Output-Determinacy and Asynchronous Circuit Synthesis

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

Mark Schaefer and Walter Vogler
Institute of Computer Science
University of Augsburg, Germany
{schaefer,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. With our theory we improve an STG decomposition algorithm, which can alleviate state explosion.*

Keywords: *output-determinacy, decomposition, asynchronous circuits, STG, OR-causality.*

1. Introduction

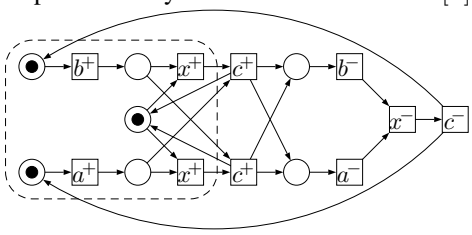
Asynchronous circuits are a promising type of digital circuits. They have lower power consumption and electromagnetic emission, no problems with clock skew and related subtle issues, and are fundamentally more tolerant of voltage, temperature and manufacturing process variations [2]. The International Technology Roadmap for Semiconductors report on Design [11] predicts that 22% of the designs will be driven by handshake clocking (i.e., asynchronous) in 2013, and this percentage will raise up to 40% in 2020.

In this paper we are concerned with an important subclass of asynchronous circuits, called *speed-independent* circuits, i.e., circuits which work correctly regardless of their gates' delays (the wires are assumed to have negligible delays). *Signal Transition Graphs (STGs)* [7] are a formalism for specification of such circuits. They are interpreted Petri nets where transitions are labelled with rising and falling edges of circuit signals.

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

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

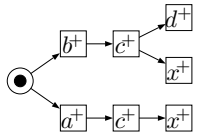
OR-causality (see [21]) When modelling with a safe Petri net that a system has to respond to any of several possible stimuli in the same way, non-determinism naturally arises. The figure below illustrates OR-causality (the 'interesting' part is framed): a^+ and b^+ are concurrent inputs, and x^+ is produced upon arrival of either of them. The x^+-transitions are in dynamic conflict, i.e., the STG is non-deterministic, but it is implemented by the deterministic circuit $[x]{=}a \vee b$.



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

A more practical example of OR-causality is a dual-rail adder cell that can determine its carry-out as soon as two of its three inputs signal the same value [13].

Hiding of signals Non-determinism naturally arises when in a deterministic specification some signals are hidden, see Figure 1. In fact, hiding of signals is an essential part of the decomposition algorithm of [19,20], which we will improve in the present paper.



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

Figure 1. After hiding a and b, the STG is non-deterministic, but is implemented by $[x]{=}\,c$ (input d can be ignored).

To the best of our knowledge, no satisfactory formal semantics of non-deterministic STGs and, in particular, of dummy transitions[1] has been given so far – as we show be-

[1] In practical STGs, the designers intuitively avoid using dummies am-

low, the language is no longer 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 [16] for the concept of determinacy.

As an important application of our theory of output-determinacy, we will generalise the decomposition algorithm of [19, 20]. Below we describe how decomposition fits into the design flow for synthesising asynchronous circuits from STGs.

PETRIFY [8, 9] is one of the commonly used tools for synthesis of asynchronous circuits from STGs. 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 synthesisable circuits, which are often restrictive, especially if the specification is not constructed manually by a designer but rather generated automatically from high-level hardware descriptions. (For example, designing a control circuit with more than 20–30 signals with PETRIFY is often impossible.) Hence, this approach does not scale.

To cope with the state space explosion problem, Chu suggested a nondeterministic method for decomposing an STG into several smaller ones [7], see also [21]. The idea is that all components together can be synthesised faster than the original STG while the corresponding circuits perform together in the same way as the circuit directly synthesised from the specification. While there are strong restrictions on the structure and labelling of STGs in [7], the improved decomposition algorithm of Vogler, Wollowski and Kangsah [19, 20] works under – comparatively moderate – restrictions on the labelling only. In these previous papers, the specifications had to be deterministic; here, we generalise this to output-determinate specifications.

