) = 1$ and $M'(p_1^b) = 1$. Hence, $M[t_1\rangle M_1[t_2\rangle$. On the other hand, we have $M[t'\rangle$ due to (1) and $t' \neq t_1$ by $(*)$. Together, this gives $M_1[t'\rangle$ and again the claim follows from the determinism of N.

Let $M' \in \mathcal{M}'$. Due to (2), there is a marking $M'' \in \mathcal{M}$ with $M''[t_2\rangle M'$ and $M''(p_1^a) = 1$. With (3), we conclude $t_2 \notin \{t, t'\}$ and $M''(p_3^a) = M'(p_3^a) = 0$. The latter shows that $p_3 \notin {}^\bullet t \cup {}^\bullet t'$ and thus $M''[t\rangle$ and $M''[t'\rangle$. Hence, there is a marking M of N with $M[t\rangle$ and $M[t'\rangle$, which proves the claim.

(5) Let $M_{N'}[v\rangle\rangle M$ for some $v \in (Sig\pm)^*$. Since N is deterministic, there is a unique transition sequence u with $l(u) = v$. Due to (1), $M_{N'}[u\rangle M'$ with $(M, M') \in \mathcal{S}$, and obviously $M_{M'}[v\rangle\rangle M'$, which proves (TC1).

If $M'[x\pm\rangle\rangle$ for some $x \in Out_{N'}$, clearly this is due to $M'[t\rangle$ with $t \neq t_2$. Then (1) implies $M[t\rangle$ and therefore $M[x\pm\rangle\rangle$.

□

Theorem 4.10(4,5) shows that IIC is indeed a TCOD-transformation. Furthermore, this result can easily be extended to more general cases:

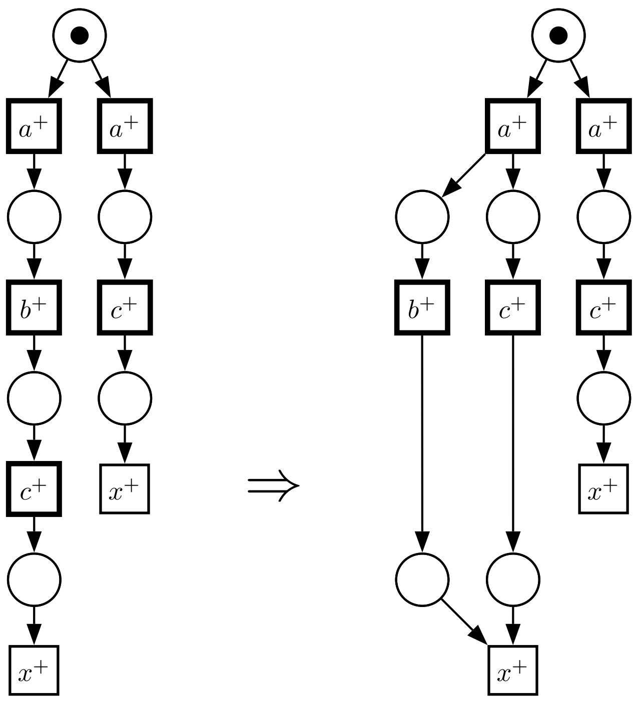


Figure 12. Counterexample for application of IIC to an output-determinate but non-deterministic STG. The STG on the left hand side is output-determinate, but the STG resulting from applying IIC for b^+ and c^+ is not, i.e. IIC is not a TCOD-transformation in this case.

- Three or more sequential transitions $t_1, \dots, t_n$ (each labelled with a different input) can be made concurrent with a similar construction.
- It is also possible to allow more than one place in the preset of t_1 and the postset of t_n.
- A bit surprisingly, t_1 could also be labelled with an output signal.

For the first two cases, the proof structure stays the same – the proof gets only slightly more complicated. In the last case, the proof stays the same since it never used the fact that t_1 is labelled with an input.

