Advanced STG Decomposition

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat)
der Fakultät für Angewandte Informatik
der Universität Augsburg

Vorgelegt von

Mark Schäfer

Gutachter

Erstgutachter: Prof. Dr. Walter Vogler
Zweitgutachter: Prof. Dr. Bernhard Möller

Mündliche Prüfung

28. Februar 2008

To my wife Maren and my daughter Charlotte

I would like to thank everybody who supported and entertained me during this four years and all the years before.

Thank you all!

Brina, as ever — Elmar, for explaining objects — Frank, for pizza
Gabriele, for all that libraries — Heinz, for bed and breakfast — Hilli, for smoking
Jan & Thomas, for määäääääh — Jonathan & Victor, for many discussions
Jürgen, for da brainz — Minna, for math — Peter, for the games
Stephan, for the ketchup — Ute, for the fun — Walter, for being sensei
Werner, for showing the way — Willi, for welding

Advanced STG Decomposition

Advanced STG Decomposition

Mark Schaefer
University of Augsburg

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

mark@markschaefer.de

Herstellung und Verlag
Books on Demand GmbH, Norderstedt

ISBN: 978-3-8370-4604-5

Contents

1 Introduction **11**
1.1 Scope . . . 12
1.2 Related Work . . . 12
1.3 Contribution and Organisation . . . 13

2 Asynchronous Circuits **15**
2.1 Digital Circuits . . . 15
2.1.1 Sequential Circuits . . . 17
2.1.2 Synchronous Circuits . . . 18
2.2 Asynchronous Circuits . . . 20
2.2.1 Timing Assumptions . . . 24
2.2.2 Operating Modes . . . 26
2.2.3 Handshake Circuits . . . 27

3 Basic Definitions **31**
3.1 Petri Nets . . . 31
3.1.1 Labelled Petri Nets . . . 35
3.1.2 Implicit and Redundant Places . . . 38
3.1.3 Unfoldings . . . 40
3.2 Signal Transition Graphs . . . 40
3.2.1 State Graph and Complete State Coding . . . 43
3.2.2 Parallel Composition, Renaming and Hiding . . . 47
3.3 Decomposition . . . 52
3.3.1 Purpose . . . 52
3.3.2 Algorithm . . . 55
3.4 Properties of Transition Contractions . . . 65
3.5 Some Considerations about Consistency . . . 70

4 Determinate Decomposition **75**
4.1 Redundant Places in Marked Graphs . . . 76
4.2 Determinacy of Petri Net Operations . . . 81
4.2.1 Decomposition as Reduction System . . . 81
4.2.2 Reduction is Locally Confluent . . . 83

5 Internal Signals **95**
5.1 Extended Correctness Definition . . . 96
5.2 Hierarchical Decomposition . . . 100
5.3 CSC Solving . . . 108
5.4 Comparison with other Approaches . . . 114
5.5 Other Implementation Relations . . . 116
5.5.1 Conformance . . . 117
5.5.2 I/O-Compatibility . . . 119
5.5.3 Strictness of Inclusions . . . 121

6 Advanced Decomposition Strategies **123**
6.1 Correctness of New Strategies . . . 124
6.2 Reordering Transition Contractions . . . 125
6.3 Lazy Backtracking . . . 125
6.4 Tree Decomposition . . . 130
6.4.1 Component Aggregation . . . 134
6.5 Undo Stack and Self-Triggering . . . 135
6.6 CSC-Aware Decomposition . . . 135
6.6.1 CSC-Aware Decomposition . . . 137
6.6.2 Safeness-Preserving Contractions . . . 141
6.6.3 Implicit Places and Dynamic Auto-Conflicts . . . 146
6.7 Conclusion . . . 147
6.7.1 Results . . . 147
6.7.2 Application to other Decomposition Approaches . . . 157
6.7.3 Conclusion . . . 158

7 Output-Determinacy **161**
7.1 Definitions . . . 163
7.1.1 Output-Determinacy . . . 165
7.2 Decomposition into Output-Determinate Comp. . . . 169
7.2.1 Valid STG transformations . . . 173
7.2.2 New Algorithm . . . 181
7.3 Output-Determinacy and Internal Signals . . . 185
7.4 Results . . . 188

8 The Tool DesiJ **191**
8.1 Implementation 193
8.1.1 STG Undo Implementation 193
8.1.2 Calculation of Decomposition Trees 196
8.1.3 Conditions, Collectors and Operations 199
8.1.4 Verification of the Implementation 200
8.2 Command Line Options and Parameters 202

9 Conclusion and Future Research **209**

Appendix **213**
Bibliography 213
List of Figures 218
List of Tables 220
List of Examples 221
Index 222

Chapter 1

Introduction

Asynchronous and synchronous circuits are subclasses of digital circuits. While the latter class is well understood and forms the base of almost all modern microcomputers, this is not the case for asynchronous circuits, and they are only used for special purposes so far. This is despite their many advantages like reduced power consumption and peak current, less emission and an average case delay rather than a worst case delay, as it is typical for synchronous circuits.

However, actually building asynchronous (or clockless) circuits is much more complicated and computationally harder than for synchronous circuits. This problem emerged – or at least became more apparent – during the last 60 years. In fact, at the beginning of the digital area in the 1950's [CKK+97]:

> *... computers were either asynchronous, or had significant asynchronous components. This was due to the fact that computer design was still mostly an art ...*

Together with the growing automation of circuit design, synchronous circuits became more and more established due to their comparatively easy analysis. At present, the synchronous design flow embraces all stages of development, from the first informal description to the final chip layout.

Nevertheless, the research on asynchronous logic was continued during the last 60 years. Especially during the last ten years there was a tendency to include asynchronous parts in synchronous designs – today there even exist some completely asynchronous microprocessors. Now, there is some strong evidence and strong believe throughout the asynchronous community that clockless circuit design starts turning from art to engineering as it happened to synchronous design before.

One of the main problems of asynchronous circuits is that *every* event in some part of a circuit potentially influences other parts of the circuit. Therefore, one has to absolutely avoid spurious and non-digital events. In order to deal with such effects, one has to analyse the timing of a circuit, which is quite complicated; to simplify this, timing assumptions are made which lead to several subclasses, formal models and synthesis algorithms of asynchronous circuits. All that is not important for synchronous circuits, where not events but rather states of the circuit at certain times (determined by a clock signal) are important for the behaviour of the circuit; hence, spurious events are allowed which simplifies the analysis.

Another problem arises from the fact that events can happen and are propagated all the time – in synchronous circuits events of a clock cycle are 'collected' and propagated together. Hence, asynchronous circuits usually exhibit a lot of concurrency, i.e independent events can happen in arbitrary orders. This leads to a large state space and makes it impossible to efficiently synthesise large circuits, i.e. to derive Boolean equations from the formal description. The aim of this thesis is to improve a method with which also large asynchronous specifications can be synthesised and smaller ones can be synthesised faster.

1.1 Scope

This thesis deals with *Signal Transition Graphs (STGs)*, which are a Petri net based formalism for the description of asynchronous circuits. In particular, it deals with the STG decomposition algorithm of Vogler, Wollowski and Khangsah [VW02, VK06], which tackles the state explosion encountered during circuit synthesis. This is done by generating several smaller STGs from a large STG which together exhibit the same behaviour as the initial one. STGs were invented by Wendt [Wen74, Wen77] and became better known due to the work of Chu [Chu87a, Chu87b], who also described a first STG decomposition algorithm. This first algorithm was quite restrictive and was significantly improved by Vogler and Wollowski.

Here, this algorithm is investigated in greater detail, its range of application is extended and its efficiency is improved. Additionally, the tool DesiJ, which implements the decomposition algorithm is presented.

1.2 Related Work

STGs are widely used and there are several approaches to their synthesis (i.e. the derivation of a physical implementation), as well as a lot of research groups using them by some means or other.

In the following, we present five approaches, research groups resp. which are closely related to this thesis:

- The PhD thesis *Trace Theory for Automatic Hierarchical Verification of Speed-Independent circuits* [Dil88] by D. Dill introduces a language based formalism for the modelling of asynchronous circuits called *trace structures*. Some of the concepts described there are still important within the asynchronous community. In Section 5.5 we compare our implementation relation with the one defined there.

- The book *Logic Synthesis of Asynchronous Controllers and Interfaces* [CKK+97] summarises the theoretical background of STGs and their synthesis. It is one of the sources for Section 3.2. It is also the base for the tool PETRIFY by J. Carmona, one of the book's authors. Although PETRIFY is quite old today (the last version is from 2003), it is still a reference for new algorithms.

- J. Carmona and J. Cortadella are working on the decomposition of STGs with the help of integer linear programming (ILP) [Car03, CC03]. Their algorithm is similar to the one described here, but uses a different approach to determine the necessary signals for the single components. In Section 5.4, we explain it in more detail, give a correctness proof for it and compare it to our approach. Lately, the group has started working on a new direct synthesis algorithm (i.e. without using decomposition), which is also based on ILP.

- C. Myers and T. Yoneda also work on a decomposition algorithm [YOM04]. They use a method different from ours and from the approach of Carmona and Cortadella to determine the sets of necessary signals. They also work on *timed STGs*, i.e. STGs with time annotations for the events, and the synthesis thereof, which tries to harness the additional timing information [YMKM05, YM06].

- V. Khomenko has developed a complete design flow for STG synthesis based on STG *unfoldings* [KK01, Kho03, KKY06]. The corresponding tools are PUNF for the efficient generation of unfoldings and MPSAT for synthesis and other related tasks. In Section 6.6 we describe how our decomposition approach can be combined with this unfolding approach.

1.3 Contribution and Organisation

The thesis is organised as follows: Chapter 2 is about asynchronous circuits in general and their relation to digital and synchronous circuits; mainly, it is a tour d'horizon containing well-known facts and some examples. There is no specific source for this

chapter, except for the asynchronous part which is mainly based on the book *Logic Synthesis of Asynchronous Controllers and Interfaces* mentioned above. Chapter 3 explains the basics of Petri nets, STGs and STG decomposition. The latter part is a summary of the related papers of Vogler, Wollowski and Khangsah [VW02, VK06]. The former parts are well-known and have been enhanced with some additional examples.

The main contribution of the thesis can be found in the succeeding chapters. They are based on articles in which the present author participated. Section 3.4 also contains new contributions, which are loosely connected and presented together there.

Chapter 4 (based on [SVJ05]) is about the decomposition of so-called *marked graphs*, which are a practically important subclass of STGs. It is shown that for marked graphs the result of the non-deterministic decomposition algorithm is always uniquely determined. Chapter 5 (based on [SV07]) applies the original decomposition approach to STGs with *internal signals*. A suitable generalised correctness notion is given and justified; furthermore, the concept of *hierarchical decomposition* is explained and proven correct. Additionally, correctness notions of other authors are investigated and compared.

Chapter 6 (based on [SVWK06, KS07]) introduces several decomposition strategies and heuristics, i.e. methods to execute the non-deterministic decomposition algorithm in a more deterministic way in order to improve its efficiency. In its second part, this chapter deals with the problem that decomposition can easily introduce *Complete State Coding conflicts* into the components. A combination of our tool DesiJ with the tool Mpsat is introduced and the theoretical foundations thereof are given. In particular, it is important here to *preserve safeness* of the STGs. Sufficient structural conditions, as well as an efficient dynamic method to guarantee this, are given.

Chapter 7 (based on [KSV07]) presents *output-determinacy*, which is a relaxation of determinacy. This new concept is first justified and then applied to improve the decomposition algorithm as well as its correctness proof. Furthermore, the improved algorithm is applied to large benchmark STGs (more than 4000 signals). Chapter 8 gives an overview of the decomposition tool DesiJ developed by the present author. Some special algorithms are discussed and an overview of its functionality is given.

Finally, Chapter 9 gives a conclusion of the thesis and an outlook to future research.

Mainly in the first half of the thesis, the reader will find boxed examples. Usually, they are longer and more detailed than a figure and a simple explanation. Although they are not absolutely necessary to understand the main text, it is strongly recommended to read them.

Chapter 2

Asynchronous Circuits

In this chapter, *asynchronous* (or *clockless*) circuits are introduced; their main property is the absence of a special clock signal which enforces a synchronisation of all operations of a circuit, as it is the case for *synchronous* circuits.

This chapter is organised as follows: the first three sections give a tour d'horizon on digital circuits, their classification and the properties of asynchronous circuits. Although this chapter is essentially self-contained, the reader should have seen a diagram of a digital circuit before, and he should know basic Boolean functions like *AND* or *XOR*.

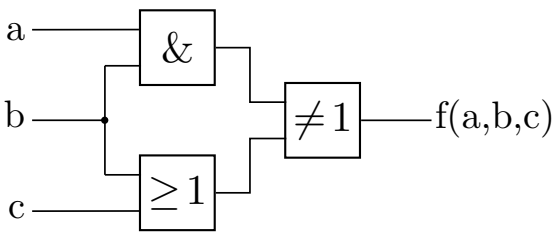
2.1 Digital Circuits

The main property of digital circuits is the binary, Boolean resp. interpretation of electric voltage, i.e. voltages in some range are considered as 0, *FALSE* or *logical low* and voltages in a disjoint range are considered as 1, *TRUE* or *logical high*. In most cases these voltage intervals are separated by a 'forbidden' interval, which should only be entered shortly when switching the values. The behaviour of a circuit is undefined when its inputs are impressed with such a voltage. For the rest of the thesis, the term 'digital' will be omitted.

Typically, circuits are built from basic elements, called *(logical) gates*. These gates work under the interpretation described above and perform Boolean functions like *AND*, *NOT* or other functions of arbitrary complexity. The gates themselves are built from transistors, capacitors etc.; here, we consider them as atomic and do not discuss their 'inner life'. The gates are connected by *wires*. These wires have a natural

direction: there is one source which determines the current voltage of the wire, but there may be several sinks, which listen to the current value of the wire. In this context, sources are the outputs of a gate or the inputs provided by the environment of a circuit, and sinks are inputs to a gates or outputs to the environment.

A wire carries the (Boolean) value of a *signal*; a change in this signal is called a *signal edge* or just *edge*. There are *raising edges* (from 0 to 1) and *falling edges* (from 1 to 0); they are denoted by $^+$, $^-$ resp. We also talk about e.g. input signal edges or input edges for short. If the context is clear, we will just use the term input to denote an input edge. We also frequently identify a wire with the signal it carries.



a	b	c	f(a,b,c)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Figure 2.1: Simple circuit and corresponding truth table

For an example, have a look at Figure 2.1 where a simple circuit with three gates is depicted. There is an *AND* (&), an *OR* (≥ 1) and an *XOR* gate ($\neq 1$). They are connected such that the signals a and b are inputs of the *AND* gate, b and c are inputs of the *OR* gate and the outputs of these gates are inputs of the *XOR* gate. The output of the latter is also the single output of the circuit, which calculates the Boolean function $f(a, b, c)$. The values of f are given in the *truth table* on the right.

Asynchronous circuits form – together with synchronous ones – the group of *sequential circuits* which is itself part of the *digital circuits*, see Figure 2.2.

One can observe that the output value of the circuit from Figure 2.1 does only depend on the current[1] values of the inputs. Hence, such a *combinatorial circuit* cannot store an internal state. The characteristic of a combinatorial circuit is the absence of *feedback (loops)*[2], where feedback means that the output of a gate A is the input of

[1] Of course, if an input value changes, it takes some time till the output value changes.

[2] In other words, if we consider the circuit as a graph with the gates as nodes and the wires as directed edges (with multiple targets however), the resulting graph is acyclic. This is not necessarily obvious from a diagram as in Figure 2.1, because more complex building blocks may contain internal feedback loops.

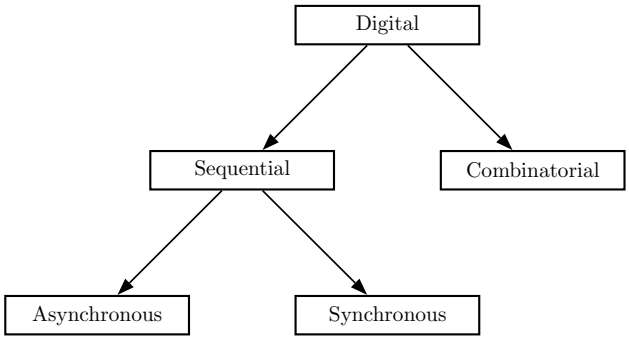


Figure 2.2: Digital circuit classification. Note that there are asynchronous specifications which have a combinatorial implementation; hence, the respective classes overlap for such cases.

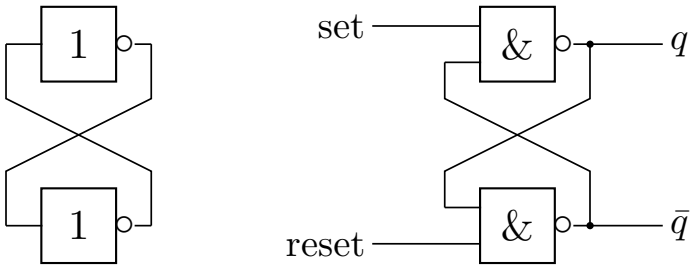
a gate B whose output affects the output of A again. Such feedback loops constitute the 'state-memory' of a circuit.

Combinatorial circuits are well understood and easy to analyse and synthesise, and it is also possible that an STGs can be implemented by a combinatorial circuit. Nevertheless, this is not the intended purpose of an STG, and therefore, we will not consider combinatorial circuits further and concentrate on the two types of sequential circuits.

2.1.1 Sequential Circuits

A circuit with a feedback loop is called *sequential*. One of its simplest examples is a *flip-flop*, whose only purpose is to store a single bit, see Figure 2.3.

In the flip-flop core, there is one feedback-loop, starting at the output of the upper NOT gate, going to the lower gate and from its output back to the upper gate. As one can easily see, there are two stable states of this circuit. The output of the upper gate is 1 causing the output of the lower gate to be 0 which causes the output of the upper gate to be 1 and so forth. In the other stable state, the output values are reversed. After initialisation, the circuit is in an instable state: both outputs are initially 0, causing both gates switching their output to 1, causing both outputs switching to 0 and so on. In practice, the gates are not perfectly equal and eventually one gate will win this so-called race. However, this can take an arbitrarily long time and the resulting stable state is undefined.



set	reset	action
0	0	not allowed
0	1	set
1	0	reset
1	1	keep value

Figure 2.3: Flip-flop core — *NAND*-Flip-flop — operations of the latter

To control the state of a flip-flop core, the *NOT* gates are replaced by *NAND* gates. As long as the two additional inputs *set* and *reset* have value 1, the behaviour of the core is not affected and it is in a stable state. If, for instance, the *set* input is lowered, the upper gate's output is forced to 1, and hence the lower gate's output is forced to 0 – the flip flop stores the value 1. If instead *reset* is lowered, eventually the other stable state is reached – the flip flop stores the value 0. Since the outputs are expected to have opposite values all the time, the input (0,0) is forbidden, since it would result in the output values (1,1).

2.1.2 Synchronous Circuits

Synchronous circuits make heavy use of flip-flops to store their current state explicitly. The basic structure of a synchronous circuit is depicted in Figure 2.4.[3]

The state of the circuit is stored in a number of flip-flops.[4] These flip-flops are clocked, i.e. they change their state only when a certain edge of the clock signal occurs. The

[3]This circuit is a *Medwedew automaton*, i.e. the current state is also the output of the circuit. A *Moore automaton* has an additional (combinatorial) *output function* which calculates the output from the current state. A *Mealy automaton* has an output function which also considers the current input.

[4]There are various types of flip-flops, differing in their state-controlling signals, e.g. the D(ata)-flip-flop which has only one input whose value is stored directly.

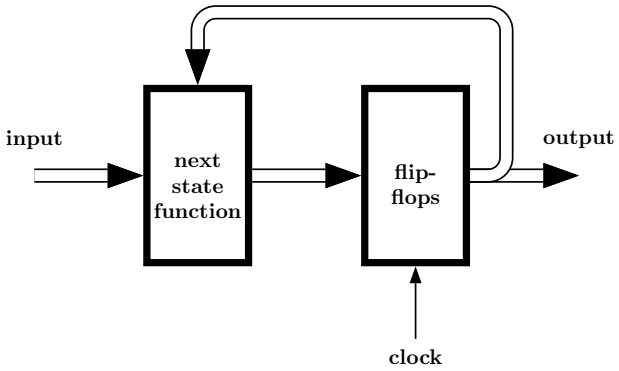


Figure 2.4: Basic structure of synchronous circuit. Double lines denote multiple wires.

clock signal itself is assumed to change its value in a periodic manner; a clock cycle is the interval between two equal edges (including only one).

Besides the feedback-loop in the flip-flops, there is also a feedback loop in the circuit itself: the states of the flip-flops are also inputs of the *next-state-function*, which usually is a combinatorial function. If the active edge of the clock signal occurs, the calculated next state is stored in the flip-flops and eventually propagated to the outputs of the flip-flops and therefore to the next-state-function. The input values might change now and a new next state is calculated and so forth.

There are two important problems:

- The value of the next-state-function has to be calculated within a single clock cycle to guarantee a proper behaviour. However, the delay of the next-state-function circuit depends on the input values. Therefore, the clock-cycle has to be long enough to cover the *worst-case-delay* of the function. This decreases the efficiency of the overall circuit.

- The next-state-function can only be stable if the inputs are stable. If the inputs change their value late within a clock cycle, the next-state-circuit also needs more time to become stable. To guarantee correctness and increase efficiency, the inputs should therefore only change at the beginning of a clock cycle.

For large synchronous circuits there are also problems on the physical level, mainly resulting from the generation and distribution of the clock signal throughout the circuit:

Energy In modern microprocessors about 1/3 of the chip area and the chip energy consumption is used to distribute the clock signal. Since the energy consumption of such processors can be up to 130W, this is a severe problem, e.g. for cooling or from an ecological point of view.

The energy is also consumed unevenly, i.e. with peaks after the active clock edges, when a new clock cycle begins.

EMI (electromagnetic influence) The clock signal is some form of alternating voltage, which induces electromagnetic radiation. This is worsened by the special form of the clock signal, which is not described by some sine function, but rather has sharp (digital) edges. This results in a whole spectrum of radiation, rather than an emission with the clock frequency only (as it would be the case for a sine function).

2.2 Asynchronous Circuits

Asynchronous circuits neither have a clock signal nor do they store their current state explicitly in flip-flops. The current state is stored implicitly in the value of all signals. A state change is not triggered by a clock signal, but is caused by the change of input signals. The circuit might observe this change without a visible reaction or change the values of its output signals. This is observed by the environment, which might as reaction change another inputs and so forth.[5] Observe that a state change has to be propagated through the circuit, but it is possible to start the next state change while a previous one is still in progress, e.g. if two inputs have changed their value roughly at the same time. However, this feature can be restricted by the operating mode of the circuit as described in section 2.2.2.

An example of an asynchronous circuit is a *C-element* [MB59], see Figure 2.5. Like a flip-flop, a C-element is a storage element for one bit: in the initial state all signals are low; if both inputs a and b have the same value the output c is set to this value, otherwise c keeps its old value. Therefore, a C-element is an implementation of the function $[c] = (a \wedge b) \vee (c \wedge (a \vee b))$, where $[c]$ is the next value of c.

In practice, C-elements are used to build asynchronous circuits with a structure like the synchronous circuit from Figure 2.4, i.e. there are some C-elements to store the state explicitly and a combinatorial next-state function which calculates proper input values for these elements. This approach makes logic decomposition and technology mapping, i.e. the mapping of functions to actually existing gates, easier. Still there are more problems than in the synchronous case, see [CKK+97] for a detailed discussion.

[5] An *autonomous circuit* has only output signals, which change their values in a predefined manner.

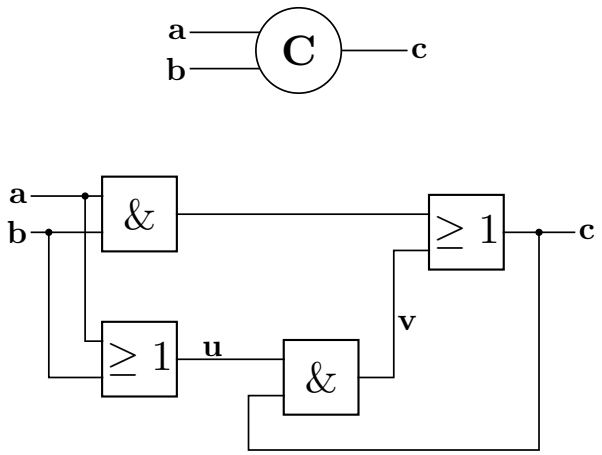


Figure 2.5: C-Element and non-atomic implementation

In practice, a C-element is implemented as a *complex gate*,[6] i.e. as an atomic gate whose inner structure and inner signals do not have to be considered. During technology mapping, it is not always possible to implement complicated functions with complex gates, but one has to combine available gates. The problem with the latter is that new internal signals between the gates have to be introduced, which have to be considered during analysis, which can easily lead to problems as described below.

Here, we will consider the non-atomic implementation of a C-element shown in Figure 2.5 to demonstrate a so-called *hazard*. The term hazard is used in a broad sense, denoting any kind of malfunction caused by the non-instantaneous propagation of signal values within a circuit. A much more detailed classification and discussion of hazards can be found in [Cav07].

For the following discussion cf. the timing diagrams in Figure 2.6, in which the values of all relevant signals are shown in the course of time ($b = 0$). At the beginning, the circuit is in the stable state $(a, b, u, v, c) = (1, 0, 1, 1, 1)$. For simplicity, we assume that each gate delays the propagation of signal edges for a certain time which is equal for all four gates.

To set $c = 0$ correctly (left), the environment has to set $a = 0$ and then has to wait a certain time. This triggers u going down (1), which triggers v going down (2), which finally triggers c going down (3). If a is raised later, this triggers u going up again

[6] They are called 'complex' because they implement a complex Boolean function.

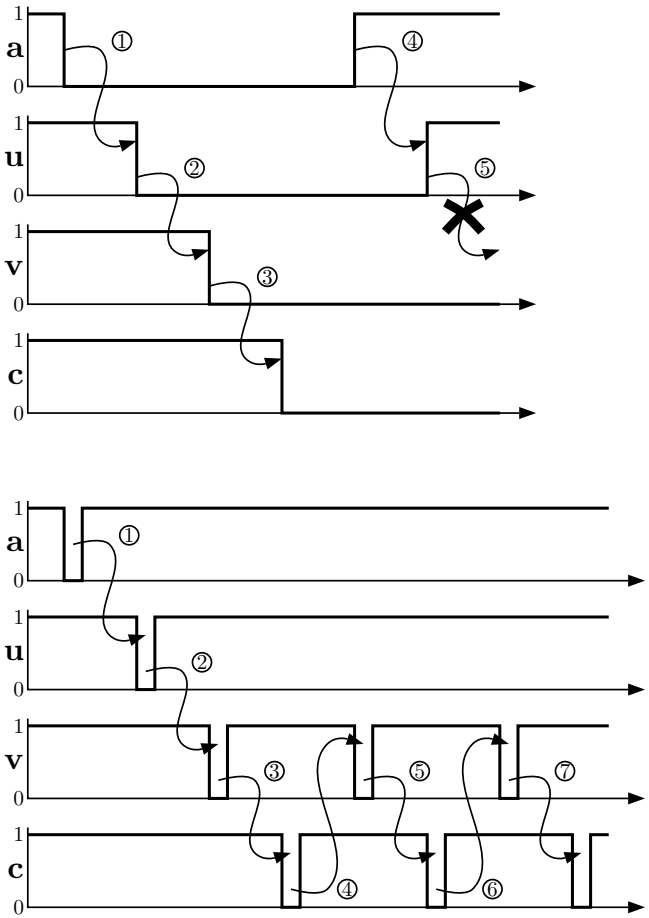


Figure 2.6: Setting $c = 0$ for a C-Element (with $b = 0$). *top*: correct behaviour. *bottom*: hazard caused by a too short low-phase of input a.

(4) which triggers nothing (5).

If the down-phase of a is too short (right), a hazard occurs: the initial down-phase of a triggers a down-phase of u (1), which triggers a down-phase of v (2), which eventually triggers a down-phase of c (3). Since the down-phase of v is shorter than the delay of of the last OR-gate, v is high again when c is set to 0. Therefore, a new down-phase of v is triggered (4), which triggers a new down-phase of c (5) and so on (6,7). In real life, the C-element will eventually reach a stable but unpredictable state (observe that for $a = 1$ and $b = 0$, $v = c = 0$ and $v = c = 1$ are both possible). This situation is called a *race* and it can also result in *meta-stability* of the signals, i.e. v and c are impressed with voltages in the forbidden interval.

Another example of an hazard is a *glitch*: an input edge which should not produce an output edge, causes *two* of them. The circuit shown in Figure 2.7 can produce a glitch in the following situation all inputs have value 1 when the value of b changes to 0 – obviously, this should not change the value of x. However, the value of the upper AND gate changes to 0, before the value of the lower AND gate changes to 1, since the latter signal edge is delayed by the NOT gate. This results in a short period in which both inputs of the OR gate are low and also x goes down shortly. In practice, this period can be so short that the OR gate does not produce proper signal edges, but a more or less distinctive voltage breakdown which cannot be interpreted digitally because it enters the forbidden voltage interval without crossing it.

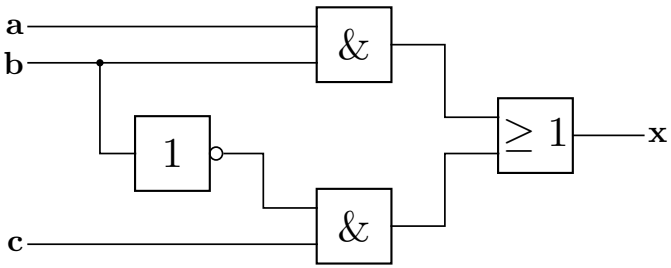


Figure 2.7: Circuit with possible glitch. The gate labelled with 1 is a NOT gate.

In synchronous circuits, such errors will cause no problems as long as the circuit is stable when the next clock edge occurs. To cope with these problems in asynchronous design, one works with timing assumptions of the gates and wires as described in the next subsection.

At the end of this subsection, we introduce the VME bus controller, which will be used a an example throughout this thesis.

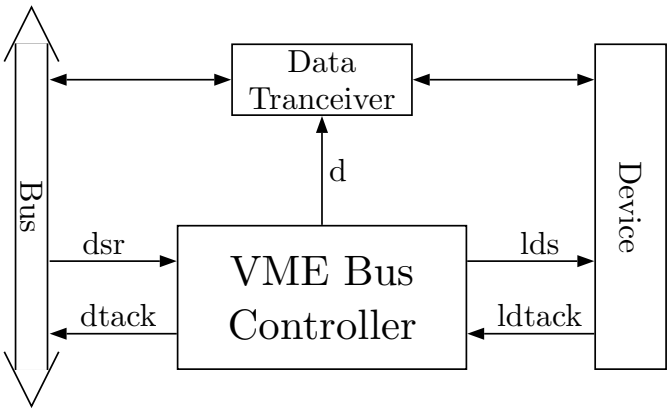


Figure 2.8: VME Bus Controller

The *VME* bus controller (VERSAmodule Eurocard bus) in Figure 2.8 is based on the VERSAbus specification, which was developed in 1979 for Motorola 68000 based systems, (see `www.vita.com` for more details). Nowadays, the VME bus and its derivatives are widely used, in particular in the aerospace industry. For instance, the computer system of the international space station (ISS) uses the VME bus.

The VME bus controller connects a device to a VME bus; to do this, also a data transceiver is needed, which connects the data path of the bus with the data channel of the device and controls the data flow direction.

The read cycle of the controller (in which data is read from the device by the bus) works as follows: a request to read is made by the signal edge dsr^+. This request is propagated to the device through lds^+. When the device has the data ready ($ldtack^+$), the controller must open the transceiver to transfer data to the bus (d^+) and inform the bus that the data can be read from the transceiver ($dtack^+$). If the bus has read the data, a dsr^- edge is sent and the transaction is complete. Afterwards, all interface signals must return to zero with maximum concurrency.

2.2.1 Timing Assumptions, Speed-Independence and Output-Persistency

To analyse the potential for hazards one has to make *timing assumptions* about the propagation of signal edges and the delays of gates.