Our approach also allows to make the decomposition algorithm more efficient. Each component is obtained from the original STG by hiding some of the signals in it, and then contracting the corresponding transitions. The success of this algorithm depends on the ability to contract all such transitions in a behaviour-preserving way. If this is not possible, the algorithm of [19] has to *backtrack* and reintroduce some of the signals into the component, even if they are not really needed for implementation. In our new version of the algorithm, one can leave such non-contracted hidden transitions in the component and proceed with synthesis for a component with fewer signals, which was obtained in a shorter time. While previously the components were deterministic and correct by construction, our components can be non-deterministic; to guarantee correctness, they have to be checked for output-determinacy in the end.

Another way to cope with the state space explosion problem is to use *syntax-directed* translation of the specification to a circuit, thus avoiding to build the state space; cf. BALSA [10] and TANGRAM [1]. This technique, although computationally efficient, often yields circuits with large area and performance overheads compared with synchronous counterparts, since the resulting circuits are highly over-encoded.

For asynchronous circuits to be competitive, one has somehow to combine the advantages of logic synthesis (high quality of circuits) and syntax-directed translation (guarantee of a solution, efficiency) while compensating for their disadvantages. A natural way of doing this is to apply logic synthesis to the control path extracted from a BALSA specification. This control path can be partitioned into smaller 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 [5] showed that this combined approach can half the area devoted to control flow and improve its latency.

The approach of [5] uses an integer linear programming (ILP) technique to resolve *CSC conflicts* (see [9]) in the specification; the resulting STG is decomposed into smaller components such that they are also free of CSC conflicts, as described in [4]. 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.

Our decomposition algorithm follows a more scalable approach, which avoids performing expensive operations 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 [5] 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 of CSC conflicts. If a component has a CSC 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 in the component. The technique described in [14] 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 conflicts. Finally, before synthesis, the remaining CSC conflicts are resolved in each component.

The paper is organised as follows: in the next section we introduce the basic concepts of Petri nets and STGs. In Section 3, the new notion of output-determinacy is intro-

biguously. However, ambiguous situations do exist, in particular when firing a dummy transition can disable other transitions.

duced and justified; we give a list of semantics-preserving transformations, and we analyse the complexity of checking output-determinacy. In the following section, we present our STG-decomposition algorithm and state its correctness. We close with some initial experimental results and a conclusion. The proofs and further details can be found in the technical report [15].

2. Basic definitions

A *Petri net* is a 4-tuple $N = (P, T, W, M_N)$ where P is a finite set of *places* and T a finite set of *transitions* with $P \cap T = \emptyset$. $W : P \times T \cup T \times P \to \mathbb{N}_0$ is the *weight function* and M_N is the *initial marking*. In addition to the standard rules about drawing nets, the following short-hand notation is used: a transition can be connected directly to another transition if the place 'in the middle of the arc' has exactly one incoming and one outgoing arc. We assume that the reader is familiar with the notions *marking, preset* ${}^\bullet x$*, post-set* $x^\bullet$*, firing rule for transitions* ($M[t\rangle M'$) *and sequences, reachability, boundedness* and *safeness*.

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 $Sig \stackrel{\text{df}}{=} In \cup Out$ 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* to *logical high* , and a minus sign denotes the other direction. We write $s^\pm$ if it is not important or unknown which direction takes place; if such a term appears more than once in the same context, it always denotes the same direction. To keep the notation short, input/output signal edges are just called input/output edges. An STG may contain transitions labelled with the empty word λ, which do not correspond to any signal change; they are called *dummy*-transitions.

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 – since λ is the empty word, e.g., $M[s^\pm\rangle\rangle M'$ 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 N if $M_N[v\rangle\rangle$. The *language* $L(N)$ of N is the set of all traces of N.

Often, nets are considered to have the same behaviour if they are language equivalent. Another, more detailed behaviour equivalence is *bisimilarity* [16]. Intuitively, *bisimilar* STGs can work side by side such that in each stage each STG can simulate the signals of the other.

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 $l(t_1) = l(t_2) \neq \lambda$.