Regarding termination, IIC can only be applied finitely many times to a final component: consider the place p_2 'in the middle', which only has a single transition in its pre- and in its postset; IIC reduces the number of such places. Observe that IIC cannot be applied to the new places (p_1^a, p_1^b, p_3^a and p_3^b).

Finally, observe that N has to be deterministic rather then output-determinate as the counterexample in Figure 12 demonstrates.

4.3. The Decomposition Algorithm

Now, we describe the extended version of the STG decomposition algorithm of [VW02, VK06], which uses the TCOD-transformations of the previous section. 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 their original signals 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 TCOD-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 TCOD-transformations. This way, we hopefully make the component STGs small enough for the output-determinacy check and subsequent synthesis to be feasible. In an *optimistic strategy*, one performs TCOD-transformations as long as possible – with our list of TCOD-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 TCOD-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 TCOD-transformations are undone.

The algorithm of this paper is a generalisation of the decomposition algorithm in [VK06], 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 just for technical reasons, e.g. because it was on a loop or had an arc with the weight greater than 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 contracted successfully. This can make the reachability graph much larger, while from the perspective of circuit synthesis 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, VK06] – 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 [VK06] is guaranteed to find a correct decomposition without any final check.

In another version discussed in [VK06], 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 for this version 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 following.

4.4. 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 4.7) 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.

Lemma 4.11. 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. $\diamond$

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 $w|_C \in L(C)$.

(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 $w|_{C_j} x^\pm \in L(C_j)$ due to the firing sequence vt with $l(t) = x^\pm$. Since $l_j(v) = w|_{C_j} = l_j(u)$ 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 $w|_C x^\pm$ is a trace of C.

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

Lemma 4.11 will be used as the induction base for our main theorem, which states the correctness of the new decomposition algorithm.

Lemma 4.12. If an STG N is not output-determinate, in every initial decomposition $(C_i)_{i\in I}$ some C_i is not output-determinate as well. $\diamond$

Proof:
Suppose that $M_N[wx^\pm\rangle\rangle$ and $M_N[w\rangle\rangle M$ in N with $x \in Out_j$. Then $M_{C_j}[w|_{C_j} x^\pm\rangle\rangle$ and $M_{C_j}[w|_{C_j}\rangle\rangle M$ 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. $\square$

Theorem 4.13. (Correctness)
Consider the application of the decomposition algorithm to an STG N.

(1) If only the TCOD-transformations from the list in Section 4.2 are applied, the algorithm terminates.

(2) 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.

(3) If the algorithm reports that N is not output-determinate, then this is the case. $\diamond$

Proof:

(1) This is essentially a result from [VK06]: 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.

Recall that IIC is only applied to the final components after determinisation and not mixed up with the other operations; as it was argued above, it can be applied only finitely often then.

(2) 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 4.12, one of them would not be output-determinate, and this would be preserved by the TCOD-transformations, contradicting our hypothesis.

This consideration also implies that all mentioned initial components are output-determinate; hence, by Lemma 4.11 this initial decomposition $(C_i)_{i\in I}$ is a trace-correct distributed implementation of N. Furthermore, due to the definition of TCOD-transformations, each final component is a trace-correct implementation of its corresponding initial component. Theorem 4.4 then implies also that the set of final components is a trace-correct distributed implementation of N.

Also determinisation of a C_i is a TCOD-transformation, and the third claim about the deterministic automatons $(DA(C_i))_{i\in I}$ follows from Theorem 4.3.