Here, the *unbounded gate delay* is used: the signal propagation is considered as instan-

taneous, i.e. it is assumed that there is no delay on wires. On the other hand, the gates are allowed to have an arbitrarily large (or short) and varying delay. Asynchronous circuits have to be synthesised such that they work correct under these assumptions.[7]

There are some justifications for this model: the wire delay is only important if a specific gate output is the source of more than one input. If an output is connected to several inputs, the unbounded gate delay model is still correct if there is an *isochronic fork*: the wire length from the output to every input is the same, such that an output change arrives at every input with the same delay. In either case, the delay of the wire can be added to the delay of the gate.

Otherwise, there will be a *skew* in the propagation of a signal. However, this problem can be ignored for small circuits, cf. [CKK+97].

The unbounded gate delay model is the base for *speed-independent (SI)* circuits: a circuit is speed-independent if its functional behaviour does not depend on the delay of its gates.

Speed-independence can be characterised as follows: a circuit is speed-independent if it is *output-persistent* in all possible states for a given environment. Output-persistency is very important for the robust behaviour of an asynchronous circuit; informally, it means that once an output edge is activated, it has to occur and must not be deactivated by the occurrence of other signals.

Formally, a violation of output-persistency is raised by a *conflict*, i.e. two signal edges are activated but only one of them can occur. Since we distinguish input and output signals, there are three cases for a conflict:

Output-output conflict Such a conflict can easily result in a race with all its consequences, as it was described for the flip-flop and C-element above. However, there are situations where a circuit has to make a random decision between two outputs[8]; for this purpose *arbiters* or *ME-elements* [YKK+96] are used which make this decision safely.[9] Typically, the specification of an asynchronous circuit will not contain an output-output conflict. The decision is 'outsourced' to an external arbiter which becomes therefore part of the environment, and the output-output conflict is transformed into an input-input conflict (see below).

Input-output conflict This conflict leads to undefined behaviour: in practice, the delay of the environment is greater than the delay of the circuit itself. Therefore,

[7] In other models the delay is associated with the wires. Independent of this, the delay can be bounded or unbounded.

[8] For example, a bus controller has to choose between two requesting bus devices.

[9] An arbiter is realised non-digitally, and can also show an arbitrarily long period of meta-stability before the decision is made.

the output will be definitely produced, but the environment will not wait for this output and produce the conflicting input signal anyway. Then, either the circuit is in a stable state, but does not expect the input edge anymore or the circuit is not stable and the unexpected edge leads to a hazard.

Input-input conflict This conflict is no problem for an asynchronous circuit; it will wait until one of the inputs will arrive, and it is the responsibility of the environment to produce exactly one of the possible input edges.

2.2.2 Operating Modes

During the history of asynchronous circuits, several design methodologies and formalisms were developed. They correspond to different *operating modes*, i.e. assumptions about the interaction of the circuit with its environment. The main question is: when are inputs and outputs allowed to change their value?

The *fundamental mode* [Huf64, Ung69] makes the following assumptions:

- Only one input changes at a time
- Afterwards, no input will change until the circuit is in a stable state

Since it is unknown how many output edges (if any) will be produced in response to an input edge, the environment has to wait a certain time to be sure that the circuit has reached a stable state. Although there is no clock signal, the situation is therefore similar to synchronous circuits.

In *Huffman mode* [Huf64], it is possible that more than one input edge occurs before outputs are produced. Depending on the specific properties of the underlying physical implementation, there are two delay values δ_1 and δ_2 ($\delta_1 < \delta_2$): if two input edges are separated by less than δ_1 they belong to the same phase and are considered as simultaneous. When all input edges of a phase have occurred, the circuit can stabilise and produce output edges. If two inputs are separated by more than δ_2 they belong to different phases. Time values in the interval $[\delta_1, \delta_2]$ are illegal and lead to undefined circuit behaviour.

The design flow in fundamental or Huffman mode starts with *flow tables*. In these tables, possible states of the circuit – resulting from all the signal values – are listed together with possible input edges. From them, the equations of all output signals are derived. This process is quite complex in order to avoid failures of the operating circuit. For a detailed description of the design process see [Cav07].

In *burst mode* every state of a circuit enables different *input bursts*, i.e. a set of input edges which can occur in any order. A signal must not occur more than once in

each burst, and for every state no burst can be a subset of another burst. The latter condition ensures that the end of a burst can be detected without using a timeout. It is possible that the circuit starts changing its state and produces outputs when an input burst is still in progress as long as the environment is not disturbed with the running burst, e.g. if the environment is slow enough to ignore these outputs for the moment.

Finally, in *input-output mode*, input and output edges can be mixed arbitrarily in time; the only restriction is the protocol of the circuit-environment behaviour itself. Prominent examples of corresponding design formalisms are STGs and handshake circuits. The latter are described below and will be considered again as a source of benchmark examples for decomposition, and in Chapter 9 as a future research topic.

2.2.3 Handshake Circuits

Handshake circuits [Ber93] (*HS circuits* for short) are a special design methodology for asynchronous circuits. An HS circuit is built from a collection of *handshake components* which are connected via *handshake channels*. Such a channel consists of two wires, signals resp.: the *request (req)* and the *acknowledge (ack)* signal. The former is produced by the active component of the channel, the latter by the passive component.

For most applications the *4-phase protocol* is used for communication on a channel: the *active ports* of a channel raises the *req* signal to initiate the handshake. This is received by the *passive ports* of the channel, which sends a raising *ack* edge back to the active port, which in turn lowers the *req* signal which eventually lowers *ack*.

This handshake sequence may be interleaved with handshakes of other channels of the respective components or data-transfers between them. A component might have more than one active port, and when and on which channel a handshake is performed depends on the specific component, but the handshake itself always works as described above.

There are two types of channels: synchronisation-only and data channels. The first type only transfers control from one component to another, while the second additionally has data signals to transfer information from one component to another.

We will clarify this explanation with the example in Figure 2.9. The HS components are drawn as large circles, the channels as lines between them, a filled dot attached to a component denotes an active port, an empty dot a passive port. If a channel is drawn as an arrow, it denotes a data channel, and the arrow direction denotes the direction of the data. Observe, that the dataflow direction is independent of the active/passive port.

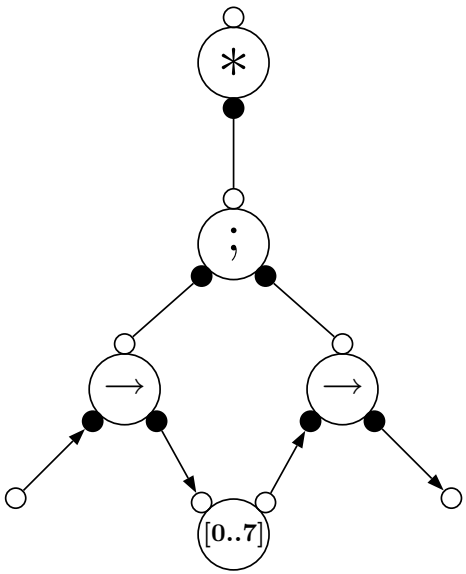


Figure 2.9: 8 bit Buffer out of handshake components

```
procedure buffer (input i : byte; output o : byte) is
  variable x : byte
begin
  loop
    i -> x ; o <- x
  end
end
```

Figure 2.10: Textual description of the 8 Bit Buffer

Here, there are four kinds of HS components:

Loop component ($*$) After an initial handshake on its passive port, the component only performs complete handshakes on its single active port. The initial handshake however, is never completed, i.e. control will never return from the loop component. This component is used to trigger infinite behaviour of some other components.

Sequencer component (;) The handshake on the passive port is split up into two sequential handshakes on the active ports. The first of these handshakes is completed before the second starts, and only if the latter is finished the handshake on the passive port is also finished, i.e. the active handshakes are embedded into the passive one.

Fetch component ($\rightarrow$) This component performs a data transfer from one active port to another when triggered on its passive port.

Observe that the components connected to the active ports may delay the handshakes till the data is available from left component, or the data was received by the right component.

Buffer component ([0..7]) Just stores some data (8 bit here) for later use. There are two passive ports but only one handshake may be in progress at a time.

The example in Figure 2.9 models an 8-bit buffer: the loop component ensures that one cycle (reading, storing and passing of an 8-bit value) is repeated infinitely often. The sequencer component first activates a fetch component which reads a value from the environment or another component[10] and stores it in the buffer. Only if this has happened, the value can be transfered by the second fetch component to the environment or another component; then the cycle starts again. The sequencer guarantees the buffer functionality, i.e. only if a value is in the buffer it can be read, and only if it was read a new value can be stored.

The similarity of the handshake components to programming language constructs is no coincidence. In fact, the HS component graph is derived nearly one-to-one and very efficiently from a textual description; this is called *syntax-directed* translation. For instance, the buffer is described as in Figure 2.10.

The main advantage of HS circuits are their relaxed timing assumptions. The handshake communication is actually *delay-insensitive*, i.e. every delay in the wire of a signal edge is tolerated by the circuit. The components itself are usually very small and

[10] If necessary, the active port can be out-shorted, i.e. the *req* output is directly connected to the *ack* input. This guarantees the 4-phase protocol, but not that the data is actually valid.

can be realised, as speed-independent (or even delay-insensitive) circuits very easily (cf. also the discussion about the usefulness of speed-independence in Section 2.2.1).

The main disadvantage is the overall complexity of the resulting circuits. Usually, they are heavily *over-encoded*, i.e. there are far more signals than necessary to store the internal state of the circuit. Furthermore, the handshake communication is rather time consuming.

Despite of this, handshake circuits are the only asynchronous circuits which are actually used for industrial designs, because other models do not allow as large specifications as they can be synthesised from HS circuits.

Handshake circuits were first developed in [Ber93], where the corresponding programming language is called TANGRAM. A recent successful approach is BALSA [EB02] (see also `intranet.cs.man.ac.uk/apt`). BALSA was used to develop several asynchronous versions of the ARM RISC microprocessor design.

The company 'Handshake Solutions' (`www.handshakesolutions.com`) has also developed the TANGRAM approach further and has developed asynchronous versions of ARM microprocessors as well as asynchronous embedded controllers.

The above program is written in BALSA, TANGRAM resp.; these languages look alike for the constructs used. We will encounter handshake circuits again in Section 6.6, in Chapter 7 and in the future research part of the conclusion.

Chapter 3

Basic Definitions

Petri nets were invented 1962 by Carl Adam Petri in his PhD thesis [Pet62]. They allow the representation of a distributed system, i.e. a system in which the global state is comprised of several local states which can change independently. Besides for the modelling of asynchronous circuits, they are used e.g. for workflow management, software engineering and verification.

In this chapter their basic notions as well as signal transitions graphs (STGs) and other related topics are introduced. We assume basic mathematic and computer science knowledge, e.g. sets, functions, graphs, languages and automata theory. A detailed introduction into Petri nets can be found e.g. in [Pet81, Rei85, Mur89]. A first introduction to STGs can be found in [CKK+02]. [Wol97] presents a much more detailed analysis of STGs and their properties.

3.1 Petri Nets

A *Petri net* is a 4-tuple $N = (P, T, W, M_N)$ with

- P is the finite set of *places* and T is the 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*
- $M_N : P \to \mathbb{N}_0$ is the *initial marking*

W defines weighted and directed connections between places and transitions (and vice versa); by the definition of W, places cannot be connected with other places and transitions cannot be connected with other transitions. As a consequence, a Petri net

can be considered as a weighted and directed bipartite graph. We say that there is an *arc* between x and y if $W(x, y) > 0$.

A place can be considered as a 'container' which stores an arbitrary number of *tokens*. The state of a Petri net is completely described by the number of tokens in each place, and a *marking* is a function $M : P \to \mathbb{N}_0$ which describes this state. We say that a place p is marked under M if $M(p) > 0$. The initial marking M_N defines the initial state of a Petri net. Whenever a Petri net N, N', N_1 etc. is introduced, the corresponding elements (P, T, W, M_N), $(P', T', W', M_{N'})$, (P_1, T_1, W_1, M_{N_1}) etc. are defined implicitly.

In the graphical representation of a Petri net, places are drawn as circles enclosing their marking; for each token, a small filled circle is drawn. Transitions are drawn as boxes and arcs as arrows labelled with their weight (if it is greater 1). See Figure 3.1 for an example.

Transitions can be considered as events which can occur if the current marking (i.e. the current state of the net) fulfils certain properties. A transition t is *enabled* under a marking M written as $M[t\rangle$ if

$$\forall p \in P.\, M(p) \geq W(p, t).$$

This gives a meaning to the weight of an arc from a place p to a transition t: it is the number of tokens which have to be in p to enable t. Keep in mind that a transition has to be enabled by all places to become enabled at all.

Usually, for a given transition t there are a lot of places p with $W(p, t) = 0$; these places never prevent the enabling of t. Therefore, only the places for which actually an arc to t exists have to be considered for the enabling of t. These places are called the *preset* of t, which is denoted with ${}^\bullet t$. They are defined by

$${}^\bullet t = \{p \in P \mid W(p, t) > 0\}.$$

Analogously, the *postset* $t^\bullet$ of t is defined by

$$t^\bullet = \{p \in P \mid W(t, p) > 0\}.$$

The pre- and postset of a place is defined analogously. The enabledness condition can now be written as

$$\forall p \in {}^\bullet t.\, M(p) \geq W(p, t).$$

An enabled transition can *fire* (or *occur*) resulting in a new marking M'. This is denoted as $M[t\rangle M'$. The firing of t removes some tokens from ${}^\bullet t$ according to the respective arc weight, i.e. just the tokens which enabled t. After this, new tokens

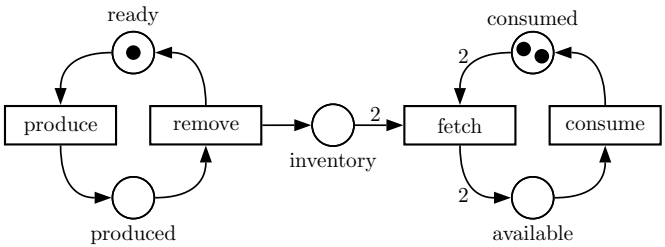


Figure 3.1: Producer-consumer Petri net.

are put into the places of $t^\bullet$, again according to the respective arc weight. Formally, $M[t\rangle M'$ if:

$$M[t\rangle \ \wedge \ \forall p \in P.\, M'(p) = M(p) - W(p,t) + W(t,p)$$

The new marking M' might enable some other transitions, the firing thereof leads to other markings and so forth. Therefore, we generalise the *firing rule* to transition sequences: a transition sequence $v = t_1 t_2 \ldots t_n$ $(n \geq 0)$ is enabled under a marking M, denoted as $M[v\rangle$ if there are markings $M_1, M_2, \ldots, M_n$ such that

$$M[t_1\rangle M_1 \ \wedge \ M_1[t_2\rangle M_2 \ \wedge \ldots \wedge \ M_{n-2}[t_{n-1}\rangle M_{n-1} \ \wedge \ M_{n-1}[t_n\rangle$$

or for short

$$M[t_1\rangle M_1[t_2\rangle M_2[t_3\rangle \ldots [t_{n-1}\rangle M_{n-1}[t_n\rangle.$$

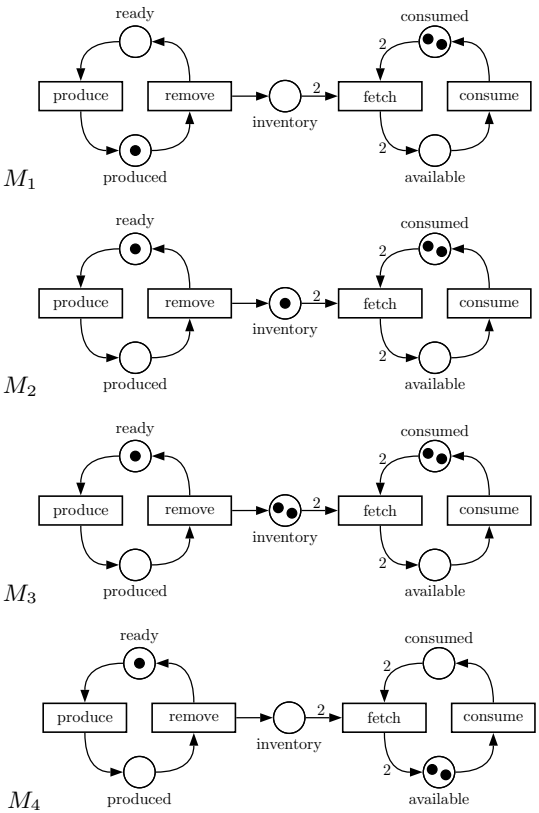
Analogously, we write $M[v\rangle M'$ if $M[v\rangle$ and the firing of v results in the marking M', i.e. $M_{n-1}[t_n\rangle M'$. A transition sequence is called *firing sequence* if it is enabled under the initial marking. The set of all firing sequences of a Petri net N is denoted as $FS(N)$.

A marking M' is called *reachable from a marking* M if there is a transition sequence v such that $M[v\rangle M'$. The set of all markings which are reachable from M is denoted as $[M\rangle$. A marking is just called *reachable* if it is reachable from the initial marking, and $[M_N\rangle$ is the set of all *reachable markings*.

A place p is called *k-bounded* for $k \geq 0$ if for every reachable marking M, $M(p) \leq k$, i.e. the marking of p cannot exceed k, and it is just called bounded if such a k exists. A place is called *safe* if it is 1-bounded. A Petri net is called (k-)bounded if every place is (k-)bounded, and it is called safe if every place is safe. One can easily see that a (finite) Petri net is bounded if and only if the set of reachable markings is finite.

A marking is called *home state* if it is reachable from every reachable marking. A Petri net is called *reversible* if the initial marking is a home state. Obviously, in a reversible Petri net every reachable marking is a home state.

As mentioned at the beginning of this chapter, Petri nets are widely used for design and modelling of distributed systems with independent parts. The Petri net in Figure 3.1 is a variation of the well-known *producer-consumer* example.
It models the following system: in a factory, a machine A **produces** items. If an item was **produced** it has to be **removed** and put into an **inventory** in order to make the machine **ready** to produce the next item. The items are **consumed** by a machine B and are **fetched** – always two at a time – to make them **available** to B. If all available items are **consumed**, the next two items are fetched.
In the initial state, only the transition *produce* is enabled; firing it leads to the marking M_1 below. Now, only *remove* can fire resulting in marking M_2. If this is repeated once more, a marking M_3 is reached where both, *produce* and *fetch*, can fire. This demonstrates that there are actually independent parts. If *fetch* fires, the inventory is emptied and the tokens can finally be consumed, see M_4.



Example 3.1: Producer-Consumer

A transition t is called *live* if $\forall M \in [M_N\rangle . \exists M' \in [M\rangle . M'[t\rangle$. Thus, a live transition can always fire again, no matter what happened before. A net N is live if every transition is live. A transition t is called *1-live* if there is a firing sequence $v \in T^*$ with $M_N[vt\rangle$, i.e. t can fire at least once. A net N is 1-live if every transition is 1-live.
Two different transitions t_1 and t_2 are in *structural conflict* if ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$. They are in *(dynamic) conflict under a marking M* if

$$M[t_1\rangle \wedge M[t_2\rangle \ \wedge \ \exists p \in {}^\bullet t_1 \cap {}^\bullet t_2 : \ M(p) < W(p,t_1) + W(p,t_2).$$

This means that both transitions are activated but only one can actually fire. We say t_1 and t_2 are in *conflict* if they are in conflict under some reachable marking.
Two transitions t_1 and t_2 are enabled *concurrently* under a marking M if

$$\forall p \in P : \ M(p) \geq W(p,t_1) + W(p,t_2).$$

This means that they are both activated under M and could fire in principle at the same time. In particular, $M[t_1 t_2\rangle$ and $M[t_2 t_1\rangle$.

Some firing sequences of the producer-consumer example from Figure 3.1 are:

- *produce remove produce remove fetch consume consume*
  This firing sequence reaches the initial marking again.
- *(produce remove)*^{k} for $k \geq 0$.
  This firing sequence puts k tokens on place *inventory*, which is therefore unbounded.

Since the place *inventory* is unbounded the Petri net itself is also unbounded, the other places are safe, 2-bounded resp.
Although there are infinitely many reachable markings, the net is reversible: starting from an arbitrary marking, fire *produce* and *remove* until there is an even number k of tokens on *inventory* and *ready* is marked. Then fire *consume* until *available* is empty and then fire *(fetch consume consume)*$^{\frac{k}{2}}$, which empties *inventory* and reaches the initial marking.

Example 3.2: Producer-Consumer continued

3.1.1 Labelled Petri Nets

A *labelled Petri net* is a 6-tuple $N = (P, T, W, M_N, \Sigma, l)$ where

- (P, T, W, M_N) is a Petri net
- Σ is an *alphabet* of labels
- $l : T \to \Sigma \cup \{\lambda\}$ is a *labelling function* where $\lambda \notin \Sigma$ denotes the *empty word*.

The labelling function l is extended to transition sequences as usual: $l(t_1 t_2 \dots t_n) = l(t_1) l(t_2) \dots l(t_n)$, deleting λ automatically. The notion of enabledness is lifted to labels: we write $M[w\rangle\rangle$ if $M[u\rangle$ and $w = l(u)$ for some $u \in T^*$. A sequence $w \in \Sigma^*$ is called a *trace of a marking* M if $M[w\rangle\rangle$ and it is called a *trace (of* N*)* if $M_N[w\rangle\rangle$.[1] The *language* $L(N)$ of N is the set of all traces.

The notions of conflict and concurrency are extended to labels as follows: e.g. two labels $v, w \in \Sigma$ are in (dynamic) conflict if there are two transitions t_1 and t_2 with $l(t_1) = v$, $l(t_2) = w$ and t_1 and t_2 are in conflict. If $v = w$, this is called an *auto-conflict*. *Structural auto-conflicts* and *auto-concurrency* are defined analogously. A Petri net is *deterministic* if it has no λ-labelled transitions and no auto-conflict or auto-concurrency.

The notions defined above for unlabelled Petri also apply for labelled Petri nets. For the rest of the thesis, we only consider labelled nets and frequently omit the term 'labelled'.

Often, nets are considered to have the same behaviour if they are language equivalent. *Bisimulation* (see [Mil89]) is a finer behaviour equivalence; it is based on *simulation*.

Definition 3.1 (Simulation)
A *simulation from* N *to* N' is a relation $\mathcal{S}$ between the markings of N and N' such that $(M_N, M_{N'}) \in \mathcal{S}$ 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 $(M_1', M_2') \in \mathcal{S}$. △

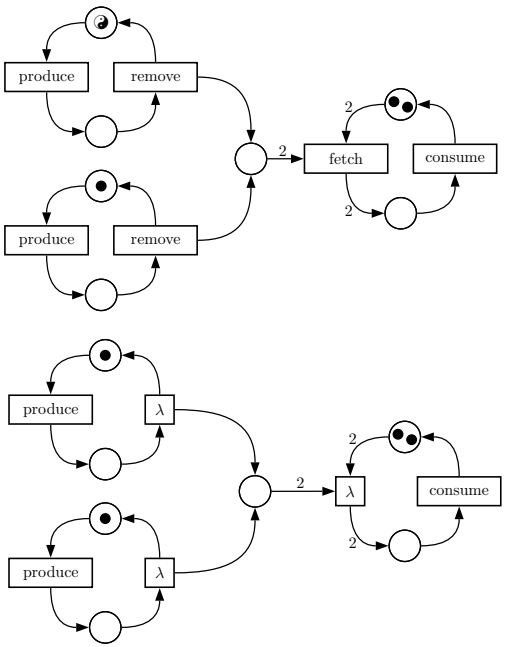
If such a simulation exists, then N' can go on simulating all events of N forever, and therefore we have $L(N) \subseteq L(N')$.

A relation $\mathcal{B}$ is a *bisimulation* between N and N' if it is a simulation from N to N' and $\mathcal{B}^{-1}$ is a simulation from N' to N. If such a bisimulation exists, we call the nets *bisimilar*; intuitively, the Petri nets can work side by side such that in each stage, each net can simulate the events of the other one, and we have $L(N) = L(N')$.

Observe that if $\mathcal{S}_1$ is a simulation between N and N' and $\mathcal{S}_2$ one in the other direction, N and N' have the same language, but it does not imply that they are also bisimilar. However, for deterministic nets, language equivalence and bisimulation coincide.

[1] Note that a (finite) trace may correspond to a firing sequence which contains arbitrarily many λ-labelled transitions.

In the first producer consumer example, every possible action corresponds to exactly one transition, which is not sufficient for more elaborate scenarios. For instance, two producers which can produce items and put them in the inventory concurrently, have to be modelled by two transitions $produce_1$ and $produce_2$ (and $remove_1$ and $remove_2$). However, this method is kind of inconvenient and, more important, does not allow to abstract the behaviour, e.g. to consider the $produce_1$ and $produce_2$ as a single *produce* event as in the labelled Petri net on the left. Here, only an abstract *produce* event occurs independent of the actual producer.



The labelling of transitions with λ allows to *suppress* the corresponding actions. (Normally the term 'hiding' would be used here, but later we define hiding in a special way.) For instance, in the net on the right side, we abstracted from the details of the internal storage and transportation system. A possible trace is:
produce produce consume produce consume produce consume consume,
and in general, every trace, where the k-th *consume* event is preceded by at least $k + k \,\%\, 2$ *produce* events.

Example 3.3: Producer-Consumer continued

The *reachability graph* $RG(N)$ of a net N is the edge-labelled directed graph (V, E) with:

- $V = [M_N\rangle$
- $E = \{(M, u, M') \in V \times (\Sigma \cup \{\lambda\}) \times V \mid M[t\rangle M' \wedge u = l(t)\}$

Therefore, the nodes represent all reachable markings and the edges all possible transitions between them; observe that each edge represents the firing of a single transition ($M[t\rangle M'$) rather than the firing of a label ($M[u\rangle\rangle M'$). The reachability graph is finite exactly if $[M_N\rangle$ is finite, i.e. N is bounded. A Petri net is deterministic exactly if its reachability graph can be considered as a deterministic automaton.

The reachability graph is an example for an *interleaving semantics* of a Petri net, i.e. even if two events could happen at the same time (like the two *produce*-labelled transitions in Example 3.3), this is not apparent in the reachability graph, because only single events are shown. Instead, only all possible interleavings of the concurrent events can be found.

3.1.2 Implicit and Redundant Places

Definition 3.2 (Implicit and Redundant Places)
Let N be a Petri net. The place $p \in P$ is called *implicit* if it can be removed from N without changing the set of firing sequences.

The place p is *(structurally) redundant* [Ber87] if there is a set of places Q – called *reference set* – with $p \notin Q$, a valuation $V : Q \cup \{p\} \to \mathbb{N}$ and some $d \in \mathbb{N}_0$ which satisfy the following properties for all transitions t:

(1) $V(p)M_N(p) - \sum_{q \in Q} V(q)M_N(q) = d$

(2) $V(p)(W(t,p) - W(p,t)) - \sum_{q \in Q} V(q)(W(t,q) - W(q,t)) \geq 0$

(3) $V(p)W(p,t) - \sum_{q \in Q} V(q)W(q,t) \leq d$

We call V *balanced* if for all transitions $t \in T$ (2) is an equality, i.e.

$$V(p)(W(t,p) - W(p,t)) - \sum_{q \in Q} V(q)(W(t,q) - W(q,t)) = 0 \qquad \triangle$$

Removing implicit places is an important operation of the STG decomposition algorithm (see below). However, implicitness is hard to decide, and therefore we actually consider only redundant places, since detecting them does not require to generate the reachability graph.

Remark: It is well-known that the reachability problem (RP) for Petri nets is ExpSpace-hard [Esp98]. This even holds for SPZ-RP (Single-Place-Zero RP) where we ask if a given place p can be emptied; we can also assume arc-weights to be 1. Given an instance of SPZ-RP, we can add a fresh t and the arcs (p,t), (t,p), and observe that p is implicit if and only if no marking with zero tokens in p is reachable. This shows ExpSpace-hardness of the implicitness problem.

Checking redundancy however, can be transformed easily into a linear programming problem [STC98] which is infeasible in the case of redundancy. Since redundancy is defined over $\mathbb{N}$, one might expect that this results in an *integer* linear programming problem, which is known to be NP-complete (e.g. [Sch86]). However, observe that in this case, a solution over $\mathbb{Q}$ suffices, since the solution can be multiplied with the common denominator. Hence, redundancy can be solved with well-known linear programming algorithms in polynomial time (e.g. [Ren95]). However, since this approach is still not efficient enough, DESIJ looks only for a subset of redundant places, called *shortcut places*, cf. Definition 4.3 and the discussion on page 80.

To understand that each redundant place is implicit, observe that the first two items of the redundancy definition ensure that p is something like a linear combination of the places in Q with factors $V(q)/V(p)$. Indeed, for the case $d = 0$, the first item says that p is such a combination initially; the second item, in the case of equality, says that this relationship is preserved when firing any transition.

The proof that a redundant place p is indeed implicit argues that initially the valuated token number of p is at least d greater than the valuated token sum on Q by the first item, and that this difference can only get greater when firing transitions by the second item; the third item says that each transition needs at most d 'valuated tokens' more from p than from the places in Q. This shows that for the enabling of a transition the presence or absence of p does not matter.

Since deletion of p preserves the firing sequences it also preserves liveness and 1-liveness. In general, implicitness does not imply redundancy, but in Chapter 4 we will show that these notions coincide for a subclass of Petri nets, called marked graphs.[2]

Throughout this paper, if a place p (p', p_1, ...) is considered to be redundant, a corresponding reference set Q (Q', Q_1, ...) and valuation function V (V', V_1, ...) are implicitly given. If some valuation function V is given explicitly, the reference set is implicitly determined by $Q = \{s \in P \mid V(s) > 0\} \setminus \{p\}$.

[2] [CCJS94] shows that the second redundancy item characterises that p is *structurally implicit*, i.e. each marking of the other places can be extended to p such that p is implicit.

3.1.3 Unfoldings

A *finite and complete unfolding prefix* of a Petri net N is a finite acyclic net which implicitly represents all reachable markings of N together with the transitions enabled under those markings. Intuitively, it can be obtained through *unfolding* N, i.e. by successively firing transitions, under the following assumptions: (1) for each new firing a fresh transition (called an *event*) is generated; (2) for each newly produced token a fresh place (called a *condition*) is generated. This process is started at the places which are marked under the initial marking, see also Example 3.4.

An unfolding is infinite whenever N has an infinite firing sequence. Actually, Petri nets have to be finite, but since we only use finite unfolding prefixes (see below), we make an exception here. However, if N has finitely many reachable states, the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix.

Efficient algorithms exist for building such prefixes [Kho03], and they ensure that the number of non-cut-off events in a complete prefix can never exceed the number of reachable states of N [ERV02].

In most cases (in particular for highly concurrent Petri nets), complete prefixes are exponentially smaller than the corresponding reachability graphs, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in reachability graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the reachability graph will be a 100-dimensional hypercube with 2^{100} markings, whereas the complete prefix will coincide with the net itself.

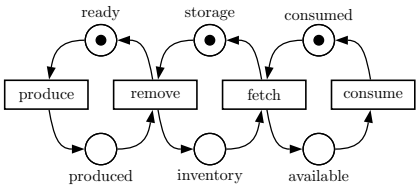
Since many practical STGs (see below) usually exhibit a lot of concurrency, but have rather few choice points, their unfolding prefixes are often exponentially smaller than the corresponding reachability graphs. In fact, in many of the experiments conducted in [Kho03, KKY04] they were just slightly bigger then the original STGs themselves. Therefore, unfolding prefixes are well-suited for both, visualisation of an STG's behaviour and alleviating the state space explosion problem.

In Section 6.6 we will explain how decomposition can be combined with the unfolding based synthesis of [Kho03, KKY04, KKY06].

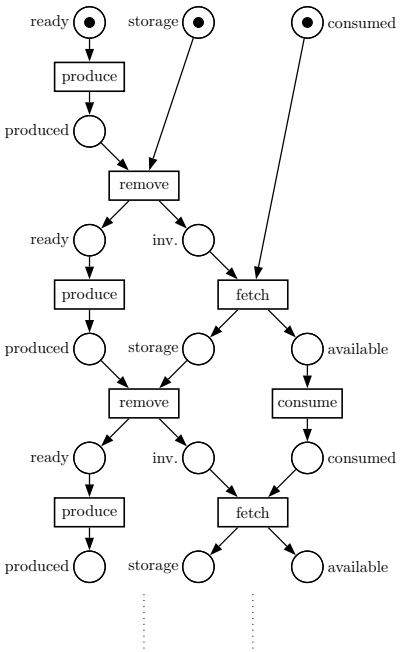
3.2 Signal Transition Graphs

In this section we introduce *Signal Transition Graphs* or *STG*s for short. They are a formalism for the modelling of asynchronous circuits, and describe both, the proper behaviour of a circuit as well as the corresponding proper environment. We distinguish

Below one can see the 'classic' version of the consumer-producer Petri net; the capacity of the inventory is limited and only one token at a time is fetched by the consumer.