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

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

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

The parallel composition of STGs N_1 and N_2 is defined if $Out_1 \cap Out_2 = \emptyset$. Let $A = Sig_1 \cap Sig_2$ be the set of common signals. If e.g., s is an output of N_1 and an input of N_2, then an occurrence of an edge $s^\pm$ in N_1 is 'seen' by N_2, i.e., it must be accompanied by an occurrence of $s^\pm$ in N_2. Since we do not know a priori which $s^\pm$-labelled transition of N_2 will occur together with some $s^\pm$-labelled transition of N_1, we have to allow for each possible pairing. Thus, the *parallel composition* $N = N_1 \parallel N_2$ is obtained from the disjoint union of N_1 and N_2 by combining each $s^\pm$-labelled transition of N_1 with each $s^\pm$-labelled transition from N_2 if $s \in A$.

Since composition is associative and commutative up to

isomorphism, 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_is. Since the place set of the composition is the disjoint union of the place sets of the components we can consider markings of the composition (regarded as multisets) as the disjoint union of markings of the components and write a marking of the composition $(C_i)_{i\in I}$ as a tuple $(M_1, \ldots, M_n)$ if M_i is a marking of C_i for $i \in I = \{1, ..., n\}$. The formal definition and an example can be found e.g., in [20].

We now introduce transition contraction (see e.g., [3] for an early reference), which will be most important in our decomposition procedure; see [19] for further discussions. In an STG N, a λ-labelled transition t can be contracted, yielding the STG $\bar{N}$, if ${}^\bullet t$ and $t^\bullet$ are disjoint and t is not adjacent to arcs with weight greater than 1. The transition t is deleted and ${}^\bullet t$ and $t^\bullet$ are replaced by their Cartesian product; cf. Figure 2. Every new place (p, p') inherits the tokens and the connections to other transitions (except t) from p and p'. A contraction is called *type-1 secure* (or just *type-1*) if $({}^\bullet t)^\bullet \subseteq \{t\}$, or *type-2 secure* (or just *type-2*) if ${}^\bullet(t^\bullet) = \{t\}$ and $M_N(p) = 0$ for some $p \in t^\bullet$.

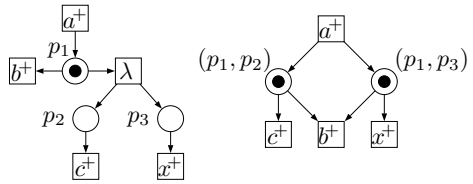


Figure 2. Before and after contraction of the λ-transition.

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*. In Figure 2 the b^+- and the c^+-labelled transition form a new conflict pair.

Another operation that can be used in our decomposition algorithm is the deletion of redundant transitions and places. A *redundant transition* is a λ-transition t, where either each place $p \in {}^\bullet t \cup t^\bullet$ forms a loop with t with two arcs of the same weight (t is a *loop-only transition*) or some other λ-transition has arcs to and from the same places with the same weight as t (which is a *duplicate transition*). In this paper, we also speak of a duplicate transition if the two transitions in question are labelled with the same signal edge. *(Structurally) redundant places* are defined e.g., in [3].

These deletions and type-1 contractions preserve the behaviour in a strong sense:

Proposition 2.1 *If N' is obtained from an STG N by deleting a redundant transition or place or by a type-1 contraction then N and N' are bisimilar.*

3. Output-determinacy

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

For the class of output-determinate STGs we show that their language is an adequate semantics, and re-formulate the notion of correct implementation purely in terms of the language. As an important application, output-determinacy plays an important role in our STG decomposition algorithm described in Section 4 and as part of the invariant in its correctness proof. Moreover, we introduce a set of semantics-preserving STG transformations, which are used in our decomposition algorithm. This set can easily be extended since the definition of 'semantics-preserving' is simple. We also give a result about the computational complexity of checking output-determinacy.

3.1. Correct implementations

An STG N specifies the behaviour of a system in the sense that the system must provide *all and only* the specified outputs and that it must allow *at least* the specified inputs. As a consequence, the system must be able to perform at least all traces of N. In fact, N also describes assumptions about the environment the system will interact with; namely, the environment will only produce the inputs specified by N. A correct implementation of N may allow additional inputs, but these inputs and subsequent behaviour will never occur in the envisaged environment. In other words, when the system runs in a proper environment, only traces of N 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 Figure 1, inputs a, b and d are irrelevant for producing x and can be ignored).

The following definition assumes a deterministic implementation (as it is the case in circuit design), but the specification can be non-deterministic. The projection of a trace w

of N onto the signals of C, obtained by deleting all signal edges where the signal belongs to $In_N \setminus In_C$, is denoted by $w\big|_C$.