(3) The algorithm reports that N is not 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. $\square$

It should also be noted that for a consistent N only consistent components are produced, cf. [VK06]. Compared to the approach of [VK06], 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 [VK06] (at the expense of requiring N to be deterministic and prohibiting dummies in the final components). Additionally, the proof in [VK06] takes care to show that, for deterministic specifications, type-2 secure contractions can be applied without restriction. Since we use the same transformations as in [VK06], 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. Experiments

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] (denoted TREEOLD here). TREEOLD 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 s can be contracted, the contraction of s is postponed to later stages of the algorithm, which is detrimental for performance due to the decreased usability of intermediate STGs. Note that *all* the transitions of s are postponed, even if only *one* of them cannot be contracted, because backtracking is performed for signals rather than individual transitions. For practical STGs however, most of the postponed signals can actually be contracted at later stages of the algorithm.

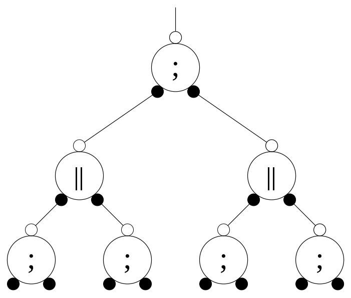
The new approach (TREENEW) can avoid such postponing of signals under certain circumstances: if in an intermediate STG a transition of a signal s cannot be contracted due to a new structural auto-conflict, postponing for s is performed as in TREEOLD. But if all transitions are non-contractible due to technical reasons only (e.g. if they are not safeness-preserving), no backtracking is performed and the remaining non-contractible transitions are left as dummies in the intermediate STG. As mentioned above, most of them will be contracted at later stages, and otherwise they will remain in the final component.

We applied TREENEW to a number of benchmarks constructed from two basic BALSA-inspired handshake components (cf. [EB02]) also used before by [CC06]: 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 Fig. 13 (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. We also worked with other benchmarks built from handshake components (e.g. trees of parallelisers only); the results did not differ much, so we consider here only SEQPARTREE-N.

In contrast to the decomposition method of [CC03,CC06], we allow components with more than one 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 ideas for the automatic detection of suitable partitions, see [SVWK06].

We applied four variants of tree decomposition to these benchmarks, as well as stand-alone synthesis with PETRIFY and MPSAT. (The tool for CSC conflict resolution and decomposition presented 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.

TREEOLD is compared with TREENEW for ordinary contractions as described in Section 4.3 as well as for *safeness-preserving contractions*, i.e. contractions which do not destroy the safeness of the STG. (This kind of contractions 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 increases the runtime. This resulted in the mentioned four series of experiments, see Table 1.



	Size	Signals
Benchmark	$\lvert P\rvert$ – $\lvert T\rvert$	$\lvert In\rvert$ – $\lvert Out\rvert$
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 13. **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 the end, the final components were synthesised (which includes the resolution of CSC conflicts) with PETRIFY, which was possible for every component. As a consequence, this shows that the decomposition is correct: a necessary condition for synthesis is the absence of CSC conflicts. However, a violation of output-determinacy is a special case of a CSC conflict, *which cannot be resolved* as discussed in the following section. Hence, the resulting components are indeed output-determinate, and Theorem 4.13(1) guarantees the correctness. Moreover, the resulting components turn out to be the same for all series, and hence the synthesis times are given only once. Observe that the latter property makes these benchmarks especially useful for comparing the four variants of the algorithm presented in Table 1.

The synthesis with stand-alone PETRIFY or MPSAT did not terminate 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 less than 11 minutes. One can see that leaving non-contractible transitions as dummies in the intermediate STGs is useful, especially for safeness-preserving contractions. The reason is that, in this variant, the decomposition algorithm encounters more λ-transitions which are non-contractible due to technical reasons (viz. they do not preserve safeness). TREEOLD would backtrack and postpone for the respective signals, which significantly increases the runtime of this approach. As one can see, the new approach leaving such transitions as dummies in the intermediate STGs is much faster.

6. Output-Determinacy and Internal Signals

Up to now, we considered STGs without *internal signals*, i.e. signals which are produced by the circuit but are not visible to the environment. Usually, such signals are introduced automatically into the STG during the synthesis process, mainly in order to resolve encoding conflicts, but also to perform logic decomposition (i.e. splitting large gates into smaller ones) or – as a recent application – to preserve speed-independence during decomposition [SVWW08].