This is a prefix of the unfolding of the above net. Observe, that each transition now represents an occurrence of the original transition and can fire only once. Although an unfolding is infinite in principle, this prefix contains every reachable marking of the net, i.e. every reachable marking is also reachable here when considering the place labels.



Example 3.4: Unfolding Example

three kind of signals: In, Out and Int are the disjoint sets of *input*, *output* and *internal* signals, where an internal signal is also an output of the circuit, but one which is unknown to the environment. The term '*signature*' refers to this partition of the signals of an STG. Additionally, we define the set of all signals $Sig = In \cup Out \cup Int$, the set of *external signals* $Ext = In \cup Out$ and the set of *local signals* $Loc = Out \cup Int$. Usually, a, b, c are input signals, x, y, z are output signals, u, v, w are internal signals and s is an arbitrary signal.

A *signal edge* is the combination of a signal and the direction of the edge: $+$ for a rising edge, and $-$ for a falling one; $Sig^{\pm} = Sig \times \{+, -\}$ is the set of all signal edges (analogously for the other signal subsets). Instead of $(a, +)$ we write a^+ and we write $a^{\pm}$ if the direction of the edge is unknown or does not matter – if $a^{\pm}$ occurs more than once in the same context, the same direction is denoted.

Hence, an STG N is an 8-tuple $(P, T, W, M_N, In, Out, Int, l)$ where $(P, T, W, M_N, Sig^{\pm}, l)$ is a labelled Petri net. If an STG has no internal signals, the $Int = \emptyset$ entry will be omitted. Since for synthesis the reachability graph of an STG has to be generated, we consider only bounded STGs.

An *input transition* is labelled with an input edge and analogously for the other signal types. A transition which is labelled with λ does not correspond to a signal change of the modelled circuit; it is called *dummy transition* or just *dummy*. In specification STGs, dummy transitions are used as a design simplification; in the decomposition algorithm they are introduced in intermediate stages and play a different role (see Section 3.3). A formal semantics for them is introduced in Section 7.

The graphical representation is naturally quite similar to ordinary Petri nets, except for two differences:

- The colour of a transition reflects the type of the corresponding signal. Input transitions are red, output transitions are blue, internal transitions are green and dummy transitions are white.
- Unmarked places with only one transition in the preset as well as in the postset, the respective arcs having weight 1, can be omitted; the respective transitions are connected directly.

An STG describes the behaviour of a circuit and the proper environment as follows: each reachable marking corresponds to a state of the circuit. If an input edge is activated under a given marking, the environment is allowed to produce it and the circuit must be ready to receive it; if an output is activated, the circuit is required to produce it and the environment must be ready to receive it; if an internal signal is activated, the circuit is required to produce it.

In practice, if the circuit or the environment (which is usually a different circuit

as well) activates some signal edge, the speed-independent model allows that it is produced immediately (and usually this is the case). This is the reason for some pitfalls:

- Input-output and output-output conflicts: cf. Section 2.2.1
- Dummy-input conflicts: the environment produces the input edge while in the meantime, the circuit decides that this input is undesired
- Activation of input signals by internal signals, i.e. $M[u^{\pm}\rangle\rangle M'[a^{\pm}\rangle\rangle$ but $\neg M[a^{\pm}\rangle\rangle$: since the environment cannot see the internal signals, it might produce the input edge before the internal edge was produced.

3.2.1 State Graph and Complete State Coding

While the state of an STG is described by a marking, the state of the corresponding circuit is described by signal values (in contrast to synchronous circuits there are no explicit storage elements like flip-flops). This is captured formally in the following way: for an STG N, a *state vector* is a function $Sig \to \{0, 1\}$ where '0' denotes logical low and '1' logical high. A *state assignment* assigns a state vector sv_M to each reachable marking M of N. The *state assignment* must satisfy the following conditions for every signal $s \in Sig$ and every pair of reachable markings $M, M' \in [M_N\rangle$ with $M[t\rangle M'$:

$$
\begin{aligned}
&l(t) = s^{+} \text{ implies } sv_M(s) = 0, sv_{M'}(s) = 1\\
&l(t) = s^{-} \text{ implies } sv_M(s) = 1, sv_{M'}(s) = 0\\
&l(t) = s'^{\pm} \text{ for } s' \neq s \text{ implies } sv_M(s) = sv_{M'}(s)\\
&l(t) = \lambda \text{ implies } sv_M = sv_{M'}
\end{aligned}
$$

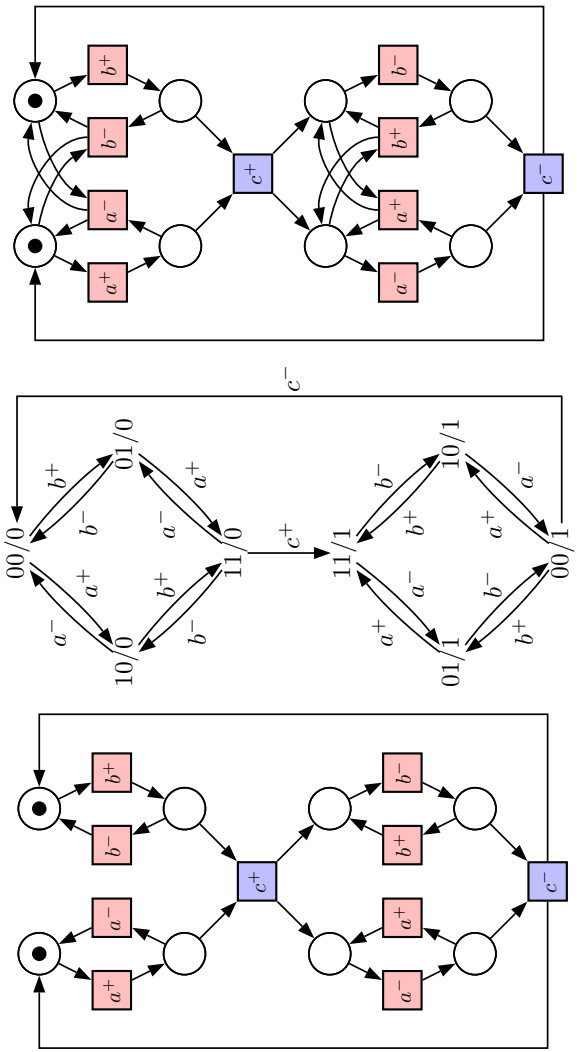
If such an assignment exists, it is uniquely defined by these properties,[3] and the reachability graph and the underlying STG are called *consistent*. For a consistent STG the *state graph* SG_N is defined as the corresponding reachability graph where each marking M is labelled with sv_M.

In contrast to this state based definition, the *language based consistency* notion demands that in every possible trace of an STG the edges of each signal have to alternate, i.e a^{+} must be followed by a^{-} must be followed by a^{+} and so forth, and that the first edge of every signal is the same in every trace.

Clearly, the state based notion implies the language based one, and except for pathological cases, the opposite is also true. It was proven in [VW02, VK06] that the

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

The STG on the left models the C-element from Figure 2.5, its state graph is shown in the middle. However, there are four input-output conflicts: between c^+ and a^-, b^- resp. and between c^- and a^+, b^+ resp. The STG on the right has no such conflicts since it forbids the respective input edges if c^+, c^- resp. is activated. Therefore the hazard described on page 21 is explicitly forbidden.



Example 3.5: C-Element: State Graph and Input-Output Conflict

language based notion is preserved by the decomposition operations discussed below. Again, for pathological cases, this is not the case for the state-based notion. In general however, these notions coincide, and if not explicitly mentioned, either of them applies. See Section 3.5 for a detailed discussion.

From an *inconsistent* STG, one cannot synthesise a circuit. Nevertheless, the decomposition algorithm and most of the results of this thesis do not require consistency; exceptions will be mentioned explicitly. Note that a consistent STG cannot have autoconcurrency, since this would imply that e.g. a^+ can fire twice under some reachable marking.

For $v \in (Sig_N^{\pm})^*$, $codeChange(S, v)$ is defined as the function over $S \subseteq Sig_N$ which maps each $s \in S$ to the difference between the numbers of s^+ and of s^- in v; for convenience, we define $codeChange(v) = codeChange(Sig, v)$, and for $v \in T^*$, $codeChange(v) = codeChange(l(v))$.

Since the only state information of a circuit are the current values of its signals, the next actions the circuit has to perform (i.e. the activated output edges) must be derivable therefrom without ambiguity. This is captured by the following notion.

Definition 3.3 (Complete State Coding)
A consistent STG has *Complete State Coding (CSC)* if:

$$\forall x \in Loc,\ M, M' \in [M_N\rangle : sv_M = sv_{M'} \Rightarrow (M[x^{\pm}\rangle\rangle \ \Leftrightarrow\ M'[x^{\pm}\rangle\rangle) \ ,$$

i.e. markings with equal state vectors have to enable the same outputs, internals resp. △

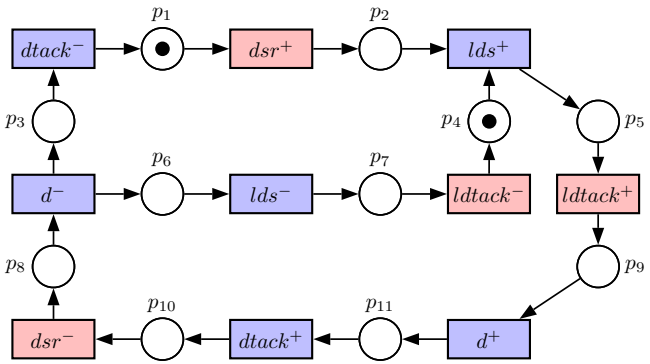
It is possible that different input signals are enabled in M and M' because these are not controlled by the circuit. A more restrictive property is *Unique State Coding (USC)* , which requires that no two markings have the same state vector.

From an STG without CSC one cannot synthesise a circuit. However, if an STG does not have CSC, this can be achieved either by timing assumptions or – more importantly – by the introduction of new internal signals.

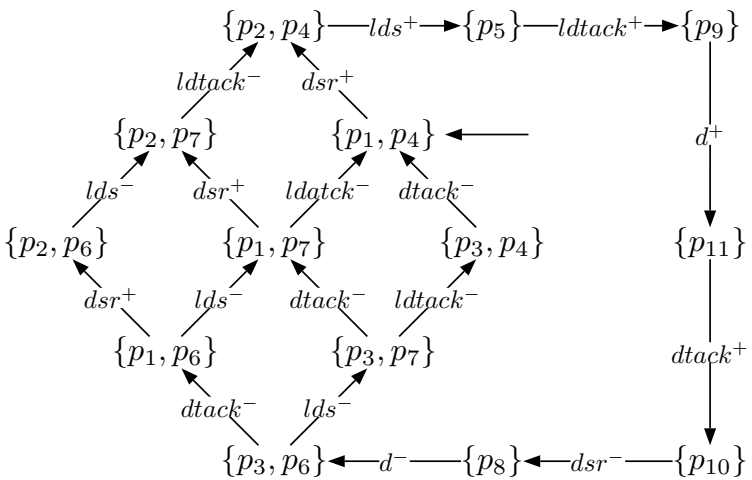
- The first method makes further assumptions on the relative order of signals, e.g. that an output is always faster than an input, even if they are concurrently enabled in the STG. Then, the state in which the input has occurred first will never be reached and can be ignored. With such assumptions, states corresponding to a CSC conflict can be deleted from the reachability graph. This method does not lead to speed-independent implementations and is not considered further.

- An interpretation of a CSC conflict is that the circuit has not enough mem-

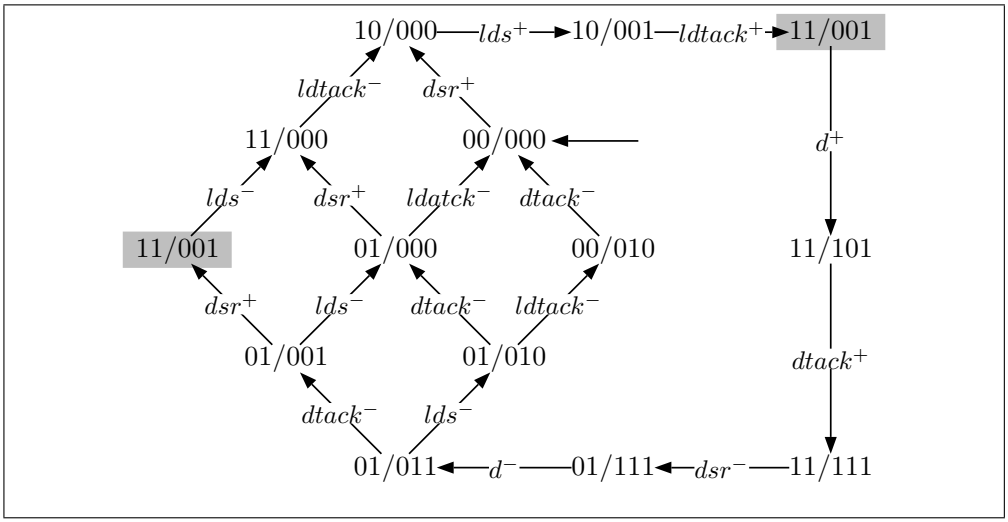
This is the STG model of the VME bus controller from Section 2.2.



Below the reachability graph and the corresponding state graph are drawn. The order of signals is *dsr ldatck / d dtack lds*. The two marked states have the same state vector 11/001 but enable different outputs, hence the STG has no CSC.
(Example 5.1 demonstrates how CSC is achieved for VME.) For another CSC related example, see Example 3.7.



Example 3.6: (continued on next page) VME Bus Controller: Reachability Graph, State Graph and CSC Conflict



ory to store its state properly.[4] The solution is therefore to increase this state memory with additional signals. Usually, these are internal signals which are introduced in a way such that the interaction with the environment is not affected, cf. Example 3.7. This topic is further discussed in Chapter 5.

Note that not all CSC-conflicts can be resolved; such CSC conflicts are called *irreducible*, see also Section 6.5 and Section 7.3.

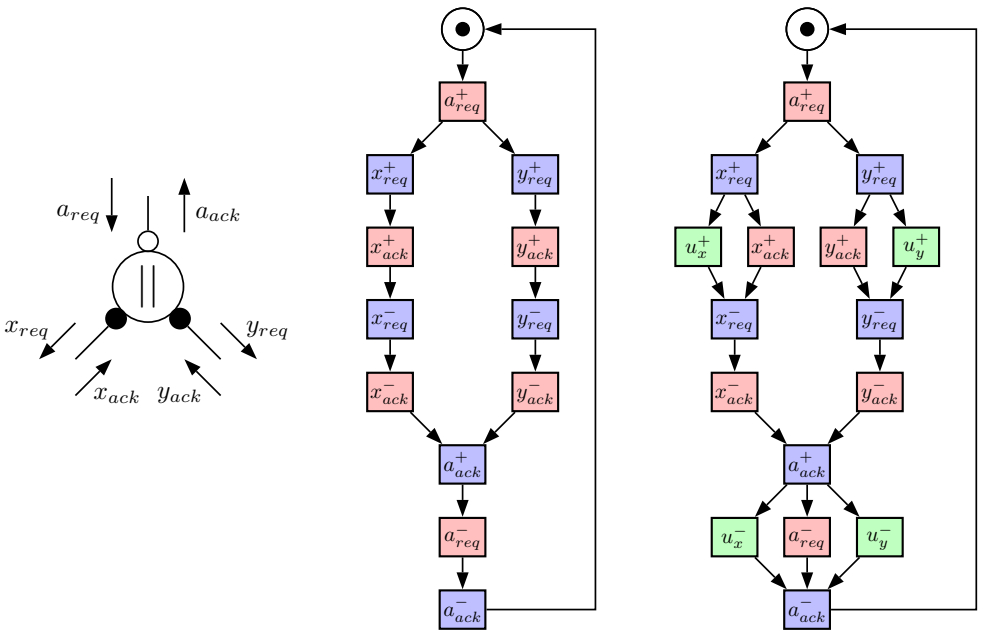
3.2.2 Parallel Composition, Renaming and Hiding

The identity of the transitions or places of an STG, as well as the names of the internal signals are not relevant. Hence, we regard STGs N and N' as equal if they are *externally isomorphic*, i.e. if they have the same input and output signals, and we can rename the internal signals of N and then map the transitions (places resp.) of the resulting STG bijectively onto the transitions (places resp.) of N' such that the weight function, the marking and the labelling are preserved. Altogether, the external signals are preserved while the internal signals might be renamed.

For the modular construction of STGs, the operations *hiding*, *relabelling* and *parallel*

[4] While in principle, n signals allow for 2^n different states, this state space cannot be used completely for asynchronous circuits, since the encoding of the states depends on the functionality of the circuit. In synchronous circuits, n flip-flops do actually allow for 2^n states, but in practice, often more flip-flops than absolutely needed are used to simplify the next-state logic.

Handshake *paralleliser* component (left) as STG (middle). The paralleliser component is quite similar to the sequencer component (see Section 2.2.3), except that the two active handshakes are performed in parallel.
The paralleliser has 6 signals, an input and an output for each channel; the corresponding STG shows the behaviour clearly. Observe the concurrent handshakes of the two active ports. The corresponding reachability graph has (not shown) 20 states – a first example for state explosion.



However, the STG does not have CSC: let M be the marking after firing a^+_{req} and let M' be the marking after firing then $x^+_{req}\, x^+_{ack}\, x^-_{req}\, x^-_{ack}$, i.e. the 'left' handshake only. The state vectors of M and M' are obviously equal and M activates x^+_{req} but M' not. There is an analogous conflict when only the 'right' handshake is performed.
In the right STG, CSC was solved by the introduction of two new internal signals; here, they are introduced fully concurrently, but it is also possible to insert them e.g. between x^+_{ack} and x^-_{req}. Furthermore, the right STG is a correct implementation of the middle one, according to Definition 5.2.

Example 3.7: Handshake Paralleliser: CSC Solving

composition are of interest. Given an STG N and a set H of signals with $H \cap In = \emptyset$, the *hiding of* H results in the STG:

$$N/H = (P, T, W, M_N, In, Out \setminus H, Int \cup H, l)$$

Given a bijection ϕ defined at least for the external signals of N, the *relabelling* of N is

$$\phi(N) = (P, T, W, M_N, \phi(In), \phi(Out), Int, \phi \circ l)$$

This assumes that, if necessary, the internal signals of N are renamed such that $Int \cap (\phi(In) \cup \phi(Out)) = \emptyset$ and ϕ is extended to be the identity on the internal signals otherwise.

Observe that hiding and relabelling preserve determinism as defined above and the same will apply for parallel composition. In particular, hiding does not change the identity of signals or removes them completely from the STG as it is done in other settings.

In reality, circuits are connected by connecting outputs of one circuit to inputs of another with a wire, while internal signals are not connected to other circuits at all.

This is captured formally in the following definition of *parallel composition* $\|$. Here, the distinction between input, output and internal signals has to be considered. The idea of parallel composition is that the composed systems run in parallel synchronising on common 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 one component can be an input of one or several other components, and in any case it is an output of the composition. Internal signals are unique for each component, and we assume that the sets of internal signals are disjoint; this can be achieved by a suitable renaming before the parallel composition.

A composition can also be ill-defined due to what e.g. Ebergen [Ebe92] calls *computation interference*; this is a semantic problem, and we will consider it later in Section 3.3.

Therefore, the *parallel composition* of STGs N_1 and N_2 is defined if $Out_1 \cap Out_2 = \emptyset$ and $Int_1 \cap Sig_2 = Int_2 \cap Sig_1 = \emptyset$. Then, 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 s in N_1 is 'seen' by N_2, i.e. it must be accompanied by an occurrence of s in N_2. Since we do not know a priori which $s^{\pm}$-labelled transition of N_2 will occur together with some $s^{\pm}$-labelled transition of N_1, we have to allow for each possible pairing.

Thus, the *parallel composition* $N = N_1 \parallel N_2$ is obtained from the disjoint union of N_1 and N_2 by combining each $s^{\pm}$-labelled transition t_1 of N_1 with each $s^{\pm}$-labelled transition t_2 from N_2 if $s \in A$; see Figure 3.2 for an example.

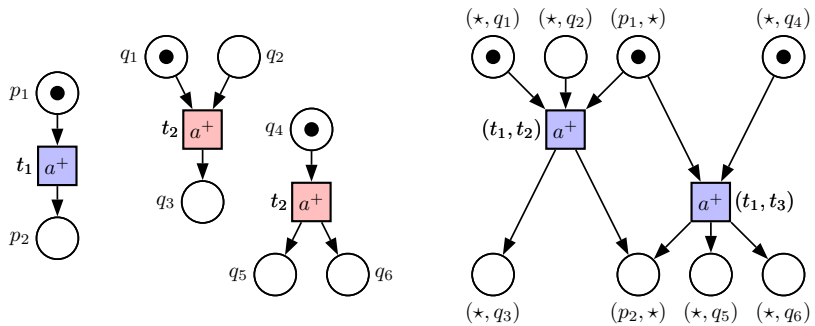


Figure 3.2: Example of a parallel composition. *left:* Part of STG N_1 with a single *output* edge a^+ in t_1. *middle:* Part of STG N_2 with two *input* edges a^+ in t_2 and t_3. *right:* Part of $N_1||N_2$ with two *output* edges a^+ in the combined transitions (t_1, t_2) and (t_1, t_3).

In the formal definition of parallel composition, $\star$ is used as a dummy element, which is formally combined with those transitions that do not have their label in the synchronisation set A. (We assume that $\star$ is not a transition or a place of any net.) Thus, N is defined by

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

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

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

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

$$
M_N = M_{N_1} \dot\cup M_{N_2}
$$

where $\dot\cup$ is the disjoint union, i.e.

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

$$
Int = Int_1 \cup Int_2 \qquad Out = Out_1 \cup Out_2 \qquad In = (In_1 \cup In_2) - Out
$$

It is not hard to see that parallel composition is associative and commutative up to external isomorphism and $||_{i\in I}N_i$ is defined if each $N_i||N_j$ is defined. Furthermore, one can consider the place set of the composition as the disjoint union of the place sets of the components. Therefore, we can consider markings of the composition (regarded as multisets) as the disjoint union of markings of the components as exemplified for M_N above; the latter makes clear what we mean by the restriction $M\big|_{P_i}$ for a marking M of the composition.

STGs together with the three operations defined above form a *circuit algebra* as defined in Dill's PhD thesis [Dil88]. In Figure 3.3, their laws are given in our notation.

$$
\begin{aligned}
&\text{(CA1)} : (N_1||N_2)||N_3 = N_1||(N_2||N_3) = N_1||N_2||N_3\\
&\text{(CA2)} : N_1||N_2 = N_2||N_1\\
&\text{(CA3)} : \phi_2(\phi_1(N)) = (\phi_2\circ\phi_1)(N)\\
&\text{(CA4)} : \phi(N_1||N_2) = \phi(N_1)||\phi(N_2) \quad {}^{(*)}\\
&\text{(CA5)} : id(N) = N\\
&\text{(CA6)} : (N/H_1)/H_2 = N/(H_1\cup H_2)\\
&\text{(CA7)} : N/\emptyset = N\\
&\text{(CA8)} : N_1/H_1||N_2/H_2 = (N_1||N_2)/(H_1\cup H_2)\\
&\qquad\qquad \text{if } H_i\cap Sig_{3-i} = \emptyset, i = 1,2\\
&\text{(CA9)} : \phi(N/H) = \phi'(N)/\phi'(H)\\
&\qquad\qquad \text{for } \phi = \phi'|_{Sig\setminus H} \text{ and } \phi(Sig\setminus H)\cap\phi'(H) = \emptyset \quad {}^{(*)}
\end{aligned}
$$

Figure 3.3: Laws of a circuit algebra. $^{(*)}$ only holds if both sides are defined.

For our further considerations we will use the properties (CA6) and (CA8). While the first is obvious, we will give a short proof for (CA8). Observe that $(N_1||N_2)/\{x\} = (N_1/\{x\})||N_2$ if $x\notin Sig_2$: since $x\notin Sig_2$, the transitions labelled with $x^{\pm}$ in N_1 are not paired with transitions of N_2 and it is therefore not important weather the hiding is done before or after the parallel composition. The main proof is by induction. Obviously, (CA8) is fulfilled for $H_1 = H_2 = \emptyset$. Let now (CA8) be fulfilled for some sets H_1 and H_2 and let w.l.o.g. $H_1' = H_1\cup\{x\}$ with $x\notin Sig_2 \Rightarrow$ $(N_1||N_2)/(H_1'\cup H_2) = (N_1||N_2)/(H_1\cup H_2\cup\{x\}) \overset{(CA6)}{=} ((N_1||N_2)/(H_1\cup H_2))/\{x\}$ $= ((N_1/H_1)||(N_2/H_2))/\{x\} = ((N_1/H_1)/\{x\})||(N_2/H_2) \overset{(CA6)}{=} (N_1/H_1')||(N_2/H_2)$.

We will come back to Dill's approach to asynchronous circuits in Section 5.5, where we compare his correctness notion to our one.

3.3 Decomposition

In this section, the STG decomposition algorithm of Vogler, Wollowski and Khangsah [VW02, VK06] is outlined, in order to understand the new contributions of this thesis properly. In the first part, the main idea of decomposition – making circuit synthesis from STGs faster – is explained. In the second part, the decomposition algorithm itself is presented. For more details and the corresponding proofs, see [VW02, VK06].

This algorithm is for *deterministic* STGs only. In particular, an STG must not have dynamic auto-conflicts (while structural ones are allowed). An algorithm for the decomposition of non-deterministic STGs is presented in Chapter 7.

3.3.1 Purpose

One of the most important problems in STG synthesis is the large computational effort: to perform synthesis for an STG N, one has to generate a representation of the state graph SG_N. As discussed above, SG_N can be exponentially large in the size of N itself. In particular, STGs often exhibit concurrent behaviour, which, on the one hand, makes the resulting circuits faster, but, on the other hand, blows up the state space. Furthermore, once SG_N is calculated, the computational effort to derive the final equations is greater than linear in $|SG_N|$.

Although practical implementations do not use naïve state space representations, there are hard constraints on the size of synthesisable STGs:

- Petrify generates a representation of SG_N based on binary decision diagrams (BDD) [Bry86]. In principle, BDDs can also be exponentially large, but on average their size is comparatively small. Nevertheless, Petrify can only handle STGs with about 20-30 signals; larger ones will usually lead to a memory overflow.

- Mpsat works on an unfolding prefix of the STG, which in most cases is quite small, in particular, if compared to the reachability graph. The actual synthesis algorithm is exponential in the number of signals. Due to a low constant factor, Mpsat can handle small examples very fast, but is unable to synthesise STGs with more than 40-50 signals.

- In general, it seems unlikely that there are fast and memory efficient algorithms for the synthesis of STGs, since even checking whether an STG is synthesisable is PSpace-complete (cf. Chapter 7).[5]

[5] This is no contradiction to decomposition combined with synthesis being fast: for many computationally hard problems like Sat, there are heuristic algorithms which perform quite well on

Decomposition tries to overcome these problems by splitting the original *specification* into several *components*. Together, they form an *implementation*, i.e. in the environment they are designed for, they show the same behaviour as the specification; this property is captured in a formal correctness notion, which is introduced below.

There are three advantages of decomposition:

- The components usually have less concurrency than the specification and in any case they are not larger. Their state graphs are therefore much smaller and since the components are synthesised separately, synthesis becomes much faster.
- Even if some components are not much smaller than the specification, this can at least help to reduce the peak memory usage during synthesis.
- Finally, for large examples (≥ 60 signals) decomposition makes synthesis possible in the first place. In Chapter 7, we will see benchmarks with more than 4000 signals which can be decomposed and synthesised.

Now, we come to the formal definition of the correctness notion for decompositions (see also Examples 3.8 and 3.9). This notion is for STGs without internal signals; in Chapter 5, an extended correctness notion for STGs with internal signals is presented.

Definition 3.4 (Correct Decomposition)
A collection of deterministic components $(C_i)_{i \in I}$ is a *correct decomposition* of (or simply *correct w.r.t.*) a deterministic STG N – also called *specification* – if $C = ||_{i \in I} C_i$ is defined, $In_C \subseteq In_N$, $Out_C \subseteq Out_N$ and there is an *STG-bisimulation* $\mathcal{B}$ between the markings of N and those of C with the following properties:

1. $(M_N, M_C) \in \mathcal{B}$
2. For all $(M, M') \in \mathcal{B}$, we have:
   - (N1) If $a \in In_N$ and $M[a^{\pm}\rangle\rangle M_1$, then either $a \in In_C$, $M'[a^{\pm}\rangle\rangle M_1'$ and $(M_1, M_1') \in \mathcal{B}$ for some M_1' or $a \notin In_C$ and $(M_1, M') \in \mathcal{B}$.
   - (N2) If $x \in Out_N$ and $M[x^{\pm}\rangle\rangle M_1$, then $M'[x^{\pm}\rangle\rangle M_1'$ and $(M_1, M_1') \in \mathcal{B}$ for some M_1'.
   - (C1) If $x \in Out_C$ and $M'[x^{\pm}\rangle\rangle M_1'$, then $M[x^{\pm}\rangle\rangle M_1$ and $(M_1, M_1') \in \mathcal{B}$ for some M_1.

average. The same is true for decomposition: in most cases, decomposition and synthesis are fast, but sometimes the final components cannot be synthesised and one has to synthesise the specification directly.

(C2) If $x \in Out_i$ for some $i \in I$ and $M' \big|_{P_i}[x^{\pm}\rangle\rangle$, then $M'[x^{\pm}\rangle\rangle$. (no *computation interference*)

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

In the most simple case, $(C_i)_{i\in I}$ consists of just one component C_1 (immediately implying (C2)); in this case, we say that C_1 is a *(correct) implementation* of N. $\triangle$

Essentially, the correctness notion is a *bisimulation* (i.e. specification and implementation can work side by side simulating the edges of each other) which respects the different roles of inputs and outputs as well as the special conditions of speed-independent asynchronous circuits:

- In general, the implementation is allowed to have fewer inputs and outputs than the specification.

  Outputs do not have to be in a component if they are never produced by N actually; this case is somewhat pathological, and the decomposition algorithm only produces implementations which contain all outputs of the specification. If an output can be produced by N, it has to be matched by C due to (N2).

  Inputs do not have to be in the implementation, even if they are actually possible in the specification. They can be omitted if they are not necessary to produce the proper behaviour and are just ignored then in (N1).

- Remarkably, there is no condition that requires a matching for an input occurring in the implementation. On the one hand, if the specification allows such an input in a matching marking, then the markings after the input must match again by (N1) due to determinism. On the other hand, there are very natural decompositions which allow additional input edges compared to the specification, and it does no harm to include these decompositions in our definition: since the specification also describes which inputs are or are not allowed for the environment, the additional inputs will actually never occur if the decomposition runs in an environment it is meant for,[6] see also Example 3.11

  As a consequence, the components might have behaviour and markings that never turn up if the components run in an appropriate environment; also, these markings do not appear in $\mathcal{B}$.

- (C2) ensures that the components can work together without disturbing each other, i.e. an output of one component is always expected by components listening to it.

[6] The additional inputs lead to states which in a way correspond to don't-care entries in a Karnaugh-diagram.

One might think that this requirement is a property of the parallel composition itself without relation to the specification. But since the implementation allows more behaviour due to additional inputs, the components might actually exhibit computation interference outside a proper environment.

3.3.2 Algorithm

Now, we come to the presentation of the decomposition algorithm itself. In [VW02, VK06], it was proven that it always calculates a correct decomposition of a deterministic specification. Confer Example 3.11, for the decomposition of the VME bus controller

STG decomposition works roughly as follows:

- A partition of the output signals of the specification STG N is chosen, such that some properties are fulfilled. For each set in this partition, a component producing these outputs will be generated.
- For each set of the partition, an *initial component* is produced as a copy of N but with unnecessary signals being *lambdarised*.
- The components are *reduced* separately, i.e. certain *reduction operations* are applied (mostly transition contractions).
- If all λ-transitions can be removed, a *final component* is produced. Otherwise, backtracking is performed: the reduction is restarted at the respective initial component in which some signals are *delambdarised*, i.e. their original label is restored. These steps are repeated till the reduction is successful.