Definition 3.1 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 M with $M_N[w\rangle\rangle M$ the following hold:

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

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

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

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

3.2. The notion of output-determinacy

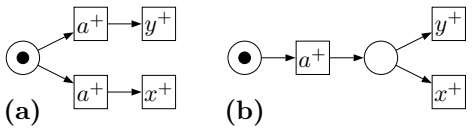
A non-deterministic specification can perform the same trace in two different ways, reaching different states M_1 and M_2. In the speed-independent context the only information available to the circuit is the execution history, i.e., the trace performed,and so an implementation cannot know if its current 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*. In such a situation, *determinacy* [16] implies that both states enable the same traces, i.e., they are indistinguishable at the level of language.

Our definition of correctness (cf. Definition 3.1) requires that the implementation must provide *exactly* the *outputs* enabled by M_1 and exactly the outputs enabled by M_2. This is only possible if M_1 and M_2 enable the same outputs. In contrast, the implementation must allow *at least* the inputs enabled under M_1 and the inputs enabled under M_2; this is very well possible even if these sets of inputs differ – i.e., the implementation may allow the union of these sets or any of its supersets. This observation leads to our central notion of output-determinacy.

Definition 3.2 An STG N is called *output-determinate*[2] if $M_N[w\rangle\rangle M_1$ and $M_N[w\rangle\rangle M_2$ implies for every $x \in Out_N$ that $M_1[x^{\pm}\rangle\rangle$ iff $M_2[x^{\pm}\rangle\rangle$. ◇

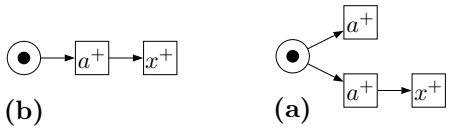
For example, the STG in Figure 1 is output-determinate after hiding a and b. Clearly, a deterministic STG is also output-determinate; note also that – in contrast to a deterministic STG – an output-determinate STG may contain λ-transitions.

[2]This notion is not to be confused with the notions of determinacy and output determinism in [12,18]; the latter two notions are related to (output) persistency (also called (semi-)modularity), while our notion describes a specific kind of non-determinism in the sense of automata theory.



input: a **outputs:** x, y

Figure 3. A semi-modular but not output-determinate STG **(a)** and the non-semi-modular STG (due to the choice between the outputs x^+ and y^+) obtained by determinisation **(b)**.



input: a **outputs:** x

Figure 4. A non-output-determinate STG **(a)**. After determinisation **(b)** it is implementable, but not correct w.r.t. the original specification, since it can cause a failure in the environment by producing x^+ when it is not expected.

Theorem 3.3 *For safe or bounded STGs, checking output-determinacy is* PSPACE*-complete; the complexity is the same if the STG is known to be consistent.*

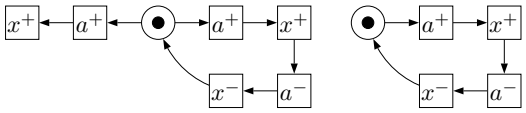
A practical test for safe or bounded divergence-free STGs can be found in [15].

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-semi-modularity [9], as illustrated in Figure 3; Figure 4 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. A non-output-determinate specification of a speed-independent system cannot be implemented deterministically and thus is ill-formed.



inputs: a; **outputs:** x

Figure 5. An output-determinate STG with a deadlock **(left)** and the deadlock-free STG obtained by determinisation **(right)**. The execution of x^- is still correct: it only occurs when the environment signals with a^- that it is in the 'right' branch of N. The circuit $[x] = a$ implements both STGs.

This rule can be justified by the following result.

Proposition 3.4 *Let C be a correct implementation of N. Then N is output-determinate.*

Observe that a non-output-determinate STG always has CSC conflicts: according to Definition 3.2, any violation of output-determinacy implies the presence of two states which can be reached by the same trace (and thus have the same encoding) and enable different sets of outputs. It can be shown that such a CSC conflict is *irreducible*, i.e., it cannot be resolved by the insertion of *internal signals* into the STG (as performed e.g., by PETRIFY or MPSAT [6]) 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. Further explanations and a proof can be found in [15].