	Safe		Non-Safe		
Benchmark	TREENEW	TREEOLD	TREENEW	TREEOLD	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 for 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.

There are several possible interpretations for internal signals, depending on the role the STGs plays. Though an STG is always a specification of an asynchronous circuit, it is common for an STG to go via a series of refinements, until eventually the ‘final’ STG is produced from which the gate-level netlist is synthesised. Hence, the ‘distance’ from the STG to the circuit netlist can vary; in particular, in the context of decomposition, the specification STG is ‘far’ from the final circuit, while the component STGs are ‘close’ to it.

As a consequence, a far-off specification should only describe the *external* behaviour of a circuit (i.e. its interface to the environment) rather than details of the physical implementation. For this purpose internal signals are not needed, and so a specification STG should not contain them. (If it does contain them, they can be treated like a designer’s suggestion; in particular, the synthesis tool is free to turn them into dummies, as they are anyway ignored by the environment.) On the other hand, for the ‘final’ STG the internal signals are useful and can be mapped to physical wires. Hence, for this STG, it makes sense to consider them as outputs of the circuit, which (unlike dummies) occur in traces and are a part of the state encoding.

The semantics of internal transitions (i.e. whether they are treated as dummies or as outputs) is important for the definition of output-determinacy; indeed, whether the STG is output-determinate or not may depend on the chosen semantics. As described above, we choose to treat internal transitions as dummies in the specification STG and as outputs in the implementation STG. Such a treatment might be seen as somewhat unusual, particularly considering the internal transitions in the specification as dummies. However, we argue that it is reasonable, as considering these transitions as outputs leads to undesirable situations where an STG is not output-determinate, but still implementable by a deterministic one, as illustrated in Fig. 14. On the other hand, the proposed treatment allowed us to lift Proposition 3.3, stating that only output-determinate STGs can be deterministically implemented, to the case of STGs with internal signals.

Formally, 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 $Int \cap (In \cup Out) = \emptyset$, and l is extended accordingly. We will denote by $Ext = In \cup Out$ the set of *external* signals of N.

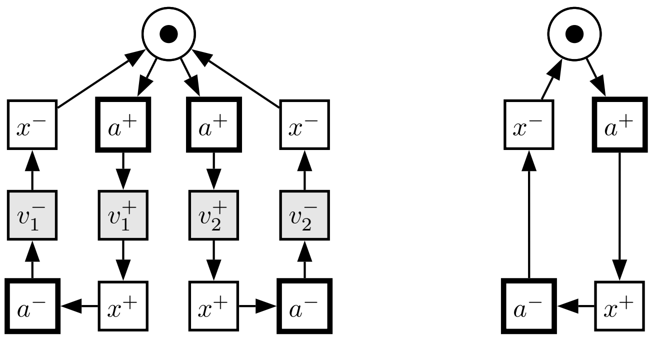


Figure 14. Counterexample: the non-output-determinate STG on the left with internal signals v_1, v_2 is implemented by the STG on the right.

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 6.1. (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:

(IC1) There is a trace v of C such that $M_C[v\rangle\rangle$ with $v|_{Ext_C} = w|_{Ext_C}$.

For every trace v of C such that $M_C[v\rangle\rangle M'$ with $v|_{Ext_C} = w|_{Ext_C}$:

(IC2) If $a \in In_N$ and $M[a^\pm\rangle\rangle$, then either $M'[a^\pm\rangle\rangle$ or $a \notin In_C$.