In the rest of this section, these operations are explained in more detail.

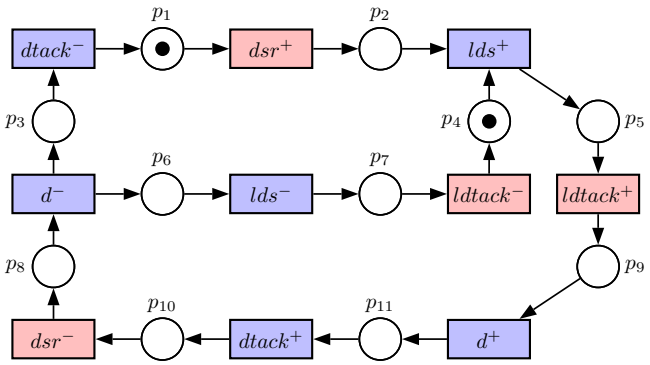
Initial Partition and Initial Components

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 $In_i \subseteq In \cup Out \setminus Out_i$, and furthermore:

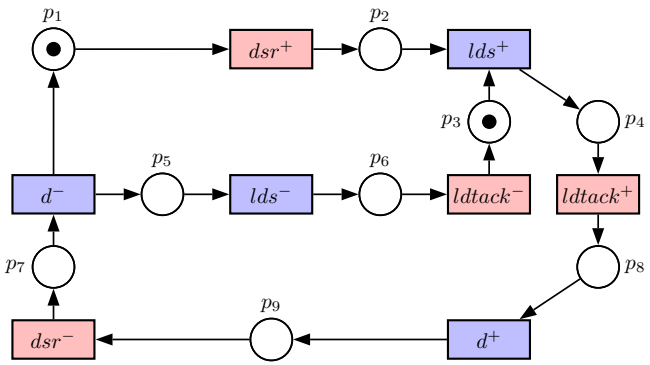
(F1) If signal s and output signal x of N are in structural conflict, then for each $i \in I$, $x \in Out_i$ implies $s \in In_i$ if $s \in In_N$ and $s \in Out_i$ if $s \in Out_N$.

The rationale for this is the following: clearly, a component responsible for output signal x must at least 'see' any signal that could be in dynamic conflict

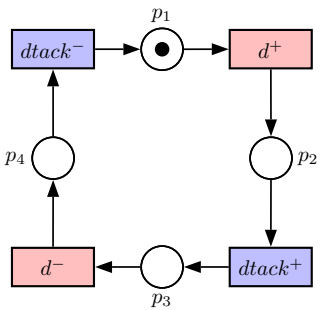
Consider the VME bus controller from Example 3.6, which is shown with all places below.



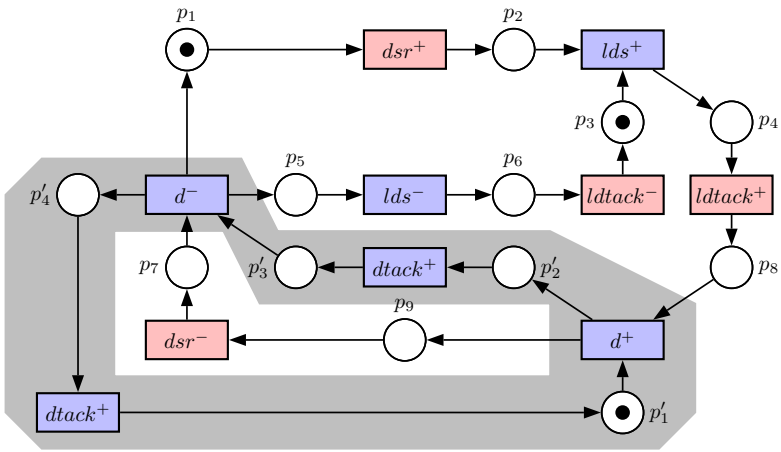
Below is an example of a decomposition for this STG (generated with DesiJ) which is correct according to Definition 3.4. Observe that the input d of component C_2 is produced by component C_1 and is an output of the parallel composition.



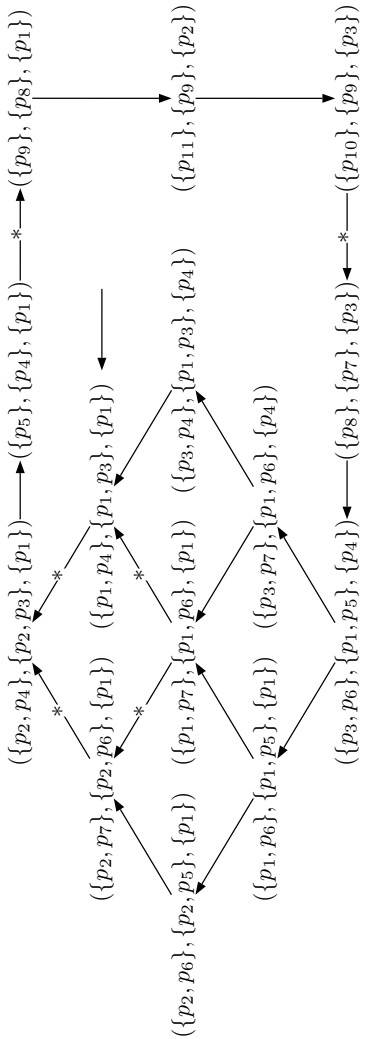
Example 3.8: Decomposition of VME Bus Controller 1 (continued on next page)



The following is the parallel composition $C = C_1||C_2$ of the two components. The parts of C_2 are marked gray, the respective places are written as p'_1 etc. Remarkably, C looks quite different to the specification N above: Now, C_1 and C_2 are mostly concurrent. In particular, after the firing of d^+ in C, not only $dtack^-$ is activated, but also dsr^-; as described above this additional activated input is allowed in a correct decomposition.



This is the corresponding STG-bisimulation $\mathcal{B}$ for the decomposition of VME from Example 3.8. The nodes are the elements of $\mathcal{B}$, an arrow indicates which element causes another element to be in $\mathcal{B}$. Arrows with a * denote that the corresponding reason is (N1), without * it is (N2) or (C1). The element with the single arrow is for the initial markings.



Example 3.9: Decomposition of VME Bus Controller 2

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 *syntactical trigger* of t', i.e. $t^\bullet \cap {}^\bullet t' \neq \emptyset$, then the signal of t is in $In_i \cup Out_i$.

The latter signal $l(t)$ 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 unique least set In_i such that also (F2) is satisfied. In many practical cases, this canonical feasible partition will have one (In_i, Out_i) for each output signal of N.

All other feasible partitions can be obtained from this finest partition by either adding input signals to a component or by merging some (In_1, Out_1) and (In_2, Out_2) to $((In_1 \cup In_2) \setminus (Out_1 \cup Out_2), Out_1 \cup Out_2)$ (an output of a component might be an input of another one).

Given an initial feasible partition, the *initial components* $(C_i)_{i \in I}$ are defined as follows: each C_i is a copy of N with a modified labelling according to the partition, i.e.

$$C_i = (P, T, W, M_N, In_i, Out_i, l|_{Sig_i} \cup l_\lambda|_{Sig_N - Sig_i})$$

where $l_\lambda \equiv \lambda$ is restricted to the signals $Sig_N - Sig_i$ not occurring in the respective component C_i; these signals are called *lambdarised* in the initial component.

In the context of decomposition and implementation, we can therefore distinguish the origin of dummy or λ-transitions. A dummy which is already there in a specification STG is called *spec-dummy*. If it is part of an intermediate component due to lambdarisation of some signal s, it is called *component-dummy* or *deco-dummy*; if one wants to stress the type of s in the specification, it is called e.g. *output-dummy* if s is an output of the specification.

For decomposition as considered in [VW02, VK06], the specification has to be deterministic; hence, no spec-dummies are allowed then. This condition is relaxed in Chapter 7.

Reduction Operations

There are three reduction operations defined:

- **Transition contraction** A λ-transition is removed and the adjacent places are merged, see below for the exact definition.
- **Deletion of an implicit place** It is often the case that after a transition contraction implicit places are produced. Such places may prevent further transition contractions and should be deleted before the reduction proceeds.
- **Deletion of a redundant transition** There are two kinds of *redundant transitions*.

  First, if there are two transitions with the same label which are connected to every place in the same way, one of them can be deleted without changing the traces of the STG.

  Second, a λ-labelled transition t with $W(p,t) = W(t,p)$ for all places p can also be deleted, since its firing does not change the marking and is not visible on the level of traces; observe, that this is valid for any marking of the adjacent places.

Intuitively, the contraction of a transition t removes t from the net, together with its adjacent places ${}^\bullet t \cup t^\bullet$, and adds new places to the net, corresponding to the elements of ${}^\bullet t \times t^\bullet$. Each new place $(p,q) \in {}^\bullet t \times t^\bullet$ inherits the connectivity of both, p and q (except that t is no longer in the net), and its initial marking is the total number of tokens which were initially present in p and q.

In the formal definition of a transition contraction, labelled Petri nets instead of STGs are considered, since we also apply contractions to these in Section 3.4.

Definition 3.5 (Transition contraction)
Let N be a labelled Petri net and $t \in T$ with $l(t) = \lambda$. If t is not incident to an arc with weight greater than 1 and ${}^\bullet t \cap t^\bullet = \emptyset$, the *contraction of* t results in the net N' with:

$$\begin{aligned}
T' &= T - \{t\} \\
P' &= \{(p,\star) \mid p \notin {}^\bullet t \cup t^\bullet\} \cup \{(p_1,p_2) \mid p_1 \in {}^\bullet t, p_2 \in t^\bullet\} \\
W'((p_1,p_2),t') &= W(p_1,t') + W(p_2,t') \\
W'(t',(p_1,p_2)) &= W(t',p_1) + W(t',p_2) \\
M'((p_1,p_2)) &= M(p_1) + M(p_2)
\end{aligned}$$

In this definition $\star \notin P \cup T$ is a dummy element used to make all places of N' to be pairs; we assume $M(\star)$, $W(\star, t')$ and $W(t', \star)$ to be 0.

Sometimes, we write $\overline{N}^t$ instead of N', and if more than one contraction is applied to a net N, e.g. for the transitions t_1 and t_2 this is denoted by $\overline{N}^{t_1,t_2}$.

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*.

The 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*).

We say that markings M of N and M' of N' satisfy the *marking equality* if for every place (p_1, p_2) of N': $M'((p_1, p_2)) = M(p_1) + M(p_2)$. △

The nets before and after a reduction operation are closely related:

Theorem 3.6 (Reduction, Simulation and Language [VW02, VK06])
Let N' be obtained from N by ...

(1) ... the secure contraction of some transition.

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

If the contraction is type-1 secure, $\mathcal{S}$ is a bisimulation.

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

(2) ... the deletion of a redundant place.

Then $\mathcal{S} = \{(M, M|_{P'}) | M \in [M_N\rangle\}$ is a bisimulation between N and N', and we define $\mathcal{S}' = \mathcal{S}^{-1}$.

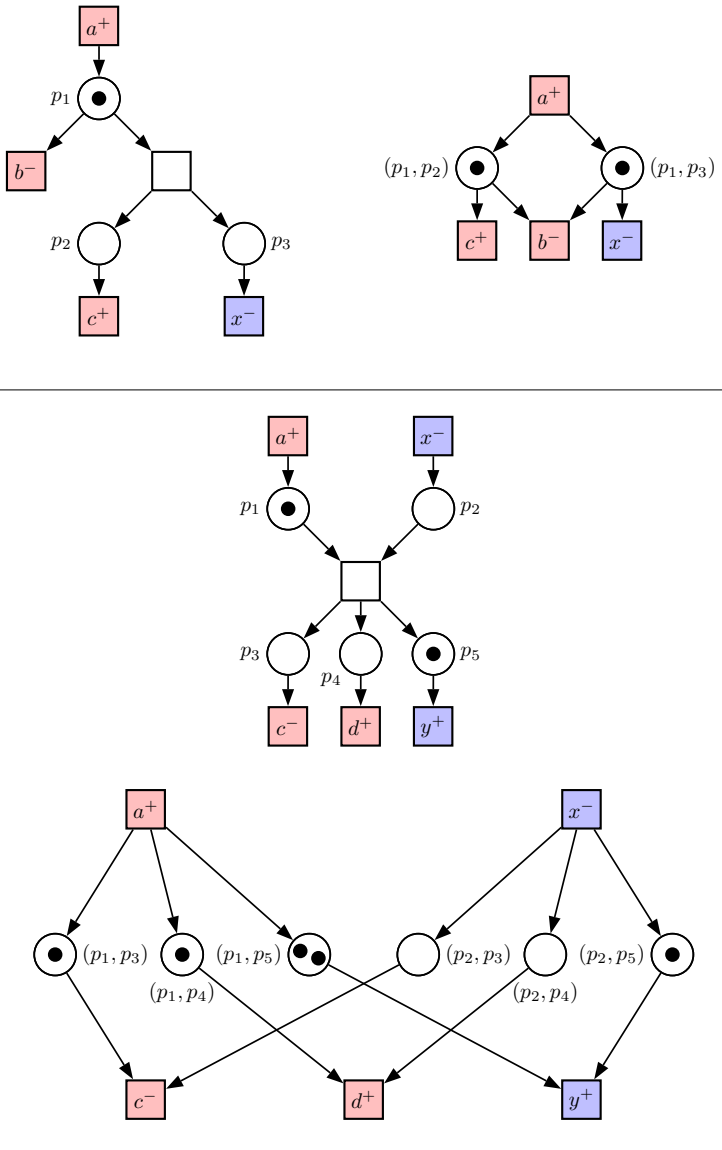
(3) ... the deletion of a redundant transition.

Then the identity $\mathcal{S}$ over $[M_N\rangle$ is a bisimulation between N and N', and we define $\mathcal{S}' = \mathcal{S}^{-1}$.

(4) In all cases, N and N' have the same language.

Note that (4) implies that all reduction operations preserves the language (and in particular language based consistency).

Below one can see two examples of a transition contraction.



Example 3.10: Transition Contraction

Finally, we define the *place projection* which gives us some information about the structure of places after a contraction.

Definition 3.7 (Place Projection)
Let N' be obtained from N by a sequence of transition contractions. Every place p' of N' is a pair, where each element is another pair etc., down to the level of places from P. The function $\Phi_N^{N'}$ assigns to every place of N' the multiset of places of N occurring in p'. We write $\Phi_N^{N'}(p')(p)$ to denote the number of occurrences of p within p'. $[\Phi_N^{N'}(p')]$ denotes the *support* of $\Phi_N^{N'}(p')$, i.e. the set of places occurring in this multiset. △

For example:

$$\begin{aligned}\Phi_N^{N'}(((p_1,p_2),(p_1,p_3))) &= \{2 \cdot p_1, p_2, p_3\}\\ \Phi_N^{N'}(((p_1,p_3),\star)) &= \{p_1,p_3\}\\ \Phi_N^{N'}(((p_1,p_3),\star)) &= \{(p_1,p_3)\} \text{ if } (p_1,p_3) \in P\end{aligned}$$

Backtracking

As it was already mentioned, it is possible that not every component-dummy of an initial component can be contracted by the decomposition algorithm. There are three reasons for this:

- The contraction is not defined (e.g. because ${}^\bullet t \cap t^\bullet \neq \emptyset$).
- The contraction is not *secure* (then the language of the STG might change).
- The contraction introduces a new dynamic auto-conflict, i.e. a new source of non-determinism which was not present in the specification is introduced; the interpretation is that the component has not enough information (viz. input signals) to properly produce its outputs).

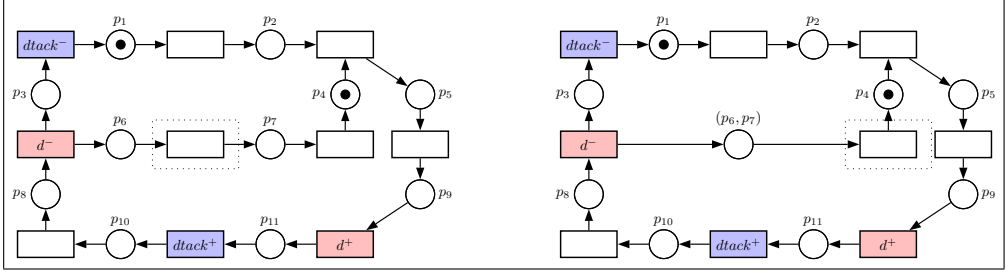
If none of the described reduction operations are applicable, but the component still has some λ-labelled transitions, *backtracking* is applied: one of these λ-labelled transitions is chosen and the corresponding original signal is *delambdarised*. Delambdarising adds this signal as an input to the respective initial partition and the new corresponding initial component is reduced from the beginning. This cycle of reduction and backtracking is repeated until all λ-labelled transitions of the initial component can be contracted. This means that backtracking is only needed to detect these additional input signals; if they are known in advance, one can perform decomposition

completely without backtracking. (In the worst case, all the lambdarised signals are delambdarised.)

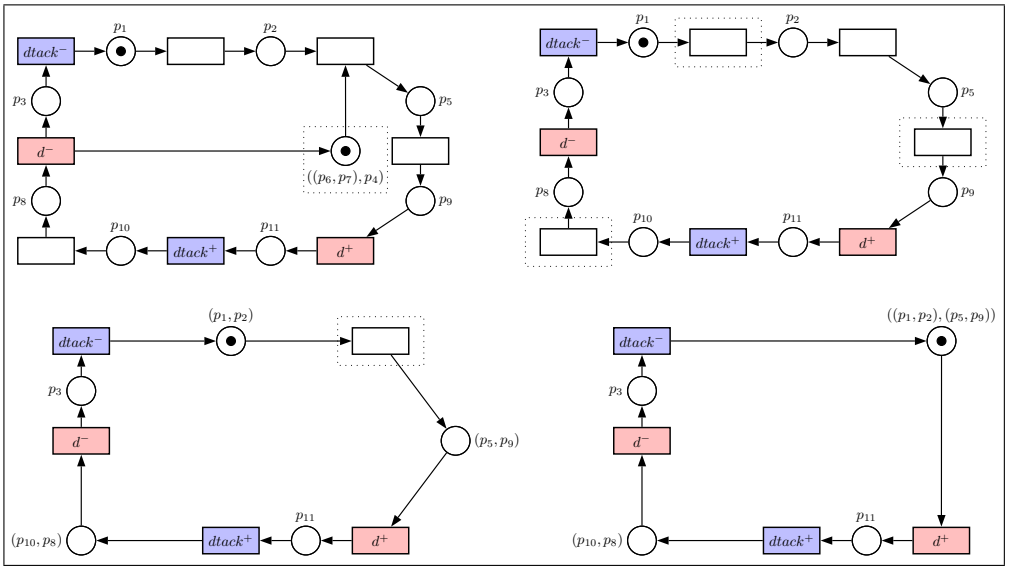
Checking for dynamic auto-conflicts is expensive, and instead backtracking is performed if a contraction generates a new *structural* auto-conflict. But it is also possible to decompose with the *risky strategy*, i.e. each encountered structural auto-conflict is assumed as not being dynamic. This strategy is feasible, because in [VK06] it was shown that dynamic conflicts are preserved in some sense during reduction. Hence, if the final component has no dynamic conflicts – which can be checked during synthesis – the risky strategy was successful and the decomposition is correct. Unfortunately, the risky approach is not very useful, i.e. it is not much faster than the normal approach and the components are not much smaller, cf. also the discussion in Section 6.7.

Furthermore, in Chapter 7 it is shown under which conditions backtracking can be avoided even for λ-transitions which are not contractible due to syntactical reasons, e.g. if the contraction is not secure.

Let us consider again the decomposition of the VME bus controller from Examples 3.8 and 3.9. We will show how the component C_2 producing output *dtack* is generated. The initial partition is $\{(\{dsr, ldtack\}, \{d, lds\}), (\{d\}, \{dtack\})\}$. Therefore, the initial component for C_2 looks like the first STG below.
To generate the final component, 7 reduction operations are performed. They are marked with a dotted rectangle; if a transition is marked it is contracted. In the third STG, the marked place is implicit and deleted. (It is a shortcut place, see Definition 4.3.) In the fourth STG, three contractions are performed at once. Backtracking is not needed here. Observe that the places are simplified here: $(p_1, \star)$ is just written as p_1.



Example 3.11: Decomposition of VME Bus Controller 3 (continued on next page)



3.4 Properties of Transition Contractions

In this section, some additional properties of transition contractions are investigated. Originally, they were distributed over several publications. They are somewhat related and therefore presented together. Since they are also valid for ordinary Petri nets, we only talk about such. In this context, transition contractions are applied to arbitrary transitions.

A simulation as in Definition 3.1 is defined for labelled Petri nets, STGs resp. only, and *traces* of one net have to be matched by the other net. Here, we will define and use simulations on the transition level, which require that *firing sequences* have to be matched in some sense. To allow this, the corresponding nets have to have (nearly) the same set of transitions; this fits well to reduction where the set of transitions is just decreased but not modified otherwise.

Definition 3.8 (Transition Simulation)
Let N and N' be Petri nets with $T' \subseteq T$. A relation $\mathcal{S}$ between the reachable markings of N and N' is a *transition simulation* between N and N' if:

(1) $(M_N, M_{N'}) \in \mathcal{S}$;

(2) $(M, M') \in \mathcal{S}$ and $M[v\rangle M_1$ with $v \in T^*$ implies $M'[v|_{T'}\rangle M_1'$ and $(M_1, M_1') \in \mathcal{S}$.

A relation $\mathcal{S}'$ between the reachable markings of N' and N is a *transition simulation* between N' and N if:

(1) $(M_{N'}, M_N) \in \mathcal{S}'$;

(2) $(M', M) \in \mathcal{S}'$ and $M'[v'\rangle M'_1$ with $v' \in T'^*$ imply that $M[v\rangle M'_1$ with $v|_{T'} = v'$ and $(M'_1, M_1) \in \mathcal{S}'$.

$\mathcal{S}$ is a *transition-bisimulation* between N and N' if it is a simulation in both directions. △

A special case of a transition simulation arises if $T = T'$.

Lemma 3.9
For two STGs N and N', the following properties are equivalent:

(1) There is a transition-simulation from N to N' and vice versa

(2) N and N' are transition-bisimilar

(3) $FS(N) = FS(N')$

Proof.
(3) → (2): $\mathcal{B} = \{(M, M') \mid M_N[v\rangle M,\ M_{N'}[v\rangle M' \text{ for some } v \in T^*\}$ is a transition-bisimulation between N and N'. Clearly, $(M_N, M_{N'}) \in \mathcal{B}$ (choose $v = \lambda$).

Let $(M, M') \in \mathcal{B}$ and w.l.o.g. $M[v'\rangle M_1$ with $v' \in T^*$. Due to the definition of $\mathcal{B}$, $M_N[v\rangle M$ and $M_{N'}[v\rangle M'$ for some $v \in T^*$. Hence, $M_N[v\,v'\rangle M_1$, and since $FS(N) = FS(N')$, also $M_{N'}[v\,v'\rangle M'_1$ with $(M_1, M'_1) \in \mathcal{B}$ for some M'_1.

(2) → (1): obvious.

(1) → (3): $v \in FS(N)$, i.e. $M_N[v\rangle$, implies $M_{N'}[v\rangle$ due to the simulation between N and N'. Hence, $FS(N) \subseteq FS(N')$. Analogous for the other direction, and therefore the claim follows. □

Transitivity of simulations is kind of folklore. Here, we provide and prove it on the level of transition simulations.

Lemma 3.10 (Transitivity of Transition Simulations)
Let N, N_1 and N_2 be nets with $T_2 \subseteq T_1 \subseteq T$. If $\mathcal{S}_1$ is a transition simulation between N and N_1, $\mathcal{S}_2$ between N_1 and N_2, $\mathcal{S}'_1 \subseteq \mathcal{S}_1^{-1}$ between N_1 and N and $\mathcal{S}'_2 \subseteq \mathcal{S}_2^{-1}$ between N_2 and N_1, then $\mathcal{S} = \mathcal{S}_1 \circ \mathcal{S}_2$ is a transition simulation between N and N_2 and $\mathcal{S}' = \mathcal{S}'_2 \circ \mathcal{S}'_1$ is one between N_2 and N with $\mathcal{S}' \subseteq \mathcal{S}^{-1}$ (cf. Figure 3.4).

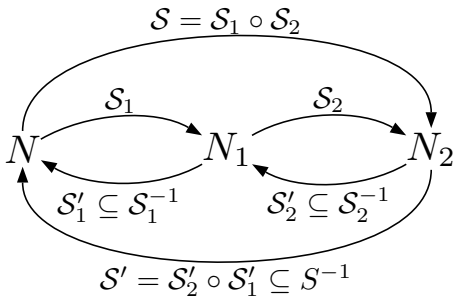


Figure 3.4: Transitivity of Simulations

Proof. Obviously, $(M_N, M_{N_1}) \in \mathcal{S}_1$ and $(M_{N_1}, M_{N_2}) \in \mathcal{S}_2$ implies $(M_N, M_{N_2}) \in \mathcal{S}$.
Let now $(M, M'') \in \mathcal{S}$. Hence, there is some M' such that $(M, M') \in \mathcal{S}_1$ and $(M', M'') \in \mathcal{S}_2$. Then, $M[v\rangle M_1$ implies $M'[v|_{T_1}\rangle M_1'$ with $(M_1, M_1') \in \mathcal{S}_1$; this implies $M''[v|_{T_2}\rangle M_1''$ with $(M_1', M_1'') \in \mathcal{S}_2$. Therefore, $(M_1, M_1'') \in \mathcal{S}$. With analogous reasoning, $\mathcal{S}'$ is a transition simulation between N_2 and N. Furthermore, $\mathcal{S}' = \mathcal{S}_2' \circ \mathcal{S}_1' \subseteq \mathcal{S}_2^{-1} \circ \mathcal{S}_1' \subseteq \mathcal{S}_2^{-1} \circ \mathcal{S}_1^{-1} = (\mathcal{S}_1 \circ \mathcal{S}_2)^{-1} = \mathcal{S}^{-1}$. □

Now, we can give a corollary of Theorem 3.6 related to transition simulations.

Corollary 3.11 (Reduction and Transition Simulation)
Let N be a net and let N' be obtained from N by ...

(1) ... the contraction of some transition t. Then $\mathcal{S} = \{(M, M') \mid M$ and M' satisfy the marking equality$\}$ is a transition simulation between N and N' and there is a transition simulation $\mathcal{S}' \subseteq \mathcal{S}^{-1}$ between N' and N.

(2) ... the deletion of a redundant place.

Then $\mathcal{S} = \{(M, M|_{P'}) | M \in [M_N\rangle\}$ is a transition bisimulation between N and N', and we define $\mathcal{S}' = \mathcal{S}^{-1}$.

(3) ... the deletion of a loop-only transition.

Then the identity $\mathcal{S}$ over $[M_N\rangle$ is a transition bisimulation between N and N', and we define $\mathcal{S}' = \mathcal{S}^{-1}$.

Proof. (1) and (3) follow from Theorem 3.6 for the following labelling l of N and N'.

$$l(t') = \begin{cases} t' & \text{for } t' \neq t \\ \lambda & \text{for } t' = t \end{cases}$$

(2) follows for the labelling $l(t) = t$ for all $t \in T = T'$. □

Note that (3) is not valid for the deletion of duplicate transitions, this case has to be handled separately if needed.

As an analogy to the marking equality for a single transition contraction, we show that the *extended marking equality* holds after a sequence of contractions.

Proposition 3.12 (Extended marking equality)
Let N be a Petri net and let N' be obtained from it by a sequence of secure contractions of some transitions. Then there is a transition simulation $\mathcal{S}$ from N to N' and a transition simulation $\mathcal{S}' \subseteq \mathcal{S}^{-1}$ from N' to N such that for every $(M, M') \in \mathcal{S}$ and every place p' of N', the extended marking equality *holds:*

$$M'(p') = \sum_{p \in P} \Phi_N^{N'}(p')(p) \cdot M(p) \quad .$$

Proof. Let $N_0 = N$, and for $1 \leq i \leq n$, let N_i be the net after the i-th transition contraction, with $N_n = N'$. Proposition 3.6 implies that there is a transition simulation $\mathcal{S}_i$ between N_{i-1} and N_i and a transition simulation $\mathcal{S}'_i \subseteq \mathcal{S}_i^{-1}$ between N_i and N_{i-1}. Recall that for every $(M, M') \in \mathcal{S}_i$, M and M' fulfil the marking equality. Let now $\mathcal{S}^i = \mathcal{S}_1 \circ \mathcal{S}_2 \circ \ldots \circ \mathcal{S}_i$. Repeated application of Lemma 3.10 gives that for each i, $\mathcal{S}^i$ is a transition simulation between N and N_i; in particular $\mathcal{S} = \mathcal{S}^n$ is a transition simulation between N and N', and there is a transition simulation $\mathcal{S}' \subseteq \mathcal{S}^{-1}$ between N' and N.

Now, we show by induction that, for each i and for every place p' of N_i, if $(M, M') \in \mathcal{S}^i$ then $M'(p') = \sum_{p \in P} \Phi_N^{N_i}(p')(p) \cdot M(p)$. For $i = 1$, this is directly implied by the marking equality for $\mathcal{S}_1 = \mathcal{S}^1$. Now assume that the claim is fulfilled for some i. (M_0^i denotes the initial marking of N_i.) Let $(M, M_{i+1}) \in \mathcal{S}^{i+1}$ due to $(M, M_i) \in \mathcal{S}^i$ and $(M_i, M_{i+1}) \in \mathcal{S}_{i+1}$. For $(p_1, p_2) \in P_{i+1}$, we obtain ($i = n$ proves the claim):

$$\begin{aligned} & \sum_{p \in P} \Phi_N^{N_{i+1}}((p_1, p_2))(p) \cdot M(p) \\ = & \sum_{p \in P} \Phi_N^{N_i}(p_1)(p) \cdot M(p) + \sum_{p \in P} \Phi_N^{N_i}(p_2)(p) \cdot M(p) \quad \text{(def. of } \Phi) \\ = & \; M_i(p_1) + M_i(p_2) \quad \text{(induction)} \\ = & \; M_{i+1}((p_1, p_2)) \quad \text{(marking equality for } \mathcal{S}_{i+1}) \quad \square \end{aligned}$$

Proposition 3.13
Secure transition contractions and deletions of implicit places and redundant transitions preserve 1-liveness, liveness and reversibility.

Proof. It suffices to show the claim for one application of these operations. If the redundant transition is a duplicate transition, the claim is obviously true; thus we only have to consider the deletion of loop-only transitions.

By Corollary 3.11, there is a transition simulation $\mathcal{S}$ between N and N' and a transition simulation $\mathcal{S}' \subseteq \mathcal{S}^{-1}$ between N' and N. Let u' be a firing sequence of N' such that $M_{N'}[u'\rangle M_1'$; then $M_N[u\rangle M_1$ for some firing sequence u of N such that $(M_1', M_1) \in \mathcal{S}'$.

Assume N is live. To show the liveness of N', suppose that one wants to enable some transition t' in N' starting from M_1' (note that $t' \neq t$ since t has been contracted). Since N is live, one can enable t' in N starting from M_1 by some transition sequence w: $M_1[w\rangle M_2[t'\rangle$. Since $(M_1, M_1') \in \mathcal{S}'^{-1} \subseteq \mathcal{S}$, $M_1'[w|_{T'}\rangle M_2'[t'\rangle$, which proves the liveness of N'. Preservation of 1-liveness follows with the same argumentation for $M_1' = M_N$.

If N is reversible then $M_1[v\rangle M_N$ for some transition sequence v of N. Therefore, $M_1'[v|_{T'}\rangle M_2'$ such that $(M_N, M_2') \in \mathcal{S}$.

If the applied operation is a contraction then, by definition of $\mathcal{S}$, M_N and $M_{N'}$ as well as M_N and M_2' fulfil the marking equality, i.e. the following equations hold for each place (p, q) of N':

$$\begin{aligned} M_{N'}((p,q)) &= M_N(p) + M_N(q) \\ M_2'((p,q)) &= M_N(p) + M_N(q)\ , \end{aligned}$$

If the applied operation is an implicit place deletion, we get by definition of $\mathcal{S}$, $M_{N'}(p') = M_N|_{P'}(p') = M_2'(p')$ for all places $p' \in P'$. If the applied operation is a loop-only transition deletion, by definition of $\mathcal{S}$, $M_{N'}(p') = M_N(p') = M_2'(p')$ for all places $p' \in P'$.