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

The proposed semantics has interesting consequences, e.g., a deadlock-free specification can be equivalent to one with deadlocks, as illustrated in Figure 5. Hence, arbitrary language-preserving transformations of output-determinate specifications are allowed, as long as the resulting STG is still output-determinate. We discuss valid transformations in Section 3.4.

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

Definition 3.6 An output-determinate STG C is a *trace-correct implementation* of an output-determinate STG N if $In_C \subseteq In_N$, $Out_C = Out_N$, and for every trace w of N the following hold:

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

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

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

Proposition 3.7 *Let N be an output-determinate STG and C be a deterministic STG such that $In_C \subseteq In_N$ and $Out_C = Out_N$. Then C is a correct implementation of N iff it is a trace-correct implementation of N.*

3.4. Valid STG transformations

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

Definition 3.8 Two STGs N and N' are *LOD-equivalent*[3] if either N and N' are both non-output-determinate, or N and N' are language-equivalent and both output-determinate. An STG transformation is an *LOD-transformation* if the original and the new STG are LOD-equivalent. ◇

One can observe that any transformation yielding a bisimilar STG is an LOD-transformation, but there are LOD-transformations which yield a non-bisimilar STG. Below we list some LOD-transformations which will be useful for our decomposition algorithm.

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

Definition 3.9 For transitions t, t', 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'.

[3] LOD stands for Language and Output Determinacy.

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 LOD-transformations

RedPD Deletion of a redundant place
RedTD Deletion of a redundant transition
SecTC1 Type-1 contraction of a λ-transition
LOD-SecTC2 Type-2 contraction of a λ-transition restricted to output-determinate STGs
SecTC2' Type-2 contraction of a λ-transition which is not in a weak syntactic conflict with an output transition

Finally, we note that also the determinisation of an *output-determinate* STG N can be seen as an LOD-transformation. If N is output-determinate, then constructing $DA(N)$ gives a language equivalent STG, which is not only output-determinate, but even deterministic. The same is true if one additionally minimises the deterministic automaton. In general, any transformation preserving the language and output-determinacy can be made into an LOD-transformation if its domain is restricted to output-determinate systems.

4. Decomposition into output-determinate components

Here, we apply the theory of output-determinacy 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 them, which is consistent with the ones developed in the previous section. Then we describe our decomposition algorithm and state its correctness.

4.1. Correct decompositions

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

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

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

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

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

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

Observe that Definition 4.1 is a generalisation of Definition 3.1: if $(C_i)_{i\in I}$ consists of only one component C_1 then $C = C_1$, no computation interference can occur, and (C4) can be dropped. Furthermore, in [19, 20], 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 implementation are known to be output-determinate.

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

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

The result below shows that this notion is equivalent to Definition 4.1, if the implementation is deterministic and the specification is output-determinate. Proposition 3.7 is obtained as a special case of this theorem by considering $I = \{1\}$ and $C = C_1$.

Theorem 4.3 *Let N be an output-determinate STG and $C \stackrel{\text{df}}{=} \|_{i\in I} C_i$ be a parallel composition of deterministic*

STGs such that $In_C \subseteq In_N$ and $Out_C = Out_N$. Then $(C_i)_{i\in I}$ is a correct distributed implementation of N iff it is a trace-correct distributed implementation of N.

4.2. The decomposition algorithm

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

• Choose a *feasible partition* $(In_i, Out_i)_{i\in I}$ for some set I with $Out_i \subseteq Out_N$ and $In_i \subseteq In_N \cup Out_N$ for each $i \in I$ (as explained in greater detail below). For each $i \in I$, a component C_i will be constructed, which produces the signals in Out_i by taking into account only the signals in $In_i \cup Out_i$.

• Construct an *initial decomposition* $(C_i)_{i\in I}$ as follows. For each $i \in I$, the *initial component* $C_i = (P, T, W, l_i, M_N, In_i, Out_i)$ is a copy of N except for the labelling and the signal classification. If $l(t) \in (In_N \cup Out_N) \setminus (In_i \cup Out_i)$, then the label is changed to $l_i(t) = \lambda$; such t and its original signal are *hidden*. In contrast, transitions which already have label λ in N are called *specification dummies*.