(IC3) If $x \in Out_N$, then $M[x^\pm\rangle\rangle$ iff $M'[v_C x^\pm\rangle\rangle$ for some $v_C \in (Int_C^\pm)^*$. $\diamond$

In this definition, the items (IC1)/(IC2)/(IC3) correspond to (C1)/(C2)/(C3) from Definition 3.1 with the following differences: clearly, every trace of the specification must be possible in the implementation. However, now the implementation might produce this trace with the help of internal signals. Hence, in (IC1) we just require that these traces coincide externally, and although C is deterministic, there is the possibility that different traces look externally equal and that a trace v of N can be matched in different ways by C. Observe that the implementation is still allowed to have fewer inputs than the specification.

Since internal signals are introduced for technical reasons like resolution of CSC conflicts, the resulting components are not to be considered as a specification but rather like a hardware-close implementation. Therefore, the internal signals should be treated like invisible outputs. This has two consequences for the handling of inputs in (IC2), resulting from the fact that the environment cannot observe the internal signals of C: inputs of the environment are produced whenever a corresponding external trace has occurred, no matter in which state the implementation is. Therefore, the implementation must be ready to receive an activated input in *all* states corresponding to an external trace, i.e. in the corresponding STG an input cannot be triggered by an internal transition. (While the latter condition is common and is also needed to guarantee speed-independence, the former condition differs from the handling of inputs in the definition of output-determinacy.) In contrast, outputs can be preceded by internal signals, because the environment will wait for the circuit until the output is produced.

Further justification of soundness and usefulness of these assumptions can be found in [SV05] in the discussion after the correctness notion with internal signals.

Violation of output-determinacy always results in a CSC-conflict, because if a trace can be executed in two different ways, the reached states obviously have the same state vector. Below we show that this special conflict cannot be resolved by insertion of internal signals. In fact, we prove a more general statement: if a specification is not output-determinate, it cannot be implemented by a deterministic STG with internal signals. Thus, if an STG is not output-determinate, then the result of a behaviour-preserving insertion of internal signals is also not output-determinate and still has CSC conflicts. The result below is an extension of Proposition 3.3 to implementations with internal signals.

Proposition 6.2. Let N be an STG without internal signals and C (with internal signals) be a correct implementation of N. Then N is output-determinate. $\diamond$

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 (IC1) of Definition 6.1, we get $M_C[v\rangle\rangle M'$ for some v such that $v|_{Ext_C} = w|_{Ext_C}$, and thus, $M'[v'x^{\pm}\rangle\rangle$ for some $v' \in (Int_C^{\pm})^*$ by $M_1[x^{\pm}\rangle\rangle$ and (IC3). Therefore, by (IC3), $M_2[x^{\pm}\rangle\rangle$. $\square$

7. 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 a specification is ill-formed if it is not output-determinate, 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, VK06]. 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 TCOD-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 of BALSA specifications, as mentioned in the introduction. An approach to re-synthesis using decomposition can be found in [SVWW08].

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. IEEE Computer Society Press, 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.

[KKTV94] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. John Wiley & Sons Ltd., 1994.

[KS07] V. Khomenko and M. Schaefer. Combining decomposition and unfolding for STG synthesis. In *ICATPN '07: 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, pages 223–243, 2007.

[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.

[Sch07] M. Schaefer. DESIJ - a tool for decomposition. Technical Report 2007-11, University of Augsburg, 2007.
URL: `http://www.informatik.uni-augsburg.de/de/forschung/reports/`.

[SKC$^+$99] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno, and A. Yakovlev. What is the cost of delay insensitivity? In *Proc. CAD'99*, pages 316–323. IEEE Computer Society Press, 1999.

[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 *ACSD*, pages 123–132, 2006.

[SVWW08] M. Schaefer, W. Vogler, D. Wist, and R. Wollowski. Avoiding irreducible CSC conflicts by internal communication. Technical Report 2008-02, University of Augsburg, 2008.
URL: `http://www.Informatik.Uni-Augsburg.DE/skripts/techreports/`.

[VK06] W. Vogler and B. Kangsah. Improved decomposition of signal transition graphs. *Fundamenta Informaticae*, 76:161–197, 2006.

[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.