In all cases, $M_2' = M_{N'}$ and N' is reversible. □

3.5 Some Considerations about Consistency

As shortly mentioned above (p. 43), state based consistency (*sbc*) and language based consistency (*lbc*) do not coincide in general. In this section, we give some examples for this, a sufficient condition for coincidence and a simple method to convert an *lbc* STG into an *sbc* one.

We start with some easy observation about *lbc* STGs. For an *lbc* STG N, one can define an *initial state vector* sv_{M_N}, which represents the initial value of each signal. If the first edge of a signal s is s^+ (s^-), we have $sv_{M_N}(s) = 0$ ($sv_{M_N}(s) = 1$). Due to the definition of *lbc*, this vector is uniquely defined for every signal which actually can occur in N. The value for the 'dead' signals can be chosen arbitrarily; to achieve uniqueness, one can define it to be 0.

Now, have a look at Figure 3.5. The STG on the left hand side is clearly *lbc*, but the empty marking (reached via firing of either transition) corresponds to the states $(x, y) = (0, 1)$ and $(x, y) = (1, 0)$, a violation of *sbc*. This is also the case for the second STG, provided that after firing the first x^+ or y^+ no other $x^\pm$ or $y^\pm$ occurs. However, if an x^- transitions can be fired within the cloud, firing y^+ first instead of x^+ would lead to a violation of *lbc* (and of *sbc*).

The problem in this example is that firing x^+ or y^+ leads to a violation of *sbc* which becomes not apparent afterwards on the language level. This leads to the conjecture that *lbc* and *sbc* coincide for *signal-live* STGs. An STG N is *signal-live* if for every signal s and for every reachable marking M, there is a trace v such that $M[v\, s^\pm\rangle\rangle$. Clearly, a live STG is also signal-live, but not vice versa.

Proposition 3.14
A signal-live lbc *STG is also* sbc.

Proof. We define a state assignment sv for N as follows:

$$\text{for } M \in [M_N\rangle \text{ let } sv_M = sv_{M_N} + \mathit{codeChange}(v) \text{ with } M_N[v\rangle\rangle M$$

Since N is *lbc*, $sv_M(s) \in \{0, 1\}$ for every reachable M and every signal s ($*$): for $sv_{M_N}(s) = 0$, the first edge of s is s^+ changing the value of s from 0 to 1, the next edge has to be s^- changing the value from 1 to 0, and so on (analogous for $sv_{M_N}(s) = 1$).

sv is well-defined, i.e. the value of sv_M does not depend on the firing sequence v reaching M. Assume the opposite: there are two traces v_1 and v_2 with $M_N[v_1\rangle\rangle M$ and $M_N[v_2\rangle\rangle M$ such that $\mathit{codeChange}(v_1) \neq \mathit{codeChange}(v_2)$, differing for some signal s.

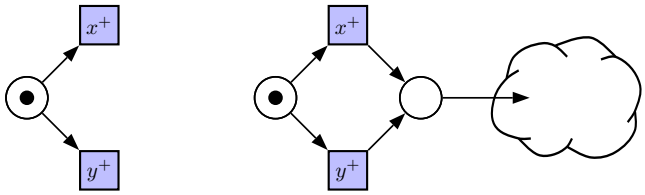


Figure 3.5: Two *lbc* but not *sbc* STGs. The cloud does not contain any $x^\pm$ or $y^\pm$ labelled transitions.

Let $sv_{M_N}(s) = 0$. Then, $(*)$ implies $codeChange(v_1)(s), codeChange(v_2)(s) \neq -1$. W.l.o.g. let $codeChange(v_1)(s) = 0$ and $codeChange(v_2)(s) = 1$. Since N is signal-live, there is a firing sequence v not containing s with $M_N[v_1\rangle\rangle M[vs^+\rangle\rangle$. However, also $M_N[v_2\rangle\rangle M[vs^+\rangle\rangle$, a contradiction to $(*)$. The case $sv_{M_N}(s) = 1$ is analogous.
Now, $M[t\rangle M'$ with e.g. $l(t) = s^+$ implies,

$$\begin{aligned} sv_{M'} &= sv_{M_N} + codeChange(vt) \text{ (for some } v \text{ with } M_N[v\rangle M) \\ &= sv_{M_N} + codeChange(v) + codeChange(s^+) \\ &= sv_M + codeChange(s^+) \end{aligned}$$

In particular, $sv_{M'}(s) = sv_M(s) + 1$, i.e. $sv_M(s) = 0$ and $sv_{M'}(s) = 1$ (analogous for the other cases). □

As already discussed, liveness and therefore also signal-liveness are sensible assumptions on STGs. However, it is possible to transform *every lbc* STG into an equivalent *sbc* one, if these assumptions are not fulfilled. For this purpose, *level places* are inserted.

Definition 3.15 (Insertion of Level Places)
Let N be an *lbc* STG. The insertion of level places yields the STG N' with: (cf. Figure 3.6)

$$\begin{aligned} P' &= P + \{s^0, s^1 \mid s \in Sig_N\} \\ T' &= T \\ W'(s^0, t) = W'(t, s^1) &= 1 \text{ for } l(t) = s^+ \\ W'(s^1, t) = W'(t, s^0) &= 1 \text{ for } l(t) = s^- \\ W' &= W \text{ otherwise} \end{aligned}$$

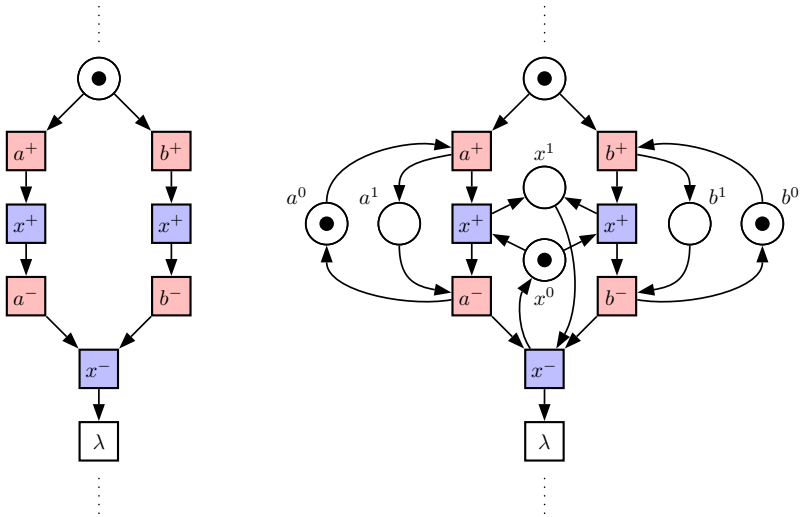


Figure 3.6: Level place insertion. Observe that there are exactly two level places per signal, even if more than two transitions are labelled with a respective signal edge; furthermore, λ-transitions are not connected to new places.

$$
\begin{aligned}
M_{N'}(s^0) &= 1 - sv_{M_N}(s) \\
M_{N'}(s^1) &= sv_{M_N}(s) \\
M_{N'} &= M_N \text{ otherwise} \\
In' = I \quad Out' &= Out \quad Int' = Int \\
l' &= l
\end{aligned}
$$

The additional level places explicitly encode the state vector of an STG: $M(s^0) = 1$ if the value of s is 0 under M, and $M(s^1) = 1$ if the value of s is 1 under M. The following proposition states that N and N' are equivalent in a very strong sense.

Proposition 3.16
Let N' be the result of a level place insertion into an lbc *N.*

(1) $FS(N) = FS(N')$.

(2) N' is sbc.

Proof.

(1) Let sv_{M_N} be the initial state vector of N. The value for 'dead' signals can be chosen arbitrarily. It is easy to show that

$$\begin{aligned}\mathcal{B} = \{(M, M') \mid\ & M'|_P = M,\ M_N[v\rangle M,\\ & \forall s \in Sig.\, M'(s^1) = (sv_{M_N} + codeChange(v))(s) \wedge\\ & M'(s^0) + M'(s^1) = 1\}\end{aligned}$$

is a transition bisimulation. The claim follows then from Lemma 3.9.

(2) The state encoding of N' can be read off directly from the inserted level places, i.e. $sv_M(s) = M(s^1)$ (for all signals s) is a feasible state encoding. □

In fact, it is not needed to add level places for all signals to get an *sbc* STG – it is sufficient to insert them just for the signals which are not signal-live, or even only for the ones which cause problems. In any case, the insertion leads to a *state duplication* in the state graph, i.e. the states which can be reached by traces with different code change, are split into several states. This can also be done on the fly while generating the state graph without modifying the underlying STG. There is the strong conjecture that the level place insertion does not introduce more states than necessary while preserving the firing sequences.

Level places can also be inserted into non-*lbc* STGs; in this case they enforce consistency but also change the behaviour of the STG.

Summing up, *lbc* and *sbc* coincide for practical STGs, and if this not the case *sbc* can be achieved easily without changing the behaviour.

Chapter 4

Determinate Decomposition

The decomposition algorithm is non-deterministic in the sense that any of the valid operations can be performed at a given time. So one might not expect the same results (i.e. final components), when performing the operations in a different order.

In particular, one might find during the processing of a component that additional signals are relevant; then, backtracking has to be performed for one of them. As a result, even in simple cases, the order of operations may influence for which signals this backtracking is performed, resulting in different components as shown in [VW02, Fig. 7]. Since this does not give much hope for a general determinacy-result, we will mostly concentrate on the subclass of live marked graphs, for which backtracking is never needed as already noted in [VW02].

Although marked graphs are a rather restricted subclass of Petri nets, the results for this subclass are non-trivial. Marked graphs are definitely of practical importance for asynchronous circuits and particularly prominent in benchmark examples studied in the respective community.

Here, we abstract from all signals or signal edges, and study the problem under which circumstances the following algorithm is determinate: given an unlabelled Petri net where some transitions are marked as 'to-be-contracted' apply secure transition contractions and redundant place deletions as long as possible.

We will show that for live marked graphs the algorithm is determinate, i.e. it produces a unique component (up to isomorphism). Parts of this result apply to general Petri nets, for which it is shown that secure transition contractions satisfy a weak diamond property. We give an easy-to-apply graph-theoretic characterisation of redundant places in marked graphs as so-called *shortcut places*; the result is a small generalisation of a result in [CCJS94] and the contribution is a much simpler proof. This result is

an important ingredient to prove the main result.

The chapter is organised as follows. In the next section, we characterise redundant places in marked graphs as shortcut places. The determinacy result is proven in Section 4.2.

4.1 Redundant Places in Marked Graphs

This section deals with redundant and implicit places in live marked graphs. The main result is that redundant and implicit places coincide in live marked graphs and furthermore they are either loop-only places or shortcut places.

We start with the definition of a marked graph.

Definition 4.1 (Marked graph)
A Petri net N is a *marked graph* (MG) (or *T-system*) if:

(1) $\forall p \in P.\ |{}^\bullet p| = 1 = |p^\bullet|$

(2) $\forall x, y \in P \cup T. W(x, y) \leq 1$ △

Lemma 4.2 (e.g. [DE95])
A live marked graph is reversible.

Furthermore, it is useful to distinguish between different types of redundant places as introduced in the following definition.

Definition 4.3
Let p be a place of a Petri net N.

(1) p is a *loop-only place* place if $\forall t \in T.\ M_N(p) \geq W(p, t) \leq W(t, p)$.

(2) If N is a marked graph, p is a *shortcut place* if a path $w = {}^\bullet p \ldots p^\bullet$ exists containing at least one place and satisfying $p \notin w$ and $M_N(p) \geq M_N(w \cap P)$. △

A special case of a shortcut places are *extended duplicates*. The place p is an extended duplicate if the corresponding path w only contains one place p'. In particular, if $M_N(p) = M_N(p')$, they are both redundant with respect to each other and one can delete either of them.

Proposition 4.4

(1) Loop-only places and shortcut places are redundant.

(2) If p is a redundant place of a Petri net N, it is a loop-only place exactly if some reference set Q is empty.

Proof. (1) For a loop-only place p set $Q = \emptyset,\ V(p) = 1$. For a shortcut place p with corresponding path w, set $Q = w \cap P,\ V(p) = 1$ and $V(q) = 1$ for $q \in Q$.
(2) The first direction follows from the proof of part (1). Therefore, assume the reference set Q to be empty. Since p is redundant, we get immediately $\forall t \in T$:

$$V(p)M_N(p) = d \qquad V(p)(W(t,p) - W(p,t)) \geq 0 \qquad V(p)W(p,t) \leq d$$

Dividing by $V(p)$ and combining the first and the last (in)equation yields $\forall t \in T$:

$$M_N(p) \geq W(p,t) \qquad W(t,p) \geq W(p,t)$$

This is equivalent to the definition of a loop-only place. □

Shortcut-places can also be defined for nets which are not marked graphs; it is enough that p itself and the places of the corresponding path w have the marked-graph property.
The first part of the following proposition was used in an alternative proof of Theorem 4.7, and we think that it is of independent interest. The second part will be applied below.

Proposition 4.5

(1) Let p be a redundant place of a live Petri net N with at least one home state. Then V is balanced.

(2) If, in an arbitrary net N, p is redundant under a marking $M \in [M_N\rangle$ with a balanced valuation, it is also redundant under M_N with the same valuation. In particular, if p is a shortcut place under M, it is also one under M_N.

Proof. (1) Let M_H be a home state of N. Using part 2 of Definition 3.2, it can be shown that $\forall t \in T.M_1[t\rangle M_2 \Rightarrow V(p)M_1(p) - \sum_{q\in Q} V(q)M_1(q) \leq V(p)M_2(p) - \sum_{q\in Q} V(q)M_2(q)$ $(*)$.
Let $M_H[v_1\rangle M[v_2\rangle M_H$, such that v_1 contains every transition $t \in T$ at least once. Such a sequence v_1 exists because N is live, v_2 exists because M_H is a home state.

Together with $(*)$ we get:

$$\begin{aligned} & V(p)M_H(p) - \sum_{q \in Q} V(q)M_H(q) \\ \leq \; & V(p)M(p) - \sum_{q \in Q} V(q)M(q) \\ \leq \; & V(p)M_H(p) - \sum_{q \in Q} V(q)M_H(q) \end{aligned}$$

Since N is live, there exists a marking $M_1 \in [M_H\rangle$ for each transition t with $M_1[t\rangle M_2$ and

$$\begin{aligned} & V(p)M_1(p) - \sum_{q \in Q} V(q)M_1(q) \\ = \; & V(p)M_2(p) - \sum_{q \in Q} V(q)M_2(q) \end{aligned}$$

Together with $M_2(s) = M_1(s) - W(s,t) + W(t,s)\ \forall s \in P$ this leads to:

$$\begin{aligned} & V(p)M_1(p) - \sum_{q \in Q} V(q)M_1(q) \\ = \; & V(p)(M_1(p) - W(p,t) + W(t,p)) \\ & - \sum_{q \in Q} V(q)(M_1(q) - W(q,t) + W(t,q)) \\ = \; & V(p)M_1(p) - \Big(\sum_{q \in Q} V(q)M_1(q) \Big) + V(p)(W(t,p) - W(p,t)) \\ & - \sum_{q \in Q} V(q)(W(t,q) - W(q,t)) \\ \Rightarrow \; & V(p)(W(t,p) - W(p,t)) - \sum_{q \in Q} V(q)(W(t,q) - W(q,t)) = 0 \end{aligned}$$

This implies directly that V is balanced.

(2) Items 2 and 3 of Definition 3.2 do not depend on the marking and item 1 follows directly from the valuation being balanced. If p is a shortcut place then the respective path induces a balanced valuation V (as observed in the proof of 4.4) and, since item (1) can be transferred from M to M_N, the marking of this path is at most the marking of p also under M_N. □

Before we prove the main theorem of this section, we note an easy lemma about liveness in marked graphs.

Lemma 4.6
Let c be a cycle of a marked graph N. For every reachable marking M, $M(c) = M_N(c)$. If N is live, c is initially marked.

Proof. Let $M[t\rangle M'$. We show that $M(c) = M'(c)$. For $t \in c$ this is trivially true, since all edge weights are 1 in marked graphs. Otherwise, t is not adjacent to any place of c, since N is a marked graph.

The second statement now follows easily; if c is not marked under M_N it is not marked under any reachable marking and therefore no transition of c can ever fire, a contradiction. □

Remark: Actually, marked graphs are live if and only if every cycle is initially marked, see e.g. [DE95]. This is a deeper result, which we do not need here. In fact, our proof of the next theorem has the advantage that it does not require profound knowledge about marked graphs, and we only proved the above lemma to demonstrate that our proof of Theorem 4.7 is indeed elementary.

Theorem 4.7
Let N be a live marked graph and $p \in P$. The following properties are equivalent:

(1) p is a redundant place

(2) p is an implicit place

(3) p is a loop-only place or a shortcut place

Proof. "1→2" even holds for arbitrary Petri nets – as we observed already –, and "3→1" follows from Proposition 4.4.

"2→3": Let p be an implicit place but not a loop-only one. We define $\{t_i\} = {}^\bullet p$ and $p^\bullet = \{t_o\}$. Obviously, $t_i \neq t_o$, because p not being a loop-only place would then imply $M_N(p) = 0$, which contradicts with liveness. Let N' be the net obtained from N by deleting t_i and all incident arcs. Observe that p is also implicit in N', since the set of firing sequence of N' coincides with the set of those firing sequences of N which do not contain t_i.

In N', starting from the initial marking, we fire transitions until a maximal set D of transitions is dead.[1] From this marking fire every transition not in D at least once;

[1] D does not necessarily contain all transitions, since we do not assume boundedness or connectedness.

we denote the marking reached by M. Observe that $(*)$ M can be reached in N by the same firing sequence.

Since t_o can fire at most $M_N(p)$ times in N', we must have $t_o \in D$. Furthermore, there exists a $p_1 \in {}^\bullet t_o,\ p_1 \neq p$ with $M(p_1) = 0$. If not, p would be the only place in ${}^\bullet t_o$ preventing the firing of t_o, hence would not be implicit in N'.

This implies ${}^\bullet p_1 \in D$; otherwise p_1 would have been marked when every transition not in D fired once. Now there is an unmarked place p_2 in ${}^\bullet({}^\bullet p_1)$ and so forth. This leads either to a cycle not containing any tokens, which is by $(*)$ a contradiction to N being live (cf. Lemma 4.6); or ends up in a place p' with an empty preset in N', hence $p' \in t_i{}^\bullet$ and so we have constructed an unmarked path from t_i to t_o not containing p. Therefore p is a shortcut place under M in N, cf. $(*)$, and we are done by Proposition 4.5.2. □

Remark: Javier Esparza pointed out (priv. comm.) that a weaker version of this theorem could be proved as follows. Assume p is a redundant place of a live and bounded marked graph N (or more generally: free-choice net N); then the removal of p results again in a live and bounded marked graph N', which is (roughly speaking) strongly connected by [Bes87]; in particular the transitions ${}^\bullet p$ and $p^\bullet$ are connected by a path in N'. This result is close to the above theorem, but it is in fact not useful for the purpose of the present paper, as it does not make any statements about the marking of such a path; the pure existence of a path is not sufficient for a place to be redundant.

A result very close to Theorem 4.7 can be found in [CCJS94]. The difference is that strong connectedness is assumed there – an assumption that we do not need. Furthermore, the proof in [CCJS94] makes heavy use of deep results about marked graphs, while our direct proof only needs elementary knowledge. [CCJS94] also considers some form of decomposition of marked graphs; we will discuss the relationship to our approach at the end of the next section.

To determine whether a place is structurally redundant, one can set up an instance of linear programming as discussed after Definition 3.2. Our theorem leads to a more efficient algorithm for live marked graphs as already noted in [CCJS94][2]: to check whether place p is structurally redundant, regard each place p_1 as an edge from ${}^\bullet p_1$ to $p_1^\bullet$, weighted according to the initial marking. Remove the edge corresponding to p and determine the shortest path from ${}^\bullet p$ to $p^\bullet$; if its length (i.e. its cumulated weight) is at most $M_N(p)$, p is redundant. With the basic version of Dijkstra's algorithm, this takes time $O(n^2)$, where n is the number of transitions.

[2] Actually, in [CCJS94], the addition of implicit places is considered.

For deciding whether a given place is redundant we note the following improvement. Dijkstra's algorithm determines all distances from ${}^\bullet p$ in increasing order. Hence, the algorithm can already be finished with a negative answer, if all transitions with a distance of no more than $M_N(p)$ have been found and if $p^\bullet$ is not among them, and indeed it is implemented in this way in DesiJ.

Furthermore, if $M_N(p) = 0$, one can delete all edges corresponding to initially marked places, and simply check for a path from ${}^\bullet p$ to $p^\bullet$ in the remainder e.g. with depth first search in time linear in the number of transitions and places.

Observe that the above considerations can be extended easily to the detection of shortcut places in nets which are not marked graphs.

4.2 Determinacy of Petri Net Operations

In this section, the determinacy of the decomposition method is studied. We only consider the operations secure transition contraction and redundant place deletion, since a marked graph cannot contain redundant transitions: the places adjacent to a loop-only transition t cannot be adjacent to other transitions and therefore this part of the net is not connected to the rest; since $l(t) = \lambda$ this is pointless and t and its adjacent places could be deleted in advance. Duplicate transitions cannot exist in a marked graph.

To show determinacy, we view these Petri net operations as a terminating reduction system, such that determinacy is related to confluence and local confluence. The notion 'reduction system' comes from the field of term rewriting. The following definition and lemma are taken from [BN98], where a detailed introduction can be found.

4.2.1 Decomposition as Reduction System

Definition 4.8
Let A be a nonempty set with $a, a', \ldots \in A$.

(1) A *reduction system* is a pair $(A, \rightarrow)$ with $\rightarrow \subseteq A \times A$. The relation $\rightarrow$ is called *reduction* or *reduction rule*; $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$, and $\rightarrow^=$ the reflexive closure.

(2) A reduction $\rightarrow$

  (a) is *terminating* if there exists no infinite chain $a_0 \rightarrow a_1 \rightarrow a_2 \ldots$

  (b) is *confluent* if $a \rightarrow^* a_1, a \rightarrow^* a_2$ implies $a_1 \rightarrow^* a', a_2 \rightarrow^* a'$ for some a'

(c) is *locally confluent* if $a \to a_1, a \to a_2$ implies $a_1 \to^* a', a_2 \to^* a'$ for some a'

(d) has the *diamond property* if $a \to a_1, a \to a_2$ implies $a_1 \to a', a_2 \to a'$ for some a'

(3) An element a is

(a) in *normal form* if $\neg\exists a'.\ a \to a'$

(b) a *normal form of* a' if $a' \to^* a$ and a is in normal form. △

Lemma 4.9
(1) A terminating reduction is confluent if and only if it is locally confluent.

(2) For a terminating and confluent reduction, every element has a unique normal form.

Next, we model the behaviour of the decomposition algorithm as a reduction system. Since the components are independently reduced, we can restrict ourselves to the processing of one net, where repeatedly structurally redundant places are removed and transitions from a distinguished set are securely contracted. Also, we concentrate on live marked graphs, although the reduction rules below can actually be defined for general nets. Theorem 4.13 gives a result for general Petri nets.

Definition 4.10
Let $MGR := \{(N, \Lambda) | N \text{ is a live marked graph}, \Lambda \subseteq T\}$, where Λ denotes the set of transitions to be contracted. We define the following reduction rules on MGR:

(1) $(N, \Lambda) \to_{stc} (\overline{N}^t, \Lambda - \{t\})$, where secure contraction of $t \in \Lambda$ is applied.

(2) $(N, \Lambda) \to_{rpd} (N', \Lambda)$ if N' is obtained from N by deleting a redundant place.

(3) $\to_{red} = \to_{stc} \cup \to_{rpd}$ △

These reductions are well-defined as the following proposition shows.

Proposition 4.11
Applying $\to_{red}$ preserves the marked graph properties (Definition 4.1) as well as liveness.

Proof. Deleting a redundant place does not change the firing sequences of the net and therefore liveness is preserved. Since the other places are not affected, the marked graph properties remain valid.

Let $p' = (p_1, p_2)$ be a place resulting from a secure contraction of a transition t. Since p_1 has exactly one transition t' in its preset and p_2 none except t, we get ${}^\bullet p' = \{t'\}$, and analogously for the postset. Preservation of liveness follows from Proposition 3.13. □

Furthermore, $\rightarrow_{red}$ is a terminating reduction, as noted in [VW02] for general Petri nets: only finite nets are considered, $\rightarrow_{stc}$ reduces the number of transitions, this stays the same under $\rightarrow_{rpd}$, and $\rightarrow_{rpd}$ reduces the number of places.

Each normal form of $(N, \Lambda) \in MGR$ is a possible result of the decomposition algorithm. Thus, by Lemma 4.9, it suffices to show that $\rightarrow_{red}$ is locally confluent in order to prove decomposition to be determinate; recall that we regard isomorphic nets as equal.

To show the local confluence of $\rightarrow_{red}$, we need to show the local confluence for every of the three combinations of $\rightarrow_{stc}$ and $\rightarrow_{rpd}$; this is done in the next subsection.

4.2.2 Reduction is Locally Confluent

Local Confluence of $\rightarrow_{stc}$

Now, we will show the local confluence for secure transition contractions in live marked graphs. Before that, a result for arbitrary transition contractions in arbitrary Petri nets similar to local confluence is given, namely Theorem 4.13, which is something like a weak diamond property.

Lemma 4.12
Let N be a Petri net, $N' = \overline{N}^{t_1,t_2}$ and $p'_1, p'_2 \in P'$. If $\overline{N}^{t_2,t_1}$ is defined as well, $\Phi_N^{N'}(p'_1) = \Phi_N^{N'}(p'_2)$ implies $p'_1 = p'_2$.

Proof. This proof works with the Tables 4.1 and 4.2. In the first one, all possibilities for the structure of a place after two transition contractions are listed. In the latter one, these 6 cases are instantiated resulting in 30 combinations of places from the original net.

As indicated in Table 4.2, many of the combinations are actually not possible for simple reasons. For example, if (p_1, p_1) is part of the place from P' then $p_1 \in {}^\bullet t_1$ and $p_1 \in t_1{}^\bullet$, a contradiction since the contraction of a transition with a loop is not defined. As another example, case 23 drops out, because p_1 belongs to the preset of t_1 due the occurrence of (p_1, p_2), and on the other hand p_1 is element of its postset, due to the occurrence of (p_2, p_1). Therefore, p_1 forms a loop with the first contracted transition. With the same argumentation cases 24 and 28 are impossible.

The remaining impossible cases 25, 27, and 30 are considered in more detail.

Group	Structure
1	$((p, \star), \star)$
2	$((p, p), \star)$
3	$((p, \star), (p, \star))$
4	$((p, \star), (p, p))$
5	$((p, p), (p, \star))$
6	$((p, p), (p, p))$

Table 4.1: For the proof of Lemma 4.12. Structures of possible places after two transition contractions. This table is obtained from all syntactically possible places by omitting cases which contains a leading $\star$, e.g. $(\star, (p, \star))$. Here, p is only a placeholder for an arbitrary place; in Table 4.2 all possible instantiations are considered.

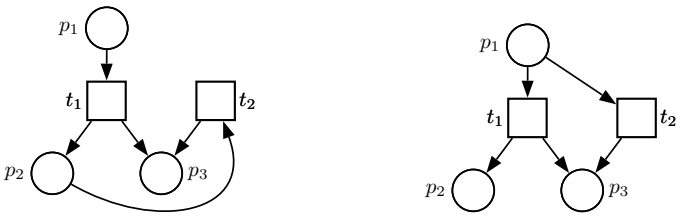


Figure 4.1: Case 25 - $p' = ((p_1, p_2), (p_1, p_3))$. p_1 has to be an element of ${}^\bullet t_1$, p_2 and p_3 have to be elements of $t_1{}^\bullet$. Then there are 4 cases: (1) $p_1 \in {}^\bullet t_2, p_3 \in t_2{}^\bullet$: loop after contracting t_1, (*left*) (2) $p_1 \in {}^\bullet t_2, p_1 \in t_2{}^\bullet$: initial loop $p_1 - t_1$ (3) $p_2 \in {}^\bullet t_2, p_1 \in t_2{}^\bullet$: loop after contracting t_1 (4) $p_2 \in {}^\bullet t_2, p_3 \in t_2{}^\bullet$: weight 2 after contracting t_2, (*right*).

Case 25 leads either to a loop after contracting t_1 or to an arc with weight 2 after contracting t_2, see Figure 4.1. Case 27 is very similar to the previous one, only the pre- and postsets of t_1 are exchanged.

At last case 30 remains which is more complicated but nevertheless turns out to be impossible, see Figure 4.2.

In summary, it suffices to consider the cases 1, 3, 5, 10 and 15 (also shown in Table 4.3, middle column, the last column is used later). We distinguish three cases for $\Phi_N^{N'}(p_1')$.

Now we consider the resulting cases for $\Phi_N^{N'}(p_1')$ and $\Phi_N^{N'}(p_2')$:

(1) $\Phi_N^{N'}(p_1') = \{p_1\} = \Phi_N^{N'}(p_2')$.

This is only possible if both p_1' and p_2' are in the form of case 1 which implies $p_1' = p_2'$.

No.	Group	Places	Example	Pos.	If not, why?
▲ 1	1	1	$((p_1, \star), \star)$	•	
2	2	1	$((p_1, p_1), \star)$	-	initial loop $p_1 - t_1$
▲ 3	2	2	$((p_1, p_2), \star)$	•	
4	3	1	$((p_1, \star), (p_1, \star))$	-	initial loop $p_1 - t_2$
▲ 5	3	2	$((p_1, \star), (p_2, \star))$	•	
6	4	1	$((p_1, \star), (p_1, p_1))$	-	type error
7	4	2	$((p_1, \star), (p_1, p_2))$	-	type error
8	4	2	$((p_1, \star), (p_2, p_1))$	-	type error
9	4	2	$((p_2, \star), (p_1, p_1))$	-	initial loop $p_1 - t_1$
▲ 10	4	3	$((p_1, \star), (p_2, p_3))$	•	
11	5	1	$((p_1, p_1), (p_1, \star))$	-	type error
12	5	2	$((p_1, p_1), (p_2, \star))$	-	initial loop $p_1 - t_1$
13	5	2	$((p_1, p_2), (p_1, \star))$	-	type error
14	5	2	$((p_2, p_1), (p_1, \star))$	-	type error
▲ 15	5	3	$((p_1, p_2), (p_3, \star))$	•	
16	6	1	$((p_1, p_1), (p_1, p_1))$	-	initial loop $p_1 - t_1$
17	6	2	$((p_1, p_1), (p_1, p_2))$	-	initial loop $p_1 - t_1$
18	6	2	$((p_1, p_1), (p_2, p_1))$	-	initial loop $p_1 - t_1$
19	6	2	$((p_1, p_2), (p_1, p_1))$	-	initial loop $p_1 - t_1$
20	6	2	$((p_2, p_1), (p_1, p_1))$	-	initial loop $p_1 - t_1$
21	6	2	$((p_1, p_1), (p_2, p_2))$	-	initial loop $p_1 - t_1$ and $p_2 - t_1$
22	6	2	$((p_1, p_2), (p_1, p_2))$	-	loop after contracting t_1
23	6	2	$((p_2, p_1), (p_1, p_2))$	-	initial loop $p_1 - t_1$
24	6	3	$((p_1, p_2), (p_3, p_1))$	-	initial loop $p_1 - t_1$
▲ 25	6	3	$((p_1, p_2), (p_1, p_3))$	-	loop after contracting t_1 or weight 2 after contracting t_2
26	6	3	$((p_1, p_1), (p_2, p_3))$	-	initial loop $p_1 - t_1$
▲ 27	6	3	$((p_2, p_1), (p_3, p_1))$	-	loop after contracting t_1 or weight 2 after contracting t_2
28	6	3	$((p_2, p_1), (p_1, p_3))$	-	initial loop $p_1 - t_1$
29	6	3	$((p_2, p_3), (p_1, p_1))$	-	initial loop $p_1 - t_1$
▲ 30	6	4	$((p_1, p_2), (p_3, p_4))$	-	loop or weight 2 after contracting t_2

Table 4.2: For the proof of Lemma 4.12. All possible places (up to isomorphism) after contraction of t_1 and t_2. They are obtained from Table 4.1 by instantiating p. The places p_i are pairwise different. The rows with a 'type error' are impossible, since a place is treated as being and at the same time as not being adjacent to a contracted transition; 'initial loop' indicates a loop at one of the transitions initially. Rows with a ▲ are explained in the text.