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

• Repeatedly apply LOD-transformations or **backtracking**, i.e., 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 a signal s and an output signal x of N are in structural conflict, then $x \in Out_i$ implies $s \in In_i$ (provided that $s \in In$) or $s \in Out_i$ (provided that $s \in Out_N$), for each $i \in I$.

The rationale for this is: clearly, a component responsible for an 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 $t, t' \in T$ 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.

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

The algorithm of this paper is a generalisation of the decomposition algorithm in [19]; the latter only dealt with deterministic specifications and for these, it considered the same partitions, transformations, and backtracking. Since the concept of output-determinacy was not available, it was required to remove all λ-transitions; thus, backtracking had also to be performed for a hidden signal if a respective transition could not be contracted, e.g., because it was on a loop or had an arc with weight greater 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 removed successfully. This can make the reachability graph much larger, while from the perspective of circuit design the additional signal might not be needed. We can avoid this in the present paper, which is an important contribution.

If a transition contraction generates a new dynamic auto-conflict, this is an indication that the original signal of the contracted transition might be important for producing the proper outputs, cf. [19, 20]; 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 [19] is guaranteed to find a correct decomposition without any final check.

In another version discussed in [19], 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 [17] is that the hope is most often in vain.

Consequently, we recommend a *conservative strategy*: if 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; such a 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 specification-dummy.

If all components are constructed successfully, circuits are synthesised from them using e.g., 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 are a correct implementation of the specification as we state now.

Theorem 4.4 *Consider the application of the decomposition algorithm to an STG N.*

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

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

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

Compared to the approach of [19], the proof of the above theorem 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 [19]. Additionally, the proof in [19] takes care to show that, for deterministic specifications, type-2 contractions can be applied without restriction. Since we use the same operations as in [19], we can read off from the correctness proof there that the same result applies here if in the specification N there are no weak triggers of or λ-transitions in structural conflict with output transitions; this observation means that we do not have to check for weak syntactic conflicts and this can save a little time.

5. Results

As described in the previous sections, it is now possible to leave λ-transitions in the final components as long as these are output-determinate. This is in particular useful for speeding up the successful decomposition strategy *tree decomposition* [17]. This strategy generates all components together; for efficiency, only some signals are contracted at each stage of the algorithm, resulting in re-usable intermediate STGs. If not all transitions of some signal can be contracted, one backtracks and the contraction of this signal and all of its transitions is postponed.

In practice, most of the postponed signals can actually be contracted at later stages of the algorithm. The new approach makes it possible to avoid postponing of signals, leaving the non-contractible transitions as dummies in the intermediate STGs. This gives better intermediate STGs and performance, since in most cases the dummies will be contracted later; otherwise they will remain in the result.

We applied this approach with the conservative strategy to a number of benchmark examples constructed from two basic BALSA handshake components [10]: the 2-way *sequencer* and the 2-way *paralleliser*; either can be described by a simple STG. The benchmark examples SEQPARTREE-N correspond to complete binary trees with alternating levels of sequencers and parallelisers (N is the number of levels); they are generated by the parallel composition of the elementary STGs corresponding to the individual sequencers and parallelisers. We also worked with other benchmarks made of handshake components (e.g., trees of parallelisers only), but the results did not differ much.

We applied four variants of tree decomposition to these benchmarks, as well as stand-alone PETRIFY and MPSAT. (The tool for CSC conflict resolution and decomposition described in [4,5] was not available from the authors.) The experiments were conducted on a PC with Pentium-IV 3GHz processor and 2GB RAM. The synthesis with stand-alone PETRIFY or MPSAT did not terminate within 12 hours, even for SEQPARTREE-05.

We performed two variants of tree-decomposition: with postponing signals as in [17], and with leaving the non-contractible transitions as dummies in the intermediate STGs as described above. Additionally, we performed some experiments using only *safeness-preserving contractions*, i.e., contractions which preserve the safeness of the STG. (This kind of contraction is needed to combine decomposition with unfolding techniques for STG synthesis, see [14].) Essentially, the preservation of safeness is another condition which can prevent some contractions and thus increase the runtime. This resulted in four series of experiments, see Table 1 for the results. In the end, the final components were synthesised with PETRIFY (including resolution of CSC conflicts), which was possible for every component. As explained in Section 3, this means that they are output-determinate, i.e., all decompositions are provably correct.