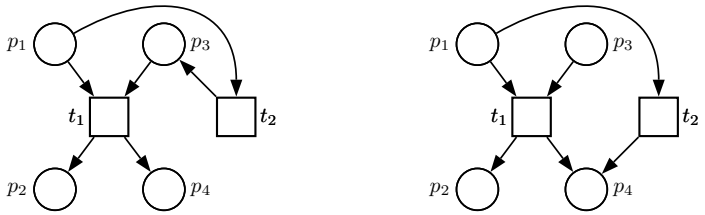


Figure 4.2: Case 30 - $((p_1, p_2), (p_3, p_4))$. p_1 and p_3 have to be in the preset of the first transition to be contracted (t_1), p_2 and p_4 in the postset. For the connection to t_2 there are several possibilities; all of them satisfy that p_1 or p_2 (or both) are in the preset and p_3 or p_4 (or both) are in the postset, which leads to 9 sub-cases. Exemplarily two of them are considered. *left*: leads to an arc with weight 2 when t_2 is contracted first. *right*: leads to a loop. The other cases are similar to these ones or contain them.

(2) $\Phi_N^{N'}(p_1') = \{p_1, p_2\} = \Phi_N^{N'}(p_2')$.

Therefore, $p_1', p_2' \in \{((p_1, p_2), \star), ((p_2, p_1), \star), ((p_1, \star), (p_2, \star)), ((p_2, \star), (p_1, \star))\}$. If a fixed p_1' from this set occurs in the net $\overline{N}^{t_1,t_2}$ it is not possible that a different element from this set occurs, too; for example: if $p_1' = ((p_1, p_2), \star)$ there is no place $p_1'' = ((p_2, p_1), \star)$, since the existence of p_1' implies that p_1 is an element of ${}^\bullet t_1$ but the existence of p_1'' implies p_1 is an element of $t_1{}^\bullet$; a contradiction, since the contraction was possible. With similar argumentations one can exclude the other combinations. Hence, $\Phi_N^{N'}(p_1') = \Phi_N^{N'}(p_2')$ implies $p_1' = p_2'$ for this case.

(3) $\Phi_N^{N'}(p_1') = \{p_1, p_2, p_3\} = \Phi_N^{N'}(p_2')$.

Analogous to the second, case we obtain twelve possible structures for p_1', p_2' resp. which all exclude each other as places of P', see the following table.

1	$((p_1, p_2), (p_3, \star))$	7	$((p_1, \star), (p_2, p_3))$
2	$((p_1, p_3), (p_2, \star))$	8	$((p_1, \star), (p_3, p_2))$
3	$((p_2, p_1), (p_3, \star))$	9	$((p_2, \star), (p_1, p_3))$
4	$((p_2, p_3), (p_1, \star))$	10	$((p_2, \star), (p_3, p_1))$
5	$((p_3, p_1), (p_2, \star))$	11	$((p_3, \star), (p_1, p_2))$
6	$((p_3, p_2), (p_1, \star))$	12	$((p_3, \star), (p_2, p_1))$

Without loss of generality, assume $p_1' = ((p_1, p_2), (p_3, \star))$ (case 1) or $p_1' = ((p_3, \star), (p_1, p_2))$ (case 11). $(p_3, \star)$ implies that p_3 is not adjacent to t_1, and therefore the existence of such a place excludes the existence of places 2,4-10.

The remaining cases 3 and 12 can be excluded, since (p_2, p_1) implies $p_1 \in t_1{}^\bullet$ whereas p_1' implies $p_1 \in {}^\bullet t_1$; in this case p_1 would be a loop place which is a contradiction. Case 1 cannot coexist with case 11, since the latter implies $(p_1, p_2) \in t_2{}^\bullet$ whereas the former case implies $(p_1, p_2) \in {}^\bullet t_2$ after contracting t_1, also a contradiction. □

No.	$\overline{N}^{t_1,t_2}$	$\overline{N}^{t_2,t_1}$
1	$((p_1, \star), \star)$	$((p_1, \star), \star)$
2	$((p_1, p_2), \star)$	$((p_1, \star), (p_2, \star))$
3	$((p_1, \star), (p_2, \star))$	$((p_1, p_2), \star)$
4	$((p_1, \star), (p_2, p_3))$	$((p_1, p_2), (p_3, \star))$ / $((p_2, \star), (p_1, p_3))$
5	$((p_1, p_2), (p_3, \star))$	$((p_1, \star), (p_2, p_3))$ / $((p_1, p_3), (p_2, \star))$

Table 4.3: Possible places after two transition contractions. In the middle column one can find the places from Table 4.2 which turned out to be possible according to Definition 3.5. In each case, there exists a place in $\overline{N}^{t_2,t_1}$ which uses the same places from N as the one in the middle column. This place is shown in the last column; for line 4 and 5 there are two possibilities, but only one of them exists.

Theorem 4.13
Let N be a Petri net and $t_1, t_2 \in T$. If both $\overline{N}^{t_1,t_2}$ and $\overline{N}^{t_2,t_1}$ are defined they are isomorphic (even if the contractions are not secure).

Proof. For this proof Table 4.3 is used. The last column shows the place of $N_2 = \overline{N}^{t_2,t_1}$, which uses the same places from N as the place from $N_1 = \overline{N}^{t_1,t_2}$ in the middle column. If there are two possibilities, only one of them exists. For lines 1-3, it is quite clear that these places exist in N_2, for line 4 see Figure 4.3: since the place $((p_1, \star), (p_2, p_3))$ exists in $\overline{N}^{t_1,t_2}$, N must contain the leftmost net fragment; observe that exactly one of the dotted arcs exists but not both (in this case contracting t_1 would generate an arc with weight 2). Depending on which arc exist in $\overline{N}^{t_2,t_1}$, exactly one of the places in the last column exists. Line 5 is analogous.

We define a relation $f \subseteq P_1 \times P_2 \cup T_1 \times T_2$ by $f|_{T_1 \times T_2} = Id$ and $(p_1', p_2') \in f \Leftrightarrow \Phi_N^{N_1}(p_1') = \Phi_N^{N_2}(p_2')$. We will show that f is an isomorphism.

a) f is a partial function: $(p_1', p_2'), (p_1', p_2'') \in f \Rightarrow \Phi_N^{N_2}(p_2') = \Phi_N^{N_2}(p_2'')$. Lemma 4.12 implies $p_2' = p_2''$.

b) f is total (surjective): After two contractions each place $p_1' \in P_1$ has a structure shown in Table 4.3, middle column, and $\Phi_N^{N_1}(p_1') = \Phi_N^{N_2}(p_2')$ holds for the corresponding place p_2' in the last column. Analogous for surjective.

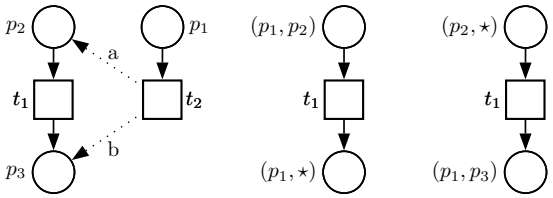


Figure 4.3: For line 4 from Table 4.3. Since the place $((p_1, \star), (p_2, p_3))$ exists in $\overline{N}^{t_1,t_2}$, N must contain the leftmost net fragment; observe that exactly one of the dotted arcs exists but not both (in this case contracting t_1 would generate an arc with weight 2). If arc a, b resp. exists, contracting t_2 first results in the middle, rightmost resp. fragment; the next contraction results in $((p_1, p_2), (p_3, \star))$, $((p_2, \star), (p_1, p_3))$ resp. as it is written in the last column.

c) f is injective: $f(p_1') = f(p_1'') \Rightarrow \Phi_N^{N_1}(p_1') = \Phi_N^{N_1}(p_1'')$. From Lemma 4.12 follows $p_1' = p_1''$.
d) f preserves the structure, i.e. $W_1(p_1', t) = W_2(f(p_1'), f(t))$ and $W_1(t, p_1') = W_2(f(t), f(p_1'))$ $\forall p_1' \in P_1, t \in T_1$. This follows from the definition of transition contraction. Since the weight of an arc incident to a composite place is the sum of the related weights of the component places, we derive that

$$W_1(p_1', t_1) = \sum_{p \in \Phi_N^{N_1}(p_1')} W(p, t_1) = \sum_{p \in \Phi_N^{N_2}(f(p_1'))} W(p, t_1) = W_2(f(p_1'), f(t_1)) \quad .$$

Observe that for every place p_1' of N_1 shown in Table 4.3, $\Phi_N^{N_1}(p_1')$ is a *set*. Analogous for the second case. □

The proof for the following lemma uses Theorem 4.13; if this is not applicable, we show that – since $N \in MGR$ – in N_1 and N_2 loop-only places can be deleted such that the contraction of t_2 and t_1 resp. is applicable afterwards. After the contraction, extended duplicates can be deleted such that the results are isomorphic.

Lemma 4.14
For $(N, \Lambda) \in MGR$, let $(N, \Lambda) \rightarrow_{stc} (N_1, \Lambda_1)$ and $(N, \Lambda) \rightarrow_{stc} (N_2, \Lambda_2)$. Then, there exists $(N', \Lambda') \in MGR$ with $(N_1, \Lambda_1) \rightarrow_{red}^* (N', \Lambda')$ and $(N_2, \Lambda_2) \rightarrow_{red}^* (N', \Lambda')$.

Proof. Let the contractions concern transition t_1 and t_2. If both $\overline{N}^{t_1,t_2}$ and $\overline{N}^{t_2,t_1}$ are defined, Theorem 4.13 implies that the results are isomorphic. In this case even the diamond property is fulfilled.

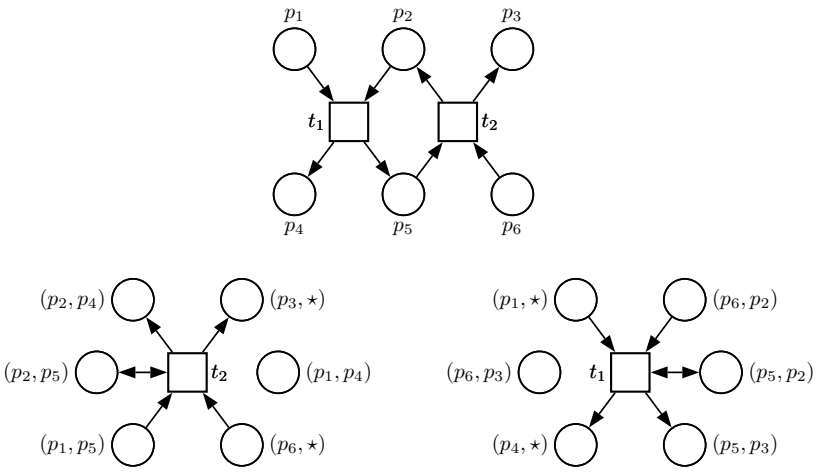


Figure 4.4: *Top*: Scheme of a net fragment where contraction generates a loop. *Left*: After t_1-contraction *Right*: After t_2-contraction.

Therefore, assume that w.l.o.g. $\overline{N}^{t_1,t_2}$ is not defined. Since $N_1 = \overline{N}^{t_1}$ is defined by hypothesis, the contraction of t_2 is not possible in N_1, although it is possible in N. Since N_1 is a marked graph — in particular no arc weight becomes greater than 1 —, the contraction of t_1 in N must have generated a loop place adjacent to t_2, because t_1 and t_2 form a cycle with two places in N. Since N is a live marked graph, this cycle contains at least one token making the loop place redundant.

This situation is schematically shown in Figure 4.4(top): each place represents a set of places connected to t_1 and t_2 in the same way, e.g. places of type p_1 are in the preset of t_1 and not adjacent to t_2. Figure 4.4(left) and (right) depict the results of contracting t_1 and t_2 resp. in the same way, e.g. places of type (p_2, p_4) are pairs (p, p') with p of type p_2 and p' of type p_4.

Places of type (p_2, p_5) and (p_5, p_2) are loop-only places, which can be removed as noted above; afterwards, the other transition contraction becomes possible. These contractions give places of types

$$((p_1, p_4), *),\ ((p_1, p_5), (p_3, \star)),\ ((p_1, p_5), (p_2, p_4)),$$

$$((p_6, *), (p_2, p_4)),\ ((p_6, *), (p_3, *))$$

in the first case and

$$((p_1, *), (p_4, *)),\ ((p_1, *), (p_5, p_3)),\ ((p_6, p_2), (p_5, p_3)),$$

$$((p_6, p_2), (p_4, *)),\ ((p_6, p_3), *)$$

in the second. We will argue that the resulting nets are isomorphic after removal of some redundant places.

As noted in the proof of Theorem 4.13, the connections of these places to the remaining transitions are determined by their at most four components, and analogously for the initial marking. In particular, places of type $((p_1, p_5), (p_2, p_4))$ are connected in the same way as places of type $((p_1, p_4), *)$ in the first case – since t_1 and t_2 are not present anymore – and they carry even more tokens, since at least one of a p_2-type and a p_5-type place is marked in N. Therefore, places of type $((p_1, p_5), (p_2, p_4))$ are extended duplicates, and so are places of type $(p_6, p_2), (p_5, p_3))$; we remove them in the two nets.

For the other types, we find a matching between $((p_1, p_4), *)$ and $((p_1, *), (p_4, *))$, and also between $((p_1, p_5), (p_3, *))$ and $((p_1, *), (p_5, p_3))$ etc., which matches each place of type $((p_1, p_4), *)$ to the place of type $((p_1, *), (p_4, *))$ with the same component-places etc. By the above, this gives an isomorphism between the remaining nets when the above extended duplicates are removed. □

Local Confluence of $\rightarrow_{rpd}$

Now, we will proceed to the next part of the local confluence proof. Although the local confluence of redundant place deletion might seem rather obvious, in fact some effort is already needed to prove it at least for marked graphs.

Let p_1, p_2 be redundant places of $N \in MGR$ with $p_1 \neq p_2$. If one of them, lets say p_1, is a loop-only place, then $p_2 \notin Q_1 = \emptyset$ and $p_1 \notin Q_2$, because p_1 is only adjacent to one transition. This case obviously fulfils the diamond property, since the deletion of one of the redundant places does neither affect the other one nor its reference set.

Due to Theorem 4.7 we can now assume that p_1 and p_2 are shortcut places and the reference sets consist of the places of the corresponding paths.

We will distinguish three cases: 1) $p_1 \notin Q_2, p_2 \notin Q_1$, 2) $p_1 \notin Q_2, p_2 \in Q_1$ (w.l.o.g.) and 3) $p_1 \in Q_2, p_2 \in Q_1$.

The first case is treated as above. For the second case take a look at Figure 4.5. Since p_1 is not a loop-only place, p_2 lies on a Q_1-path $w_1 = {}^\bullet p_1 q_1^1 \dots q_1^m p_1{}^\bullet$. Since p_2 is not a loop-only place either, a Q_2-path $w_2 = {}^\bullet p_2 q_2^1 \dots q_2^n p_2{}^\bullet$ exists. This implies that there is a path w connecting ${}^\bullet p_1$ and $p_1{}^\bullet$ and using only places from $q_1^1 \dots q_1^m$ excluding p_2 and from $q_2^1 \dots q_2^n$. $M_N(p_1) \geq \sum_{i=1}^m M_N(q_1^i)$ and $M_N(p_2) \geq \sum_{i=1}^n M_N(q_2^i)$ (Definition 3.2(1)) directly imply that $M_N(p_1) \geq \sum_{i=1}^m M_N(q_1^i) - M_N(p_2) + \sum_{i=1}^n M_N(q_2^i)$; hence, w also shows that p_1 is redundant; the corresponding reference set does not contain p_2 and we are done by case (1).

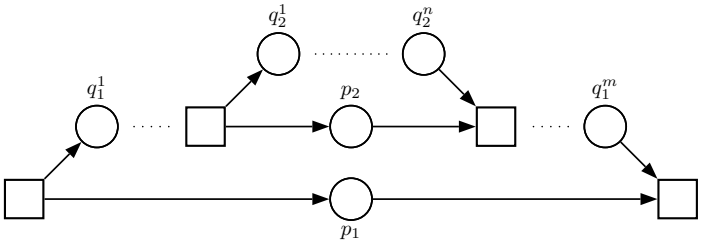


Figure 4.5: Two redundant places p_1, p_2 with $p_1 \notin Q_2, p_2 \in Q_1$

The last case $p_1 \in Q_2, p_2 \in Q_1$ is impossible, because it implies

$$M_N(p_1) \geq \sum_{q \in Q_1 \setminus \{p_2\}} M_N(q) + M_N(p_2) \qquad M_N(p_2) \geq \sum_{q \in Q_2 \setminus \{p_1\}} M_N(q) + M_N(p_1)$$

From this we get immediately:

$$M_N(p_1) = M_N(p_2) \quad \text{and} \quad \sum_{q \in Q_1 \setminus \{p_2\}} M_N(q) \; = \sum_{q \in Q_2 \setminus \{p_1\}} M_N(q) = 0 \quad (*)$$

Since $p_1 \in Q_2$, there are Q_2-paths ${}^\bullet p_2 \dots {}^\bullet p_1$ and $p_1{}^\bullet \dots p_2{}^\bullet$ not using p_1, and analogously there are Q_1-paths ${}^\bullet p_1 \dots {}^\bullet p_2$ and $p_2{}^\bullet \dots p_1{}^\bullet$ not using p_2. Therefore, either a cycle c using only places from $(Q_1 \cup Q_2) \setminus \{p_1, p_2\}$ exists which contradicts N being live by Lemma 4.6, since $(*)$ implies $M_N(c) = 0$; or $(Q_1 \cup Q_2) \setminus \{p_1, p_2\} = \emptyset$. In the latter case, p_1 and p_2 are extended duplicates of each other with the same initial marking; thus, removing either of them gives the same net up to isomorphism.

Altogether the following lemma holds.

Lemma 4.15
Let $(N, \Lambda) \to_{rpd} (N_1, \Lambda_1)$ *and* $(N, \Lambda) \to_{rpd} (N_2, \Lambda_2)$ *for some* $(N, \Lambda) \in MGR$. *Then an* $(N', \Lambda') \in MGR$ *exists with* $(N_1, \Lambda_1) \to_{rpd}^{=} (N', \Lambda')$ *and* $(N_2, \Lambda_2) \to_{rpd}^{=} (N', \Lambda')$.

Observe that two steps of $\to_{rpd}$ fulfil the diamond property or lead to isomorphic results; in particular we have not used $\to_{stc}$.

Local confluence of $\rightarrow_{stc}$ and $\rightarrow_{rpd}$

Lemma 4.16
Let $(N,\Lambda) \rightarrow_{rpd} (N_1,\Lambda_1)$ and $(N,\Lambda) \rightarrow_{stc} (N_2,\Lambda_2)$ for some $(N,\Lambda) \in MGR$. Then, there exists an $(N',\Lambda') \in MGR$ with $(N_1,\Lambda_1) \rightarrow^*_{red} (N',\Lambda')$ and $(N_2,\Lambda_2) \rightarrow^*_{red} (N',\Lambda')$.

Proof. Let p be the redundant place and t the transition to be contracted. In live marked graphs p is either a loop-only place or a shortcut place.

In the first case, t and p are not adjacent because the contraction of t is possible for (N,Λ), i.e. p forms a loop with another transition and the operations can be performed independently.

If p is a shortcut place, there are the following possibilities: 1) t is neither adjacent to p nor part of the path making p redundant; then both operations are independent of each other again. 2) t is part of the path but not adjacent to p. The contraction of t shortens the path but does not interrupt it, and also the sum of the markings remains unchanged; hence, the two operations are independent. 3) t is adjacent to the path *and* p – leading to two sub-cases, one of them shown in Figure 4.6(left). In the other one, analogously the path starts from t and $p \in t^\bullet$.

We will only consider the first case depicted on the left hand side, with the results of contraction and deletion shown in the middle, right hand side resp.. Each place (p_s, p_{x_i}) in the middle is a shortcut place of $\{(p_1,*), \ldots, (p_{n-1},*), (p_n, p_{x_i})\}$ because they give a path and the initially marking of this path as well as $M_N(p_s)$ are increased by the same value $M_N(x_i)$. Therefore, these shortcut places can be deleted yielding a net which also results from the net on the right hand side when contracting t. □

Altogether, our results can be collected in the central theorem of this chapter.

Theorem 4.17
The reduction $\rightarrow_{red}$ is confluent and terminating for live marked graphs.

Corollary 4.18
The STG-decomposition algorithm is determinate for live marked graphs.

In [CCJS94] a decomposition of strongly connected live marked graphs into two components is considered. In this approach, the nets are unlabelled, while our STG decomposition is directed by the labelling with signal transitions. Therefore, the decomposition of [CCJS94] is not applicable in our setting.

What is interesting is that in the decomposition of [CCJS94] a whole subnet is removed and this could be used in our setting to remove several internal transitions

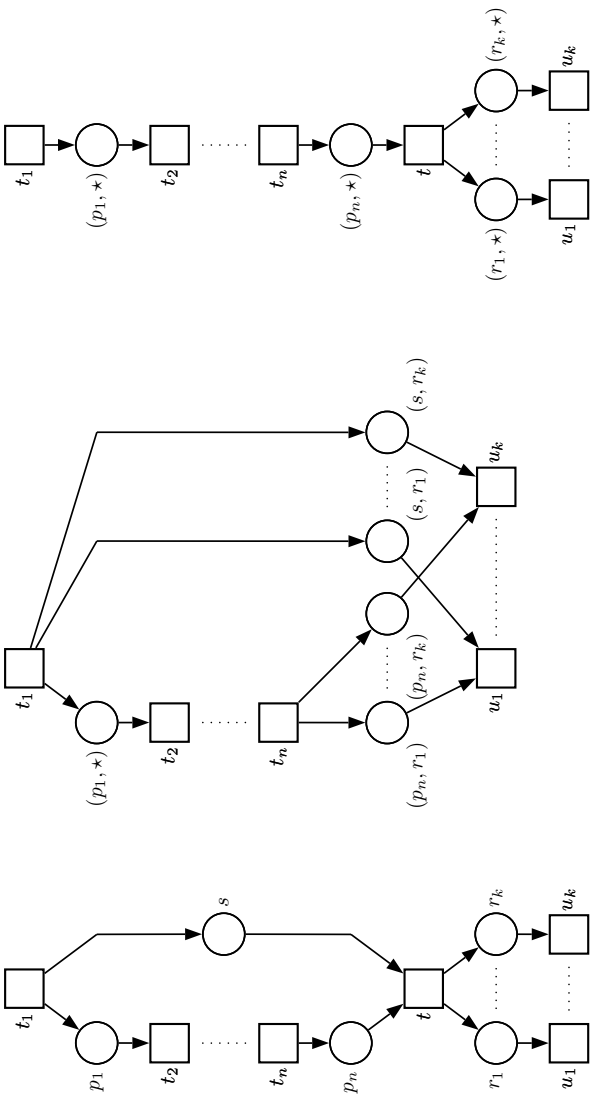


Figure 4.6: Confluence of shortcut place deletion and transition contraction. *left:* p_s is a shortcut place of $\{p_1, \ldots, p_n\}$ and t is the transition to be contracted. *middle:* after contraction of t *right:* after deleting s.

together. A result of [CCJS94] implies that this removal preserves the language, but this does not immediately imply that subnet removal can be used to determine correct STG decompositions in the sense of [VW02, VK06]. In fact, the correctness criterion of [VW02, VK06] is of bisimulation type, but does not imply language equivalence. Furthermore, redundant place deletion and secure transition contractions always lead to a correct decomposition, while subnet removal presupposes liveness and strong connectedness. In fact, liveness only is a precondition for determinacy of STG decomposition but not for its correctness.

Nevertheless, subnet removal might be closely related to redundant place deletion and secure transition contractions. If one could show some sort of coincidence this might lead to an alternative proof of our determinacy result. Such a result would not imply that subnet removal is more efficient; the latter involves solving an all-pairs shortest paths problem, which takes time of $O(n^3)$ where n is the number of removed internal transitions *plus* the number of 'neighbouring' non-internal transitions.

Chapter 5

Internal Signals and STG-Bisimulation

Complete State Coding (CSC) is an important property of STGs and must be achieved before an asynchronous circuit can be synthesised. While some decomposition methods [CC03, YOM04] have to assume that the original STG satisfies CSC, our decomposition algorithm is more general since it does not presuppose this; on the other hand, the methods in [CC03, YOM04] construct components with CSC, while our components might not have CSC.

For each such component, one can solve CSC and synthesise a separate circuit; compared to solving CSC for the original STG (with its potentially huge reachability graph) and synthesising one circuit, this can be much faster, see experimental results in [VK06, Chapter 5].

One would expect that the components generated by our decomposition algorithm are still correct when they have been modified to achieve CSC, and in fact it would also be very interesting in what sense CSC-solving with Petrify is correct – independently of the issue of decomposition; it seems that no correctness for this has been proven so far. For such correctness results, one needs a correctness definition that takes internal signals into account.

The purpose of this chapter is to enhance the correctness notion of Definition 3.4 appropriately, to study its properties and give applications in the area of decomposition and CSC-solving.

As the main property of the new correctness notion, we show that it is preserved when decomposition is performed *hierarchically*. This correctness of top-down decomposi-

tion is of interest in itself, but it also implies that the implementation relation arising from our correctness notion is a preorder, and it can in particular be used to improve the efficiency of our decomposition algorithm; an application of this and other methods can be found in Chapter 6. Then we prove that CSC-solving for speed-independent circuits as performed by PETRIFY is correct in our sense. With our result on the correctness of top-down decomposition, we then conclude that speed-independent CSC-solving can indeed be combined with the decomposition algorithm described in Section 3.3. As another contribution, we prove that the decomposition method in [CC03] is correct in the sense of our enhanced correctness definition; in [CC03] itself, no correctness proof is given. Finally, we compare our implementation relation with existing concepts.

The chapter is organised as follows. In the next section, the new correctness definition with internal signals is introduced. In Section 5.2, we prove top-down decomposition correct in terms of our enhanced correctness definition; the succeeding section studies correctness of speed-independent CSC solving on its own and in combination with decomposition. Section 5.4 shows the correctness for the approach of [CC03], which is followed by the comparison of our implementation relation with the one of Dill, Carmona and Cortadella respectively.

5.1 Extended Correctness Definition

Before we come to our new correctness definition for decomposition, we introduce an important notion, which is related to the speed-independent model. As mentioned before, PETRIFY can modify an STG such that CSC is satisfied. If one is interested in speed-independent circuits, as we are here, one can require that PETRIFY preserves the following important property.

Definition 5.1 (Input Properness)
An *STG* is *input proper* if no input signal becomes enabled by the occurrence of an internal signal, i.e. $M_1[s^{\pm}\rangle\rangle M_2$ with M_1 a reachable marking, $\neg M_1[a\rangle\rangle$ and $M_2[a\rangle\rangle$, $a \in In$, implies $s \notin Int$. △

Recall that an STG also specifies which inputs the environment may perform; if the environment performs an input that is not enabled in the current marking of the STG, then such an unexpected input may lead to a malfunction of the circuit as described previously. To meet this specification, the environment must 'know' whether an input is expected or not. Therefore, speed-independent asynchronous circuits have to be input proper: as an example where input-properness is violated, consider the case when the environment will produce an input signal after the circuit produced a

certain output, but the circuit must produce some internal signal before it is ready to receive this input. Since the SI-model allows that the production of the internal signal is delayed, the input from the environment might arrive too early.

Actually, the implementation of non-input-proper STGs is still possible, but one has to make *timing assumptions* about the relative order of signal transitions, e.g. one might assume that an input is slower than an internal signal if both are triggered by the same output.

Now, we give our improved correctness definition, which considers internal signals; afterwards, we will explain its specific properties and why they are sound. In addition to internal signals of the components, we allow for the components to communicate with each other internally.[1] Such signals have to be outputs of a component, but internal signals from the perspective of the specification. Thus, we allow such signals to be hidden globally on the level of the parallel composition using the signal set H.

Definition 5.2 (Correct Decomposition with Internal Signals)
A collection of deterministic components $(C_i)_{i\in I}$ is a *correct decomposition* of (or simply *correct w.r.t.*) a deterministic STG N – also called *specification – when hiding* H, if $C = (||_{i\in I} C_i)/H$ is defined, $In_C \subseteq In_N$, $Out_C \subseteq Out_N$ and there is an *STG-bisimulation* $\mathcal{B}$ between the markings of N and those of C with the following properties:

1. $(M_N, M_C) \in \mathcal{B}$
2. For all $(M, M') \in \mathcal{B}$, we have:
   - (N1) If $a \in In_N$ and $M[a^{\pm}\rangle\rangle M_1$, then either $a \in In_C$, $M'[a^{\pm}\rangle\rangle M_1'$ and $(M_1, M_1') \in \mathcal{B}$ for some M_1' or $a \notin In_C$ and $(M_1, M') \in \mathcal{B}$.
   - (N2) If $x \in Out_N$ and $M[x^{\pm}\rangle\rangle M_1$, then $M'[vx^{\pm}\rangle\rangle M_1'$ and $(M_1, M_1') \in \mathcal{B}$ for some M_1' with $v \in (Int_C^{\pm})^*$.
   - (N3) If $u \in Int_N$ and $M[u^{\pm}\rangle\rangle M_1$, then $M'[v\rangle\rangle M_1'$ and $(M_1, M_1') \in \mathcal{B}$ for some M_1' and $v \in (Int_C^{\pm})^*$.
   - (C1) If $x \in Out_C$ and $M'[x^{\pm}\rangle\rangle M_1'$, then $M[vx^{\pm}\rangle\rangle M_1$ and $(M_1, M_1') \in \mathcal{B}$ for some M_1 with $v \in (Int_N^{\pm})^*$.
   - (C2) If $x \in Out_i$ for some $i \in I$ and $M' \big|_{P_i}[x^{\pm}\rangle\rangle$, then $M'[x^{\pm}\rangle\rangle$. (no *computation interference*)
   - (C3) If $u \in Int_C$ and $M'[u^{\pm}\rangle\rangle M_1'$, then $M[v\rangle\rangle M_1$ and $(M_1, M_1') \in \mathcal{B}$ for some M_1 and $v \in (Int_N^{\pm})^*$.

[1] For the time being, our decomposition method does not produce such components, but this is definitely a future research topic. A recent approach can be found in [WW07].

In the most simple case, $(C_i)_{i\in I}$ consists of just one component C_1 (immediately implying (C2)) and H is empty; in this case we say that C_1 is a *(correct) implementation* of N. △

All the remarks after Definition 3.4 concerning $In_C \subseteq In_N$, $Out_C \subseteq Out_N$, computation interference and no matches for inputs of the implementation are also valid here. Regarding the last item, (C2) is actually also satisfied for $x \in Int_i$, since internal signals of one component are by the definition of parallel composition unknown to the other components.

The new features deal with internal signals; they extend the Definition 3.4 conservatively: for STGs without internal signals, the two correctness notions coincide. The consequence will be that the result about top-down decomposition in the next section also applies in the setting of the presented decomposition algorithm, where internal signals are not considered.

First of all, we allow the hiding of some output signals H in the parallel composition of the components; this concerns additional signals to enable communication between the components. It is no problem that we allow hiding at the 'top-level' only: by way of an example, assume that the components C_1 and C_2 communicate via a signal x which should not be visible to the other components; this would be modelled by $\big(((C_1||C_2)/\{x\}) \;||\; (||_{i\in I\setminus\{1,2\}}C_i)\big)/H$. Now this equals $(||_{i\in I}C_i)/(H\cup\{x\})$ by the properties (CA8) and (CA6) of a circuit algebra (p. 51), where (CA8) is applicable since x is only known to C_1 and C_2 and hence assumed to be not a signal of $||_{i\in I\setminus\{1,2\}}C_i$. We will use similar reasoning in Section 5.2 where a component will be replaced by a decomposition of its own.

In (N2) and (C1) outputs do not have to be matched directly; (N2) allows the components to prepare the production of this output by some internal signals, e.g. to achieve CSC or to inform other components; (C1) allows the specification to perform also internal signals. In any case, from an external point of view each output is matched by the same output.

In contrast, input signals must be matched directly; if the implementation could precede the input by some internal signals, the environment could produce the input as specified in N at a stage where the implementation is not ready yet to receive it, which could lead to malfunction as discussed above in connection with speed-independence and input properness. As for computation interference, the absence of this malfunction is only checked for markings appearing in $\mathcal{B}$, since only for these the problem is practically relevant.

In fact, the direct matching of inputs implies that the implementation is in a sense input proper, at least in its 'reachable behaviour': assume that $M_1[u^{\pm}\rangle\rangle M_2$ with

$u \in Int_C$, M_1 a reachable marking of C, and $M_2[a^{\pm}\rangle\rangle$ for some $a \in In_C$; then either there is no pair (M, M_1) in the STG-bisimulation (hence, M_1 will not be reached if C works in a proper environment) or $\neg M[a\rangle\rangle$ (a proper environment will not produce a) or $M_1[a\rangle\rangle$ by (N1).

Finally, (N3) and (C3) prescribe the matching of an internal signal by a sequence of internal signals – just as in ordinary weak bisimulation. Note that we have several internal signals, since these have to be implemented physically; but regarding external behaviour, the identity of an internal signal does not matter. In principle, performing an internal signal could make a choice, e.g. by disabling an output; according to these clauses, this choice has to be matched.

Translating the treatment of internal signals in the definition of the somewhat related notion of I/O-compatibility [CC02] to our setting (see also Section 5.5), one would require that e.g. in (N3) $(M_1, M') \in \mathcal{B}$ without involving any u – and analogously in (C3); the idea is that internal signals cannot make decisions in digital circuits. There are several reasons not to follow this idea. First of all, this concerns a property one might like all STGs to have and it is not related to comparing STGs or to the communication between circuits – in contrast to e.g. computation interference; if one wants this property in order to ensure physical implementability, it has to hold also for markings not appearing in $\mathcal{B}$. Therefore, this property has no adequate place in a correctness definition and should be required separately. Second, one might want to use ME-elements (see p. 25), which can make decisions; the respective signals could be internal to the parallel composition. We see it as an advantage that we can cover such cases. Finally, the alternative definition turned out to be technically inconvenient.

Observe that the alternative definition coincides with ours if the specification does not have internal signals; then, (N3) is never applicable, and in (C3) we have $v = \lambda$ and $M = M_1$.

There is another important comment. Our correctness definition concerns the correctness of a decomposition, but it also covers the question whether one STG is an implementation of another. With this notion, we will prove in Section 5.3 that speed-independent CSC-solving with Petrify produces a correct implementation.

One would like this implementation relation to be a preorder. Reflexivity is obvious (choose $\mathcal{B}$ as the identity), and transitivity will follow from our first main result in the next section. One would also like it to be a precongruence for the operations of interest. This is obvious for relabelling and easy for hiding (use the same STG-bisimulation). The much more important case of parallel composition will be discussed in the next section.

Actually, one can see a more general result for hiding just as easily: $(*)$ if $(C_i)_{i \in I}$ is

correct w.r.t. N when hiding H, then $(C_i)_{i\in I}$ is also correct w.r.t. N/H' when hiding $H \cup H'$. As a consequence, we can apply our decomposition algorithm [VW02, VK06] also to an STG N_1 with internal signals as follows. Since the algorithm can only decompose STGs without internal signals, we change the internal signals of N_1 to outputs obtaining an STG N_2 with $N_1 = N_2/Int_1$. Then we decompose N_2, obtaining a correct decomposition $(C_i)_{i\in I}$ of N_2. After that, the formerly internal signals are hidden in N_2 and in $||_{i\in I}C_i$ and from $(*)$ we get that $(C_i)_{i\in I}$ is a correct decomposition of $N_1 = N_2/Int_1$ when hiding Int_1.

5.2 Hierarchical Decomposition

In this section, we will show that correctness is preserved when we decompose a component of an STG decomposition into *subcomponents*. This result makes it possible to design and implement STGs in a top-down fashion.

In particular, such top-down decomposition can be useful for efficiency of our decomposition algorithm. For example, consider a case where only one component C_i of a decomposition needs a specific input signal a, which therefore will be removed from every other component by the decomposition algorithm. Alternatively, the algorithm could first construct a component C_j which generates every output signal that is not produced by C_i, and afterwards decompose it into smaller components. This way, the signal a will only be removed from one component C_j, which can improve performance. This and other strategies for decomposition are studied in Chapter 6.

Top-down decomposition as described above is possible under two minor conditions stated in the following theorem: the parallel composition of the subcomponents must have all output signals of the decomposed component and its internal signals must be unknown to the other components. The first condition is often automatically true or can be achieved easily by adding outputs just formally, the latter one is an obvious restriction required by our definition of parallel composition and can trivially be fulfilled by renaming internal signals.

Theorem 5.3 (Correctness of top-down Decomposition)

(1) Let N be an STG and $(C_i)_{i\in I}$ a correct decomposition of N when hiding H_C. Furthermore let $(C_k)_{k\in K}$ be a correct decomposition of some C_j when hiding H_K ($j \in I$, $I \cap K = \emptyset$). Then $(C_i)_{i\in I'}$ with $I' := I \cup K - \{j\}$ is a correct decomposition of N when hiding $H_C \cup H_K$ if $\bigcup_{k\in K} Out_{C_k} \setminus H_K = Out_{C_j}$ and $(\bigcup_{k\in K} Int_{C_k} \cup H_K) \cap \bigcup_{i\in I\setminus\{j\}} Sig_{C_i} = \emptyset$.

(2) The implementation relation is a preorder.

Proof.
(1) Let $C = (||_{i\in I} C_i)/H_C$, $C_K = (||_{k\in K} C_k)/H_K$ and $C' = (||_{i\in I'} C_i)/H$, where $H := H_C \cup H_K$. Without loss of generality assume $I = \{1, 2, \ldots, \mid I \mid\}$, $j = \mid I \mid$ and $K = \{\mid I \mid +1, \mid I \mid +2, \ldots, \mid I \mid + \mid K \mid\}$. We will write Out_i for Out_{C_i} etc.

First, we show that the parallel composition of $(C_i)_{i\in I'}$ is defined.
Obviously, $Loc_i \cap Loc_{i'}$ for different i, i' with either $i, i' \in I \setminus \{j\}$ or $i, i' \in K$, because $||(C_i)_{i\in I}$ and $||(C_k)_{k\in K}$ are defined. Therefore let $k \in K, i \in I\setminus\{j\}$; then $Loc_k \cap Loc_i = (Int_k \cup Out_k) \cap Loc_i = Int_k \cap Loc_i \cup Out_k \cap Loc_i = \emptyset \cup Out_k \cap Loc_i$, by assumption about Int_k. $Out_k \cap Loc_i \subseteq (Out_j \cup H_K) \cap Loc_i = (Out_j \cap Loc_i) \cup (H_K \cap Loc_i) \subseteq (Loc_j \cap Loc_i) \cup (H_K \cap Loc_i) = \emptyset$ by the assumption about H_K and because $||_{i\in I} C_i$ is defined.

For i, i' as above, $Int_i \cap In_{i'} = \emptyset$. Let therefore i, k be as above, then $In_k \cap Int_i \subseteq In_j \cap Int_i = \emptyset$ and $Int_k \cap In_i = \emptyset$ by the assumptions.

Next, we show the requirements for the sets of output and input signals.

$$\begin{aligned}
Out_{C'} &= \bigcup_{i\in I'} Out_i \setminus H \\
&= (\bigcup_{i\in I\setminus\{j\}} Out_i \cup \bigcup_{k\in K} Out_k) \setminus H \\
&= (\bigcup_{i\in I\setminus\{j\}} Out_i \cup (\bigcup_{k\in K} Out_k \setminus H_K)) \setminus H_C \\
&= (\bigcup_{i\in I\setminus\{j\}} Out_i \cup Out_j) \setminus H_C \\
&= \bigcup_{i\in I} Out_i \setminus H_C \\
&\subseteq Out_N
\end{aligned}$$

The third equality holds by the second assumption on H_K.
For the input signals, we have

$$In_C = \bigcup_{i\in I} In_i \setminus \bigcup_{i\in I} Out_i \subseteq In_N$$

and

$$\bigcup_{k\in K} In_k \setminus \bigcup_{k\in K} Out_k \subseteq In_j$$

It follows that

$$\begin{aligned}
In_{C'} &= \bigcup_{i\in I'} In_i \setminus \bigcup_{i\in I'} Out_i \\
&= (\bigcup_{i\in I\setminus\{j\}} In_i \cup \bigcup_{k\in K} In_k) \setminus (\bigcup_{i\in I\setminus\{j\}} Out_i \cup \bigcup_{k\in K} Out_k) \\
&\subseteq (\bigcup_{i\in I\setminus\{j\}} In_i \cup In_j) \setminus (\bigcup_{i\in I\setminus\{j\}} Out_i \cup \bigcup_{k\in K} Out_k) \\
&\overset{(*)}{\subseteq} \bigcup_{i\in I} In_i \setminus (\bigcup_{i\in I\setminus\{j\}} Out_i \cup Out_j) = In_C \subseteq In_N
\end{aligned}$$

The inclusion $(*)$ might fail if we had only $(\bigcup_{k\in K} Out_k) \setminus H_K \subseteq Out_j$. Observe that we do not need to consider hiding here.

We proceed with the main part of this proof: the requirements for an STG-bisimulation between the markings of N and C'.

Let $\mathcal{B}_1$ be the STG-bisimulation between the markings of N and C and $\mathcal{B}_2$ the one between the markings of C_j and C_K. We define a relation $\mathcal{B}$ between the markings of N and C' as follows:

$$\begin{aligned}
& (M_1, M_2, M_3) \in \mathcal{B} \\
\Leftrightarrow\ & (M_1, M_2, M_j) \in \mathcal{B}_1 \text{ and } (M_j, M_3) \in \mathcal{B}_2 \text{ for some } M_j
\end{aligned}$$

where M_1 denotes a marking of N, M_2 a marking of $C_1||\ldots||C_{|I|-1}$, M_j one of C_j and M_3 a marking of $C_{|I|+1}||\ldots||C_{|I|+|K|}$; thus (M_2, M_j) can be regarded as marking of C and (M_2, M_3) as one of C' and we write (M_1, M_2, M_3) instead of $(M_1, (M_2, M_3))$ etc. We will show that $(C_i)_{i\in I'}$ is a correct decomposition of N when hiding H due to STG-bisimulation $\mathcal{B}$.

(1): Obviously fulfilled by definition of $\mathcal{B}$.

(2): Let $(M_1, M_2, M_3) \in \mathcal{B}$ due to $(M_1, M_2, M_j) \in \mathcal{B}_1$ and $(M_j, M_3) \in \mathcal{B}_2$ for some marking M_j of C_j.

(N1): $a \in In_N$ and $M_1[a^{\pm}\rangle\rangle\hat{M}_1$.

1. Let $a \in In_{C'} \subseteq In_C$ (see above) and therefore $(\mathcal{B}_1)$ $(M_2, M_j)[a^{\pm}\rangle\rangle$ $(\hat{M}_2, \hat{M}_j)$ and $(\hat{M}_1, \hat{M}_2, \hat{M}_j) \in \mathcal{B}_1$ for some $(\hat{M}_2, \hat{M}_j)$.

   (a) If $a \notin In_j$ we get $M_j = \hat{M}_j$ and immediately $(M_2, M_3)[a^{\pm}\rangle\rangle$ $(\hat{M}_2, M_3)$ with $(\hat{M}_1, \hat{M}_2, M_3) \in \mathcal{B}$.

(b) $a \in In_j$ with $M_j[a^{\pm}\rangle\rangle\hat{M}_j$ implies $(\mathcal{B}_2)$ either $a \in In_K$ and $M_3[a^{\pm}\rangle\rangle\hat{M}_3$, $(\hat{M}_j, \hat{M}_3) \in \mathcal{B}_2$ for some $\hat{M}_3$ or $a \notin In_K$ and $(\hat{M}_j, M_3) \in \mathcal{B}_2$.
In the first case, we get $(M_2, M_3)[a^{\pm}\rangle\rangle(\hat{M}_2, \hat{M}_3)$ with $(\hat{M}_1, \hat{M}_2, \hat{M}_3) \in \mathcal{B}$.
In the latter one we get $(M_2, M_3)[a^{\pm}\rangle\rangle(\hat{M}_2, M_3)$ with $(\hat{M}_1, \hat{M}_2, M_3) \in \mathcal{B}$.

2. Let $a \notin In_{C'}$. There are two reasons for this:

   (a) $a \notin In_C$. Then, $(\hat{M}_1, M_2, M_j) \in \mathcal{B}_1$ and by definition $(\hat{M}_1, M_2, M_3) \in \mathcal{B}$.

   (b) $a \in In_j$, but $a \notin In_i$ for $i \neq j$ and $a \notin In_K$ (a only element of C_j). Therefore, $a \in In_C$ and $(M_2, M_j)[a^{\pm}\rangle\rangle(M_2, \hat{M}_j)$ with $(\hat{M}_1, M_2, \hat{M}_j) \in \mathcal{B}_1$, since a cannot be a signal of any other component. $a \notin In_K$ implies $(\hat{M}_j, M_3) \in \mathcal{B}_2$ and by definition $(\hat{M}_1, M_2, M_3) \in \mathcal{B}$.

(N2): Let $x \in Out_N$ and $M_1[x^{\pm}\rangle\rangle\hat{M}_1$.
Then $(\mathcal{B}_1)$ $(M_2, M_j)[vx^{\pm}\rangle\rangle(\hat{M}_2, \hat{M}_j)$ and $(\hat{M}, \hat{M}_2, \hat{M}_j) \in \mathcal{B}_1$ for some $(\hat{M}_2, \hat{M}_j)$ and $v \in (Int_C^{\pm})^*$. Let $v' = vx^{\pm} = v_1 v_2 \dots v_n$ with $v_i \in Sig_C^{\pm}$ for $i = 1, \dots, n$ and

$$(M_2, M_j) = (M_2^0, M_j^0)[v_1\rangle\rangle(M_2^1, M_j^1)[v_2\rangle\rangle(M_2^2, M_j^2)\dots$$

$$(M_2^{n-2}, M_j^{n-2})[v_{n-1}\rangle\rangle(M_2^{n-1}, M_j^{n-1})[v_n\rangle\rangle(M_2^n, M_j^n) = (\hat{M}_2, \hat{M}_j)$$

We will show by induction over $m \in \{0, \dots, n\}$ that

$$\exists w_m \in (Int_{C'}^{\pm})^*\{x^{\pm}, \lambda\}, M_3^m : (M_2, M_3)[w_m\rangle\rangle(M_2^m, M_3^m)$$

$$\wedge w_m{\downarrow_{Ext_{C'}}} = (v_1 \dots v_m){\downarrow_{Ext_C}} \wedge (M_j^m, M_3^m) \in \mathcal{B}_2$$

Observe that the case $m = n$ proves our claim.
For $m = 0$ let $M_3^0 = M_3$ and $w_0 = \lambda$. By assumption $(M_j^0, M_3^0) \in \mathcal{B}_2$.
Let now $m < n$ and:

- $(M_2, M_3)[w_m\rangle\rangle(M_2^m, M_3^m)$
- $w_m{\downarrow_{Ext_{C'}}} = (v_1 \dots v_m){\downarrow_{Ext_C}}$
- $(M_j^m, M_3^m) \in \mathcal{B}_2$
- $(M_2^m, M_j^m)[v_{m+1}\rangle\rangle(M_2^{m+1}, M_j^{m+1})$

We distinguish several cases, where in items (3) - (5) we have either $v_{m+1} \in H_C$ or $v_{m+1} = v_n = x^{\pm}$:

1. $v_{m+1} \in Int_i$ for $i \in I \setminus \{j\} \Rightarrow M_j^{m+1} = M_j^m, M_3^{m+1} = M_3^m$, $w_{m+1} = w_m v_{m+1}$ and $(M_j^{m+1}, M_3^{m+1}) \in \mathcal{B}_2$.

2. $v_{m+1} \in Int_j \Rightarrow M_2^{m+1} = M_2^m, M_3^m[w'_m\rangle\rangle M_3^{m+1}$, $w'_m \in (Int_K^{\pm})^*$, $w_{m+1} = w_m w'_m$ and $(M_j^{m+1}, M_3^{m+1}) \in \mathcal{B}_2$.

3. $v_{m+1} \in Out_i$ for $i \in I \setminus \{j\}$ and $v_{m+1} \notin In_j$: ref. item (1).

4. $v_{m+1} \in Out_i$ for $i \in I \setminus \{j\}$ and $v_{m+1} \in In_j$: (N1) implies

   (a) $v_{m+1} \in In_K \Rightarrow M_3^m[v_{m+1}\rangle\rangle M_3^{m+1}$ with $(M_j^{m+1}, M_3^{m+1}) \in \mathcal{B}_2$ and $w_{m+1} = w_m v_{m+1}$.

   (b) $v_{m+1} \notin In_K \Rightarrow M_3^{m+1} = M_3^m$ with $(M_j^{m+1}, M_3^{m+1}) \in \mathcal{B}_2$ and $w_{m+1} = w_m v_{m+1}$.

5. $v_{m+1} \in Out_j$: this implies $M_3^m[w'_{m+1} v_{m+1}\rangle\rangle M_3^{m+1}$ with $w'_{m+1} \in (Int_K^{\pm})^*$, $(M_j^{m+1}, M_3^{m+1}) \in \mathcal{B}_2$ and $w_{m+1} = w_m w'_{m+1} v_{m+1}$.

(N3): Let $u \in Int_N$ and $M_1[u^{\pm}\rangle\rangle \hat{M}_1$. Therefore $(M_2, M_j)[v\rangle\rangle(\hat{M}_2, \hat{M}_j)$ with $(\hat{M}_1, \hat{M}_2, \hat{M}_j) \in \mathcal{B}_1$ and $v \in (Int_C^{\pm})^*$ for some $(\hat{M}_2, \hat{M}_j)$. At this point the proof can be continued analogously to the previous item.

(C1): Let $x \in Out_{C'}$ and $(M_2, M_3)[x^{\pm}\rangle\rangle(\hat{M}_2, \hat{M}_3)$.

1. If $x \in Out_K \setminus H_K = Out_j$ it follows that $M_j[v x^{\pm}\rangle\rangle \hat{M}_j$, $(\hat{M}_j, \hat{M}_3) \in \mathcal{B}_2$ for some $\hat{M}_j$ and $v \in (Int_j^{\pm})^*$.

   Let $M_j[v\rangle\rangle M'_j[x^{\pm}\rangle\rangle \hat{M}_j$; then $(M_2, M_j)[v\rangle\rangle(M_2, M'_j)$ and by (C3) $M_1[w_1\rangle\rangle M'_1$ with $w_1 \in (Int_N^{\pm})^*$ and $(M'_1, M_2, M'_j) \in \mathcal{B}_1$ for some $\hat{M}'_1$.

   Since $(M_2, M'_j)[x^{\pm}\rangle\rangle(\hat{M}_2, \hat{M}_j)$ (where $M_2[x^{\pm}\rangle\rangle \hat{M}_2$ or $\hat{M}_2 = M_2$), we get by (C1) for $\mathcal{B}_1$ that $M'_1[w_2 x^{\pm}\rangle\rangle \hat{M}_1$ and $(\hat{M}_1, \hat{M}_2, \hat{M}_j) \in \mathcal{B}_1$ for some $\hat{M}_1$ and $w_2 \in (Int_N^{\pm})^*$. Altogether, we get that $M_1[w_1 w_2 x\rangle\rangle \hat{M}_1$ with $w_1 w_2 \in (Int_N^{\pm})^*$ and $(\hat{M}_1, \hat{M}_2, \hat{M}_3) \in \mathcal{B}$.

2. If $x \notin Out_K \setminus H_K = Out_j$, there exists an $m \in I \setminus \{j\}$ such that $x \in Out_m \subseteq Out_C$.

(a) $x \notin In_j$ implies that neither M_3 nor M_j are changed when firing $x^{\pm}$: $\hat{M}_3 = M_3$ and $(M_2, M_j)[x^{\pm}\rangle\rangle(\hat{M}_2, M_j)$; we get directly $(\mathcal{B}_1)$ $M_1[wx^{\pm}\rangle\rangle\hat{M}_1$, $(\hat{M}_1, \hat{M}_2, M_j) \in \mathcal{B}_1$ for some $\hat{M}_1$ and $w \in (Int_N^{\pm})^*$, and by definition of $\mathcal{B}$: $(\hat{M}_1, \hat{M}_2, \hat{M}_3) \in \mathcal{B}$.

(b) If $x \in In_j$ then $M_j[x^{\pm}\rangle\rangle\hat{M}_j$ by (C2) for $\mathcal{B}_1$ and by (N1)

i. $x \notin In_K$, $\hat{M}_3 = M_3$ and $(\hat{M}_j, M_3) \in \mathcal{B}_2$.

ii. $x \in In_K$ and $(\hat{M}_j, \hat{M}_3) \in \mathcal{B}_2$.

In any case, $(M_2, M_j)[x^{\pm}\rangle\rangle(\hat{M}_2, \hat{M}_j)$ and $M_1[wx^{\pm}\rangle\rangle\hat{M}_1$ with $w \in (Int_N^{\pm})^*$ and $(\hat{M}_1, \hat{M}_2, \hat{M}_j) \in \mathcal{B}_1$, hence $(\hat{M}_1, \hat{M}_2, \hat{M}_3) \in \mathcal{B}$.

(C2): Let $x \in Out_m, m \in I'$ and $(M_2, M_3)|_{P_m}[x^{\pm}\rangle\rangle$.

1. $x \in Out_m$ for some $m \in I \setminus \{j\}$ and $M_2|_{P_m}[x^{\pm}\rangle\rangle$. Then $(\mathcal{B}_1)$ $(M_2, M_j)[x^{\pm}\rangle\rangle$ and therefore $(M_2, M_3)[x^{\pm}\rangle\rangle$, because either $x \notin In_j$ and $x \notin Sig_k$ for $k \in K$ or $x \in In_j$, and by (N1) for $\mathcal{B}_2$ either $M_3[x^{\pm}\rangle\rangle$ or $x \notin Sig_k$, $k \in K$.

2. $x \in Out_m$ for some $m \in K$ and $M_3|_{P_m}[x^{\pm}\rangle\rangle$. Hence, $M_3[x^{\pm}\rangle\rangle$ by (C2) for $\mathcal{B}_2$.

   (a) If $x \in Out_K$, then $M_j[v\rangle\rangle M_j'[x^{\pm}\rangle\rangle$ by (C1) for $\mathcal{B}_2$ for some $v \in (Int_j^{\pm})^*$ and M_j'. Since C_j can fire its internal signals without changing the state of the other components, we get $(M_2, M_j)[v\rangle\rangle(M_2, M_j')$ and $(M_1', M_2, M_j') \in \mathcal{B}_1$ for some M_1'. Applying (C2) for $\mathcal{B}_1$ we get $(M_2, M_j')[x^{\pm}\rangle\rangle$ and therefore $(M_2, M_3)[x^{\pm}\rangle\rangle$, too.

   (b) If $x \in Int_K$, i.e. $x \in H_K$, then $M_3[x^{\pm}\rangle\rangle$ proves immediately $(M_2, M_3)[x^{\pm}\rangle\rangle$.

(C3): Let $u \in Int_{C'}$ and $(M_2, M_3)[u^{\pm}\rangle\rangle(\hat{M}_2, \hat{M}_3)$.

(1) If $u \in Int_i$ for $i \in I \setminus \{j\}$, then $\hat{M}_3 = M_3$, $(M_2, M_j)[u^{\pm}\rangle\rangle(\hat{M}_2, M_j)$ and by (C3) for $\mathcal{B}_1$: $M_1[v\rangle\rangle\hat{M}_1$, $w \in (Int_N^{\pm})^*$ and $(\hat{M}_1, \hat{M}_2, M_j) \in \mathcal{B}_1$ for some $\hat{M}_1$ and therefore $(\hat{M}_1, \hat{M}_2, \hat{M}_3) \in \mathcal{B}$.

(2) If $u \in Int_k$ for $k \in K$, then $\hat{M}_2 = M_2$ and by (C3) for $\mathcal{B}_2$: $M_j[v\rangle\rangle\hat{M}_j$, $v \in (Int_j^{\pm})^*$ and $(\hat{M}_j, \hat{M}_3) \in \mathcal{B}_2$ for some $\hat{M}_j$. Let $v = v_1 v_2 \dots v_n$, $v_i \in Int_j^{\pm}$ and $M_j = M_j^0[v_1\rangle\rangle M_j^1 \dots M_j^{n-1}[v_n\rangle\rangle M_j^n = \hat{M}_j$. By (C3) for $\mathcal{B}_1$ we get that $M_1 = M_1^0[w_1\rangle\rangle M_1^1[w_2\rangle\rangle \dots M_1^{n-1}[w_n\rangle\rangle$ $M_1^n = \hat{M}_1$ with $w_i \in (Int_N^{\pm})^*$ and $(M_1^m, \hat{M}_2, M_j^n) \in \mathcal{B}_2$ for every $m = 0, \dots, n$. In particular, $M_1[w_1 \dots w_m\rangle\rangle\hat{M}_1$, $(\hat{M}_1, \hat{M}_2, \hat{M}_j) \in \mathcal{B}_1$ and therefore $(\hat{M}_1, \hat{M}_2, \hat{M}_3) \in \mathcal{B}$.

(3) If $u \in Out_i \cap H_C$ for $i \in I \setminus \{j\}$.

   (a) $u \in In_j$. Then $(M_2, M_j)[u^{\pm}\rangle\rangle(\hat{M}_2, \hat{M}_j)$ and by (C3) for $\mathcal{B}_1$: $M_1[v\rangle\rangle\hat{M}_1$, $v \in (Int_N^{\pm})^*$ and $(\hat{M}_1, \hat{M}_2, \hat{M}_j) \in \mathcal{B}_1$; (N1) for $\mathcal{B}_2$ implies either $M_3[u^{\pm}\rangle\rangle\hat{M}_3$ or $u \notin In_K$ and $\hat{M}_3 = M_3$; in both cases $(\hat{M}_j, \hat{M}_3) \in \mathcal{B}_2$ and it follows that $(\hat{M}_1, \hat{M}_2, \hat{M}_3) \in \mathcal{B}$.

   (b) $u \notin In_j$. Analogous to item (1)

(4) If $u \in Out_k \cap H_C$ for $k \in K$. Then, $M_3[u^{\pm}\rangle\rangle\hat{M}_3$ and by (C1) for $\mathcal{B}_2$ $M_j[vu^{\pm}\rangle\rangle\hat{M}_j$ and $(\hat{M}_j, \hat{M}_3) \in \mathcal{B}_2$ for $v \in (Int_j^{\pm})^*$. Furthermore, $(M_2, M_j)[vu^{\pm}\rangle\rangle(\hat{M}_2, \hat{M}_j)$ and by (C3) for $\mathcal{B}_1$ we get $M_1[w\rangle\rangle\hat{M}_1$, $w \in (Int_N^{\pm})^*$ and $(\hat{M}_1, \hat{M}_2, \hat{M}_j) \in \mathcal{B}_1$ (with $\hat{M}_2 = M_2$ if $u \notin In_m$ for $m \in I \setminus \{j\}$). It follows that $(\hat{M}_1, \hat{M}_2, \hat{M}_3) \in \mathcal{B}$.

(5) If $u \in Out_k \cap H_K$ for $k \in K$. Then, by (C3) for $\mathcal{B}_2$: $M_j[v\rangle\rangle\hat{M}_j$, $v \in (Int_j^{\pm})^*$ and $(\hat{M}_j, \hat{M}_3) \in \mathcal{B}_2$ and $\hat{M}_2 = M_2$. Thus, $(M_2, M_j)[v\rangle\rangle$ $(\hat{M}_2, \hat{M}_j)$ and by (C3) for $\mathcal{B}_1$: $M_1[w\rangle\rangle\hat{M}_1$, $w \in (Int_N^{\pm})^*$ and $(\hat{M}_1, \hat{M}_2, \hat{M}_j) \in \mathcal{B}_1$. Therefore by definition of $\mathcal{B}$, $(\hat{M}_1, \hat{M}_2, \hat{M}_3) \in \mathcal{B}$. □

Regarding (2), we already know that the implementation relation is reflexive; transitivity is just a special case of (1), where both hiding sets are empty and the decompositions have just one component each. So (1) tells us that, if C is an implementation of N and C' an implementation of C, then C' is an implementation of N – except that we do not have the two extra conditions required in (1). Observe that the second condition is trivially true since $\bigcup_{i \in I \setminus \{j\}} Sig_{C_i}$ is empty. The first condition is only needed to prove claims that are obvious for this restricted case, namely that the parallel composition C' (which has only one component here) is defined and that $In_{C'} \subseteq In_N$. □

Remark: One might expect that refining a component C_j of

$$(||_{i \in I} C_i)/H_C \text{ with } (||_{k \in K} C_k)/H_K$$

would give the STG

$$\left(||_{i \in I \setminus \{j\}} C_i \;||\; (||_{k \in K} C_k/H_K)\right)/H_C\,,$$

where there is not just one hiding on the top-level as in the theorem. With the same reasoning already used in the discussion of Definition 5.2, we can derive from the properties (CA8) (use the second assumption on H_K) and (CA6) of a circuit algebra

that for $H = H_C \cup H_K$:

$$\begin{aligned} & \left(||_{i \in I \setminus \{j\}} C_i \;||\; \left(||_{k \in K} C_k / H_K\right)\right) / H_C \\ = \; & \left(\left(||_{i \in I'} C_i\right) / H_K\right) / H_C \\ = \; & ||_{i \in I'} C_i / H \end{aligned}$$

As explained after Definition 5.2, our correctness definition coincides with Definition 3.4 if we restrict ourselves to STGs without internal signals; hence, the above theorem also holds in this setting (where of course no hiding is applied, i.e. the hiding sets are taken to be empty). Therefore, the theorem can indeed be used to improve our decomposition algorithm as explained at the beginning of this section.

Surprisingly, the theorem has also an impact on the question whether the implementation relation between STGs is a precongruence for parallel composition, which we will show under some mild restrictions now. Recall that, for some $N_1||N_2$ to be defined, we only had some syntactic requirements regarding the signal sets; but the composition only makes sense in the area of circuits, if we also ensure absence of computation interference; for the following definition cf. the discussion on condition (C2) of Definition 5.2.

Definition 5.4 (Interference-free)
A parallel composition $N_1||N_2$ is *interference-free* if, for all its reachable markings $M_1 \dot\cup M_2$, $i \in \{1,2\}$ and $x \in Out_i$, $M_i[x^{\pm}\rangle\rangle$ implies $(M_1 \dot\cup M_2)[x^{\pm}\rangle\rangle$. △

Corollary 5.5
If N_2 is a correct implementation of N_1, N_1 and N_2 have the same output signals, and $N_1||N$ is a well-defined and interference-free parallel composition, then (N_2, N) is a correct decomposition of $N_1||N$, i.e. $N_2||N$ is a well-defined and interference-free parallel composition and a correct implementation of $N_1||N$.

Proof. Since the composition is interference-free, the identity is an STG-bisimulation showing that the family (N_1, N) is a correct decomposition of $N_1||N$; note that in this setting all conditions for an STG-bisimulation are trivially fulfilled except for (C2). With this observation, the claim follows from Theorem 5.3. □

Note that each of our operations hiding, renaming and parallel composition with another STG changes the set of output signals in the same way, such that equality of these sets is preserved.

Corollary 5.6 (Implementation relation as precongruence)

The implementation relation is a precongruence for hiding, relabelling and parallel composition when restricted to STGs with the same output signals and interference-free parallel compositions.

5.3 CSC Solving

In this section, we will prove that speed-independent CSC-solving fits into our correctness definition, i.e. that it leads to a correct implementation. Theorem 5.3 then implies that speed-independent CSC-solving can be combined with our decomposition algorithm. The latter could be shown directly without this theorem, but its use makes the following proof much easier, because we have to consider only one component. First, we will introduce the operation of input proper event insertion, which e.g. is used by the tool PETRIFY to achieve CSC.

Given an STG without CSC, one can (in many cases) insert internal signals into the STG such that their values distinguish between the markings with equal state vectors but different enabled outputs. This insertion takes place on the level of reachability graphs. It is also possible to derive an STG for the modified reachability graph, and although this is not important for the synthesis of a circuit, it fits our manner-of-speaking well. In Example 5.1, CSC is solved for the VME controller of the previous examples according to the following considerations.