			Safe		Non-Safe	
N	$\|P\| - \|T\|$	$\|In\| - \|Out\|$	New	Old	New	Old
05	382 – 252	33 – 93	6	6	6	7
06	798 – 508	65 – 189	20	20	19	21
07	1566 – 1020	129 – 381	30	31	30	31
08	3230 – 2044	257 – 765	1:19	1:34	1:21	1:23
09	6302 – 4092	513 – 1533	2:48	2:57	2:54	2:59
10	12958 – 8188	1025 – 3069	10:35	47:09	11:21	11:36

Table 1. Results for SEQPARTREE-N — Columns 2 and 3 show the size and the number of signals, Columns 4 – 7 give the decomposition time plus the PETRIFY synthesis time for the components. Times are given in seconds or as min : sec. *Safe* means safeness-preserving contractions; the *old* method does not leave λ-transitions in the intermediate STGs, the *new* one does.

We consider the short runtimes for the examples as

a notable achievement of the proposed approach — e.g., SEQPARTREE-10 with more than 4000 signals was synthesised in about 10–11 minutes. One can see that leaving non-contractible transitions as dummies in the intermediate STGs is useful, especially for the case of safeness-preserving contractions. The reason for the latter effect is that in our benchmarks most λ-transitions are contractible but many cannot be contracted while preserving safeness.

6. Conclusion

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

One of the main applications 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 [19, 20]: it can be applied to non-deterministic specifications, it no longer requires that all the λ-transitions must be contracted in the final components, and it can use more net reductions; moreover, the list of such reductions can easily be extended by adding new LOD-transformations. The experimental results show that our decomposition algorithm can handle very large STGs efficiently. Combined with tools for logic synthesis [14], it can be used in the context of control re-synthesis, as explained in the introduction.

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

References

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

[2] K. v. Berkel, M. Josephs, and S. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87:223–233, 1999.

[3] G. Berthelot. Transformations and decompositions of nets. In *Petri Nets: Central Models and Their Properties*, LNCS 254, pages 359–376. Springer, 1987.

[4] J. Carmona and J. Cortadella. ILP models for the synthesis of asynchronous control circuits. In *Proc. ICCAD'03*, pages 818–825, 2003.

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

[6] J. Carmona, J. Cortadella, V. Khomenko, and A. Yakovlev. Synthesis of asynchronous hardware from petri nets. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, LNCS 3098, pages 345–401. Springer, 2004.

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

[8] 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. Inf. and Systems*, E80-D, 3:315–325, 1997.

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

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

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

[12] M. Josephs. An analysis of determinacy using a trace-theoretic model of asynchronous circuits. In *Proc. ASYNC'03*, pages 121–131. IEEE Comp. Soc. Press, 2003.

[13] M. Josephs and D. Furey. A Programming Approach to the Design of Asynchronous Logic Blocks. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design, Advances in Petri-Nets*, LNCS 2549, pages 34–60. Springer, 2002.

[14] V. Khomenko and M. Schaefer. Combining decomposition and unfolding for STG synthesis. In *Proc. ATPN'07*, to appear.

[15] V. Khomenko, M. Schaefer, and W. Vogler. Output-determinacy for asynchronous circuit synthesis. Technical Report 2007-02, Univ. of Augsburg, 2007. URL: `www.informatik.uni-augsburg.de/skripts/techreports`.

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

[17] M. Schaefer, W. Vogler, R. Wollowski, and V. Khomenko. Strategies for optimised STG decomposition. In *Proc. ACSD'06*, 2006.

[18] T. Verhoeff. Analyzing specifications for delay-insensitive circuits. In *Proc. ASYNC'98*, pages 172–183. IEEE Comp. Soc. Press, 1998.

[19] W. Vogler and B. Kangsah. Improved decomposition of signal transition graphs. In *Proc. ACSD'05*, pages 244–253, 2005.

[20] W. Vogler and R. Wollowski. Decomposition in asynchronous circuit design. In *Concurrency and Hardware Design*, LNCS 2549, pages 152 – 190. Springer, 2002.

[21] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with OR causality. *FMSD*, 9:189–233, 1996.