We take the following definition of input proper event insertions from [CKK+02]. One can perform a number of these operations, arriving at an STG with CSC, and this we call *speed-independent CSC-solving*.

Definition 5.7 (Event insertion)
Let N be a deterministic STG, $u^{\pm}$ a signal transition not appearing in N for a (possibly new) internal signal u and $R \subseteq [M_N\rangle$. The *event insertion of* $u^{\pm}$ *at region* R *into* N modifies the reachability graph RG_N (and results in a corresponding STG N') as follows (cf. Fig. 5.1):

(1) For every marking $M \in R$ add a duplicate M' and add the transition $M[u^{\pm}\rangle\rangle M'$.

(2) If $M_1, M_2 \in R$ and $M_1[s^{\pm}\rangle\rangle M_2$, add the transition $M_1'[s^{\pm}\rangle\rangle M_2'$.

(3) If $M_1 \in R$, $M_2 \notin R$ and $M_1[s^{\pm}\rangle\rangle M_2$, remove this transition and add $M_1'[s^{\pm}\rangle\rangle M_2$.

(4) The initial marking of N' is the same as that of N. Add u to *Int*.

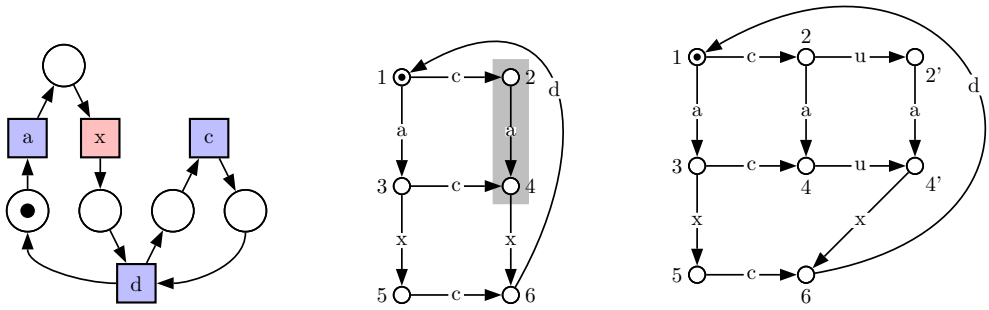


Figure 5.1: Example of an event insertion. *left:* A Petri net (to keep it small, transitions are labelled with signals) *middle:* Its reachability graph. The two gray states are the region R where the new event u will be inserted. *right:* The reachability graph with the inserted event u. The marking relation is $\mathcal{M} = \{(1,1),(2,2),(2,2'),(3,3),(4,4),(4,4'),(5,5),(6,6)\}$.

The insertion is called *input proper*, if there is no $M_1[a^{\pm}\rangle\rangle M_2$ in RG_N with $a \in In$, $M_1 \in R$ and $M_2 \notin R$.

We define the *marking relation* $\mathcal{M}$ between the markings of N and of N' such that $(M_1, M_2) \in \mathcal{M}$ if $M_2 = M_1$ or $M_2 = M_1'$.

It is not hard to see that N' as above is deterministic again. An example for an event insertion can be found in Figure 5.1.

The following result explains the definition of an input proper event insertion and why we speak of speed-independent CSC-solving.

Of course, the insertion of just a single signal edge will generate an inconsistent STG. This is no problem for our correctness notion, but for practical purposes event insertions are always performed in pairs (u^+ and u^- for a new internal signal u), such that the result is consistent again; this is called *signal insertion*.

The following proposition is needed to prove the main theorem of this section and it explains why we speak of an input proper insertion.

Proposition 5.8
Let N be an input proper STG and let N' be obtained by the insertion of $u^{\pm}$ at R. Then N' is input proper if and only if the insertion is.

The proof needs the following lemma.

Lemma 5.9
Let N be an STG and N' be obtained from N by the event insertion of $u^{\pm}$ at region R. Let $(M_1, M_2) \in \mathcal{M}$ and $a \in Sig_N$.

(1) If $M_2 = M_1'$, then $M_1[a^{\pm}\rangle\rangle\hat{M}_1$ in N implies $M_2[a^{\pm}\rangle\rangle\hat{M}_2$ in N' with $(\hat{M}_1, \hat{M}_2) \in \mathcal{M}$.

(2) $M_2[a^{\pm}\rangle\rangle\hat{M}_2$ in N' implies $M_1[a^{\pm}\rangle\rangle\hat{M}_1$ in N with $(\hat{M}_1, \hat{M}_2) \in \mathcal{M}$.

Proof.

(1) $M_2 = M_1'$ implies $M_1 \in R$ and by Definition 5.7.2,3 $M_2[a^{\pm}\rangle\rangle\hat{M}_2$ in N' with $(\hat{M}_1, \hat{M}_2)$, where we have $\hat{M}_2 = \hat{M}_1'$ if case 2 is applicable and $\hat{M}_2 = \hat{M}_1$ if case 3 is applicable.

(2) The reasoning is similar. □

Now, we come to the main results of this section.

Proof of Proposition 5.8. Assume the insertion is not input proper because of $M_1[a^{\pm}\rangle\rangle M_2$; then we have in N': $M_1[u^{\pm}\rangle\rangle M_1'$ due to Definition 5.7.1 and $\neg M_1[a^{\pm}\rangle\rangle$ and $M_1'[a^{\pm}\rangle\rangle M_2$ due to 3.

Vice versa, assume that N' is not input proper due to $M_1[v^{\pm}\rangle\rangle M_2$, $\neg M_1[a^{\pm}\rangle\rangle$ and $M_2[a^{\pm}\rangle\rangle M_3$ for some $a \in In$. If $v^{\pm}$ is the newly inserted $u^{\pm}$, then $M_2 = M_1'$; we cannot have $M_2[a^{\pm}\rangle\rangle M_3$ due to Definition 5.7.2 because then $M_1[a^{\pm}\rangle\rangle$, and thus we must have $M_2[a^{\pm}\rangle\rangle M_3$ due to Definition 5.7.3, i.e. the insertion is not input proper because of $M_1[a^{\pm}\rangle\rangle M_3$ in N.

Otherwise, there are $\hat{M}_1$ and $\hat{M}_2$ with $(\hat{M}_1, M_1) \in \mathcal{M}$, $(\hat{M}_2, M_2) \in \mathcal{M}$, $\hat{M}_1[v^{\pm}\rangle\rangle\hat{M}_2$ in N and $\hat{M}_2[a^{\pm}\rangle\rangle$ in N by Lemma 5.9.2. Since N is input proper, this implies $\hat{M}_1[a^{\pm}\rangle\rangle$; since $\neg M_1[a^{\pm}\rangle\rangle$ in N', this in turn implies $\hat{M}_1 = M_1$ by Lemma 5.9.1. Thus, $\hat{M}_1$ has *lost* an $a^{\pm}$-transition during the event insertion; this can only be due to Definition 5.7.3, and also in this case, the insertion is not input proper. □

Theorem 5.10 (Correctness of CSC solving)
Let N be an STG and N' be obtained from N by speed-independent CSC-solving, then N' is a correct implementation of N and vice versa.

Proof. N' is obtained from N by a sequence of input proper event insertions. It suffices to show the claims for one such insertion, and then the theorem follows from Theorem 5.3.2. Thus, assume that N' is obtained from N by the input proper insertion of $u^{\pm}$ at R, with $\mathcal{M}$ being the corresponding marking relation. Obviously, we have $In_N = In_{N'}$, $Out_N = Out_{N'}$.

N' is a correct implementation of N: we will show that $\mathcal{M}$ is an STG-bisimulation for N and N'.

(1): Fulfilled by definition of event insertion.

(2): For this part, observe that (C2) is trivially fulfilled, because we consider only one component. Let now $(M_1, M_2) \in \mathcal{M}$.

For (N1)–(N3), we only have to consider $M_2 = M_1$ due to Lemma 5.9.1. If $a \in In_N$ and $M_1[a^{\pm}\rangle\rangle\hat{M}_1$ in N, then Definition 5.7.3 cannot be applicable since the insertion is input proper, hence $M_1[a^{\pm}\rangle\rangle\hat{M}_1$ in N' as well with $(\hat{M}_1, \hat{M}_1) \in \mathcal{M}$ and (N1) follows.

Now let $x \in Loc_N$ and $M_1[x^{\pm}\rangle\rangle\hat{M}_1$ in N. Then we have in N' that $M_1[x^{\pm}\rangle\rangle\hat{M}_1$ if $M_1 \notin R$ or $M_2 \in R$ and $M_1[u^{\pm}x^{\pm}\rangle\rangle\hat{M}_1$ otherwise; obviously $(\hat{M}_1, \hat{M}_1) \in \mathcal{M}$ and (N2) and (N3) follow.

(C1/C3): Let $x \in Loc_{N'}$ and $M_2[x^{\pm}\rangle\rangle\hat{M}_2$ in N'. If $x^{\pm}$ is not the inserted $u^{\pm}$, Lemma 5.9.2 implies $M_1[x^{\pm}\rangle\rangle\hat{M}_1$ and $(\hat{M}_1, \hat{M}_2) \in \mathcal{M}$. Otherwise, we have $M_2 = M_1$, $\hat{M}_2 = M_1'$ and $(M_1, \hat{M}_2) \in \mathcal{M}$.

N is a correct implementation of N': we will argue that $\mathcal{M}^{-1}$ is an STG-bisimulation for N' and N.

(1): Fulfilled by definition of event insertion.

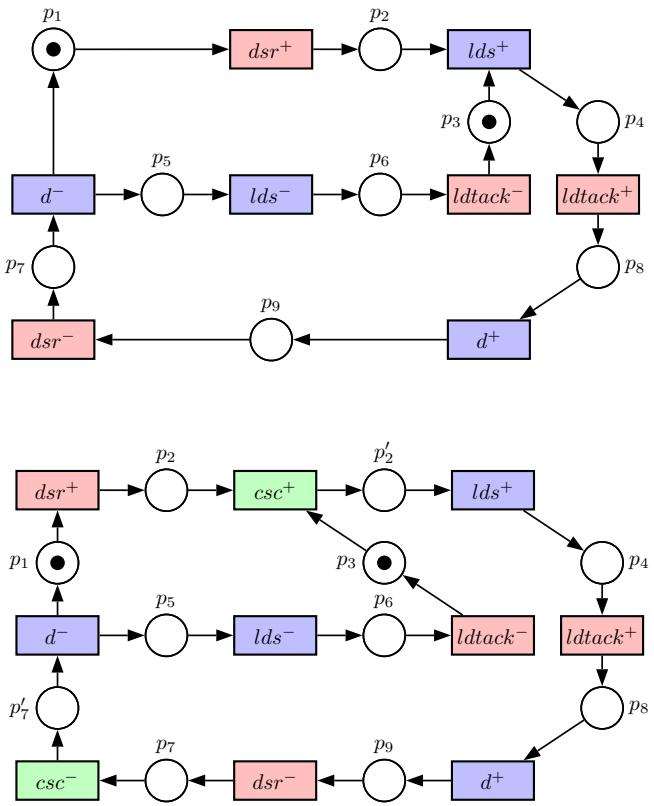
(2): For this part, observe that (C2) is trivially fulfilled, because we consider only one component. Furthermore, N2/N3/C1/C3 are dual to C1/C3/N2/N3 above, so we only have to check (N1), which follows directly from Lemma 5.9.2, since $In_N = In_{N'}$. □

Now we can conclude that speed-independent CSC-solving can be combined with decomposition.

Corollary 5.11
Let $(C_i)_{i\in I}$ be a correct decomposition of N when hiding H, and let C_i' be obtained from C_i by speed-independent CSC-solving for all $i \in I$. Then $(C_i')_{i\in I}$ is a correct decomposition of N when hiding H.

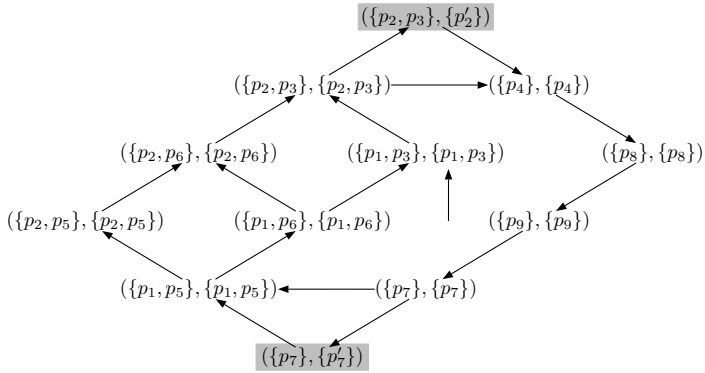
Proof. It is sufficient to consider one component C_j and to apply induction afterwards. By Theorem 5.10, C_j' is a correct decomposition of C_j. Furthermore it fulfils the preconditions of Theorem 5.3.1, esp. the crucial first one on H_K since $H_K = \emptyset$ and event insertion does not change the sets of output and of input signals. Therefore, $((C_i)_{i\in I\setminus\{j\}}, C_j')$ is a correct decomposition of N. □

In Example 3.6 it was shown that the VME bus controller does not fulfil CSC. Consequently, also component C_1 (left, see Example 3.9) does not have CSC.
Solving CSC in C_1 yields C_1' (right). This is done by adding one new internal signal *csc* in the form of two new events $csc+, csc-$ as described in Definition 5.7.

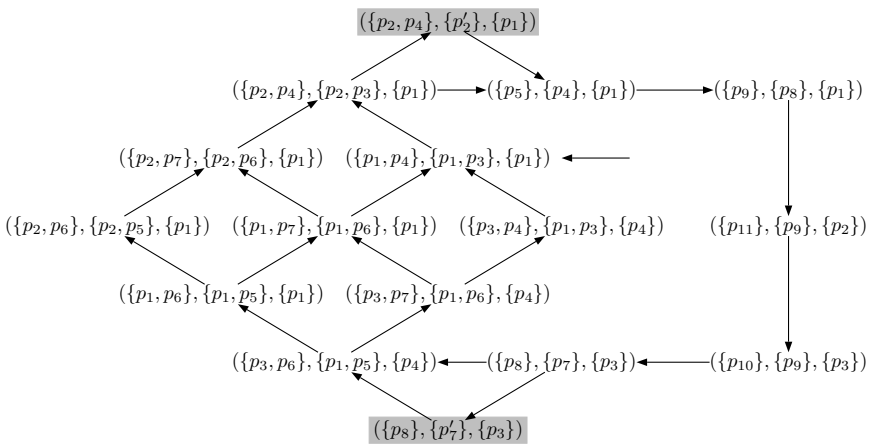


Example 5.1: VME Bus Controller: CSC solving (continued on next page)

C_1' is a correct implementation of C_1 due to the following STG bisimulation $\mathcal{B}'$. The two elements b_1', b_2' of $\mathcal{B}'$ not belonging to the identity relation over $[M_{C_1}\rangle$ are shaded.



The following STG-bisimulation $\mathcal{B}''$ for the correct decomposition (C_1', C_2) of VME, which fulfils CSC, can be derived from $\mathcal{B}$ (Example 3.9) and $\mathcal{B}'$ as described in the proof of Theorem 5.3. The elements of $\mathcal{B}''$ not belonging to $\mathcal{B}$ are shaded; they correspond to the shaded elements of $\mathcal{B}'$.



5.4 Comparison with other Approaches

In this section we will show that the decomposition method of Carmona and Cortadella [Car03, CC03], which has not been proven correct so far, yields components which are a correct decomposition according to our definition. For this method, it is assumed that an STG with CSC is given, where CSC can also be achieved by modifications on the STG-level, i.e. without considering the reachability graph. (It can also be given due to a suitable translation from a description in a high-level language to STGs as in [YOM04]). As explained on p. 100, we can assume that there are *no internal signals*.

The method of [Car03, CC03] works roughly as follows. Starting with a deterministic STG N that already has CSC, for every output signal x a *CSC support* is determined; this is a set of signals, which guarantees CSC for x. Here is the formal definition:

Definition 5.12 (CSC Support)
For an STG N, $S \subseteq Sig_N$ is called *CSC support for the output signal* x if $M_N[v_1\rangle\rangle M_1$, $M_N[v_2\rangle\rangle M_2$ for some $v_1, v_2 \in (Sig_N^{\pm})^*$ with $codeChange(S, v_1) = codeChange(S, v_2)$ implies $M_1[x^{\pm}\rangle\rangle \Leftrightarrow M_2[x^{\pm}\rangle\rangle$.

From the previous definition one can derive an integer linear programming problem (ILP) for an output x and a signal set S. The infeasibility of this problem then implies that S is a CSC support for x. Actually, the algorithm of Carmona and Cortadella uses a slightly weaker definition of CSC support, which nevertheless coincides with the given one for most practical STGs.[2] The ILP problem in [CC03] can easily be modified to match the more accurate Definition 5.12, see also [GVC97].

The algorithm starts for every output x with the set including the syntactical triggers of x and x itself, and iteratively adds signals until it is a CSC support for x, which is checked with the ILP problem mentioned above. Since the original STG has CSC, this algorithm is always successful. An advantage is therefore that this method produces components with CSC.

After that, for every output signal the original STG is *projected* onto the corresponding CSC support: the other signals are considered as dummies, and as far as possible these dummies and redundant places are removed much as in our decomposition algorithm. If the resulting component still contains dummies, then [priv. comm.]: the reachability graph is generated and viewed as a finite automaton with dummies regarded as the empty word. Now the automaton is made deterministic with well-known methods, which in particular remove all λ-labelled edges. Finally, we can regard this automaton

[2] In [CC03], S is called CSC support for the output x if for all reachable markings M_1, M_2 with $M_1[v\rangle\rangle M_2$ and $sv_{M_1}|_S = sv_{M_2}|_S$ one has: $M_1[x^{\pm}\rangle\rangle \Leftrightarrow M_2[x^{\pm}\rangle\rangle$. This definition is equivalent to ours for a reversible STG, i.e. if M_N is reachable from every reachable marking.

as an STG again, which has the edges of the automaton as transitions.

The projection part is similar to our algorithm, the difference is where backtracking is performed: the method of [Car03, CC03] uses some form of backtracking when determining the CSC support as described above — our algorithm uses backtracking when the contraction of a dummy signal is not possible.

The CSC-support algorithm produces components $(C_i)_{i\in I}$ with the following properties, which we use for the proof of Theorem 5.13. Actually, Item 2 allows components with more than one output signal, making our result stronger.

(1) Every component is deterministic

(2) The signals of every C_i are a CSC support of their output signals

(3) $\forall i \in I : L(C_i) = L(N)\downarrow_i$

In the last item, $L(N)\downarrow_i$ denotes the projection of $L(N)$ onto the signals of C_i, i.e. all signal transitions $s^\pm$ for which $s \notin Sig_i$ are removed from the words in $L(N)$. Note that this item is not equivalent to $L(||_{i\in I}C_I) = L(N)$. Now we can prove that $(C_i)_{i\in I}$ is a correct decomposition according to Definition 5.2.

Theorem 5.13 (Correctness of the CSC-support algorithm)

Let N be an STG and $(C_i)_{i\in I}$ be given as above. Then, $(C_i)_{i\in I}$ is correct w.r.t. N. The approach of [CC03] is correct.

Proof. The second claim is a corollary of the first. Let $C = ||_{i\in I}C_i$. We define a relation $\mathcal{B}$ between the markings of N and C by

$$(M, (M_i)_{i\in I}) \in \mathcal{B} \Leftrightarrow \exists w : M_N[w\rangle\rangle M \wedge \forall i \in I : M_{C_i}[w\downarrow_i\rangle\rangle M_i$$

where $(M_i)_{i\in I}$ denotes the disjoint union of the M_i, i.e. a marking of $||_{i\in I}C_i$.

We will show that $\mathcal{B}$ is an STG-bisimulation.

(1): Obviously fulfilled for $w = \lambda$.

(2): Let $(M, (M_i)_{i\in I}) \in \mathcal{B}$. Therefore $\exists w : M_N[w\rangle\rangle M \wedge \forall i \in I : M_{C_i}[w\downarrow_i\rangle\rangle M_i$. Since there are no internal signals, we do not have to consider (N3) and (C3).

(N1): Let $a \in In_N$ and $M[a^\pm\rangle\rangle\hat{M}$. This implies $wa^\pm \in L(N)$ and therefore $\forall i \in I : (wa^\pm)\downarrow_i \in L(C_i)$. If $a \notin In_C$ we are done, otherwise it follows from the determinism of the components that every C_i with $a \in In_i$ can fire $a^\pm$: there is only one transition sequence v with $l(v) = w\downarrow_i$ and one sequence v' with $l(v') = w\downarrow_i a^\pm$, obviously v is a prefix of v' and reaches M_i, and therefore $M_i[a^\pm\rangle\rangle\hat{M}_i$.

This holds for every component with $a \in In_i$ and therefore:
$(M_i)_{i\in I}[a^{\pm}\rangle\rangle(\hat{M}_i)_{i\in I}$ where $\hat{M}_i = M_i$ if $a \notin In_i$, and by definition of $\mathcal{B}$ we get $(\hat{M}, (\hat{M}_i)_{i\in I}) \in \mathcal{B}$.

(N2): Analogous to (N1), since we do not have to consider internal signals.

(C2): Let $x \in Out_j$ and $M_j[x^{\pm}\rangle\rangle$. Therefore, $w{\downarrow_j}x^{\pm} \in L(C_j) = L(N){\downarrow_j}$ which implies that there exists a $w' \in (Sig_N^{\pm})^*$ with $M_N[w'\rangle\rangle M'[x^{\pm}\rangle\rangle$ and $w{\downarrow_j} = w'{\downarrow_j} \in (Sig_j^{\pm})^*$. Since Sig_j is a CSC-support for $x \in Out_j$ and obviously $codeChange(Sig_j, w) = codeChange(Sig_j, w')$, we conclude that $M[x^{\pm}\rangle\rangle$. Applying (N2) proves the claim: $(M_i)_{i\in I}[x^{\pm}\rangle\rangle$.

(C1): Let $x \in Out_C$ and $(M_i)_{i\in I}[x^{\pm}\rangle\rangle(\hat{M}_i)_{i\in I}$.
If $x^{\pm}$ is produced by component j, we get $M_j[x^{\pm}\rangle\rangle$; then our considerations for (C2) imply $M[x^{\pm}\rangle\rangle\hat{M}$. By definition of $\mathcal{B}$: $(\hat{M}, (\hat{M}_i)_{i\in I}) \in \mathcal{B}$. □

5.5 Other Implementation Relations

As mentioned at the end of Definition 5.2, the simplest decomposition of a specification N_1 consists of only one component N_2, such that no hiding is needed.[3] We write $N_1 \lhd_{sv} N_2$ and say that N_1 is *SV-implemented* by N_2, if N_2 is correct w.r.t. N_1. Since the other two implementation relations introduced below require $In_1 = In_2$ and $Out_1 = Out_2$, we assume further that $N_1 \lhd_{sv} N_2$ only if these equations hold; this is no real restriction since missing signals can be added easily to N_2.[4]

We have already shown that $\lhd_{sv}$ is a preorder, and in this section we will compare it to two other implementation relations. Recall that we require STGs to be *deterministic*.

The second implementation relation $\lhd_{Dill}$ is based on the notion of *prefix-closed trace structures*, defined by Dill [Dil88]. Trace structures do not have internal signals, and therefore we will restrict ourselves in Theorem 5.18 to STGs for which this holds, too.

The third implementation relation $\lhd_{cc}$ is based on the notion of *I/O-compatibility* [CC03] which defines when an STG works safely together with another STG, which can be considered as environment. Since no implementation relation is defined in [CC03], we have decided to base one on I/O-compatibility in the spirit of Dill. Since I/O-compatibility requires the STGs to be *livelock-free* (see below), we restrict ourselves to STGs with this property in Theorem 5.21.

[3] If one wants to hide some signals, one can directly do so in N_2; the external hiding of Definition 5.2 was introduced for *inter*-component communication, cf. the discussion before the definition.

[4] Output signals can be added formally to Out_2; for every missing input signal a, the transitions $a+$, $a-$ resp., can be added such that one of them is always activated and consistency is preserved, e.g. as a ring with two transitions.

5.5.1 Conformance

To define and study $\lhd_{Dill}$, we first repeat the basic definitions of [Dil88].

Definition 5.14 (Prefix-Closed Trace Structures)
A *prefix-closed trace structure* is a tuple $\mathcal{T} = (In, Out, \mathcal{S}, \mathcal{F})$. In is the set of input signals, Out the set of output signals. We define $Sig := In \cup Out$. $\mathcal{S}, \mathcal{F} \subseteq (Sig^{\pm})^*$ are the sets of *success traces*, *failure traces* respectively. We define the set of possible traces $\mathcal{P} := \mathcal{S} \cup \mathcal{F}$. $\mathcal{S}$ and $\mathcal{P}$ must be prefix-closed, and the trace structure has to be *receptive*, i.e. $\mathcal{P} \cdot In^{\pm} \subseteq \mathcal{P}$. △

A trace structure[5] represents an asynchronous circuit of which the traces are partial executions. Success traces do not lead to malfunction of the circuit, while failure traces do. Obviously, prefix-closedness is a sound requirement, because a circuit has to perform all prefixes of v in order to perform v itself. Receptiveness means that every possible trace can be extended by an input signal, i.e. there are no restrictions for the environment, but such an extension can turn a success trace into a failure trace. In the original definition of Dill, traces are sequences over a set of signals instead of signal transitions, which is obviously not a real change.

There are two problems with prefix-closed trace structures. First, they can have inherent non-determinism in the form of traces $v \in \mathcal{S} \cap \mathcal{F}$, i.e. traces which could lead to malfunction or not. Second, there can exist success traces s which can be extended by a sequence w of only output signals to a failure trace $f = sw$, i.e. the malfunction of the circuit is caused by the circuit itself and cannot be avoided by the environment.

These problems can be corrected by removing traces as s and v from the success set and adding them to the failure set. The resulting trace structures are called canonical prefix-closed trace structures.

Definition 5.15 (Canonical Prefix-Closed Trace Structures)
A trace structure $\mathcal{T} = (In, Out, \mathcal{S}, \mathcal{F})$ is called *canonical* if $\mathcal{S} \cap \mathcal{F} = \emptyset$, $\mathcal{F}/Out^{\pm} \subseteq \mathcal{F}$ [6] and $\mathcal{F} \cdot Sig^{\pm} \subseteq \mathcal{F}$. △

For a canonical trace structure, the failure set is completely determined by the success set:

$$\mathcal{F} = ((\mathcal{S} \cdot In^{\pm}) - \mathcal{S}) \cdot (Sig^{\pm})^*$$

Hence, failure traces are success traces which are extended with an 'unexpected' input and possibly with additional signal edges. It is therefore sufficient to give $(In, Out, \mathcal{S})$.

[5] We will frequently omit 'prefix-closed'.

[6] Let X and Y be two sets of strings; the *quotient* X/Y is defined as $\{x \mid \exists y \in Y : xy \in X\}$. Actually, $\mathcal{F}/Out^{\pm} \subseteq \mathcal{F}$ is equivalent to $\mathcal{F}/(Out^{\pm})^* \subseteq \mathcal{F}$.

Clearly, a deterministic STG N without internal signals can be transformed into the canonical trace structure $(In, Out, L(N))$, and for the rest of this section we will identify an STG with its corresponding canonical trace structure.

Dill also defines the operations parallel composition, hiding and renaming for trace structure, which can be used together with trace structures to build expressions, e.g. $(\mathcal{T}_1||(\mathcal{T}_2 \setminus \{a, b\}))||\mathcal{T}_3$, where the $\mathcal{T}_i$ are trace structures. An *expression context* $\varepsilon[]$ is such an expression which contains one free variable which can be replaced by some trace structure $\mathcal{T}$, written $\varepsilon[\mathcal{T}]$. An expression context can be interpreted as an environment for the trace structure $\mathcal{T}$ which is inserted; $\mathcal{T}$ fits into this environment if the resulting trace structure is failure free, i.e. the environment works perfectly together with $\mathcal{T}$. This is the base for the following definition of conformance.

Definition 5.16 (Conformance, $\lhd_{Dill}$)
Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two trace structures. $\mathcal{T}_2$ conforms to $\mathcal{T}_1$, written $\mathcal{T}_1 \lhd_{Dill} \mathcal{T}_2$ if: $In_1 = In_2$, $Out_1 = Out_2$ and for every expression context $\varepsilon[]$: $\varepsilon[\mathcal{T}_1]$ is failure free implies $\varepsilon[\mathcal{T}_2]$ is failure free. □

Implementation in the sense of conformance means therefore that the implementation $\mathcal{T}_2$ works at least in every environment which is suitable for the specification $\mathcal{T}_1$.

In the sense of this implementation relation, each trace structure is equivalent to a canonical one, such that we can restrict attention to these. Finally, Dill gives a characterisation of his implementation relation for canonical trace structures, and we use this to define the implementation relation $\lhd_{Dill}$.

Definition 5.17
Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two canonical trace structures. $\mathcal{T}_1$ is *Dill-implemented* by $\mathcal{T}_2$, $\mathcal{T}_1 \lhd_{Dill} \mathcal{T}_2$, if $In_1 = In_2$, $Out_1 = Out_2$, $\mathcal{F}_2 \subseteq \mathcal{F}_1$ and $\mathcal{P}_2 \subseteq \mathcal{P}_1$. △

Since STGs without internal signals can be identified with canonical trace structures, we can apply $\lhd_{Dill}$ to such STGs.

Theorem 5.18
For deterministic STGs without internal signals we have $\lhd_{sv} \subseteq \lhd_{Dill}$.

Proof. Let N_1, N_2 be two STGs with $N_1 \lhd_{sv} N_2$. Due to the definition of $\lhd_{sv}$ we have $In_1 = In_2 =: In$ and $Out_1 = Out_2 =: Out$. Furthermore, there exists an STG-bisimulation $\mathcal{B}$ between the markings of N_1 and N_2.

(1) $\mathcal{F}_2 \subseteq \mathcal{F}_1$. Since $\mathcal{F}_i \cdot Sig_i^{\pm} \subseteq \mathcal{F}_i$, it is sufficient to show that all *minimal* (regarding prefix) failure traces of N_2 are also failure traces of N_1. Let $f \in \mathcal{F}_2$ be a minimal failure trace of N_2. Clearly, $f = sa^{\pm}$ with $a \in In$ and $s \in \mathcal{S}_2$.

If $s \in \mathcal{F}_1$, so is f and we are done. If otherwise $s \in \mathcal{S}_1$, the existence of $\mathcal{B}$ implies by (N1) and (N2) that $M_{N_1}[s\rangle\rangle M_1$ and $M_{N_2}[s\rangle\rangle M_2$ (observe that there are no internal signals) with $(M_1, M_2) \in \mathcal{B}$. $M_1[a^{\pm}\rangle\rangle$ would imply by (N1) $M_2[a^{\pm}\rangle\rangle$ and $f \in \mathcal{S}_2$, therefore $\neg M_1[a^{\pm}\rangle\rangle$ and $f \in \mathcal{F}_1$.

If $s \notin \mathcal{S}_2$, let $s = s'b^{\pm}s''$ with $b \in Sig$ and s' the longest prefix of s with $s' \in \mathcal{S}_1$. As above, the existence of $\mathcal{B}$ implies that $M_{N_1}[s'\rangle\rangle M_1$ and $M_{N_2}[s'\rangle\rangle M_2$. Since $s'b^{\pm} \in \mathcal{S}_2$, $M_2[b^{\pm}\rangle\rangle$. Therefore, b cannot be an output signal, because (C1) would imply that $M_1[b^{\pm}\rangle\rangle$. Hence, $b \in In$ and $s'b^{\pm} \in \mathcal{F}_1$ implying $f \in \mathcal{F}_1$.

(2) $\mathcal{P}_2 \subseteq \mathcal{P}_1$. Let $p \in \mathcal{P}_2$ be a trace of N_2. If $p \in \mathcal{F}_2$, using (1) we get $p \in \mathcal{F}_1 \subseteq \mathcal{P}_1$. So assume $p \in \mathcal{S}_2$. If $p \in \mathcal{S}_1$ we are done, if not let $p = p'a^{\pm}p''$ with p' the longest prefix of p with $p' \in \mathcal{S}_1$. As above, $p' \in \mathcal{S}_2$ and (C1) implies $a \in In$ and therefore $p'a^{\pm} \in \mathcal{F}_1$ and $p \in \mathcal{F}_1 \subseteq \mathcal{P}_1$. □

5.5.2 I/O-Compatibility

We come now to the comparison of $\lhd_{sv}$ and $\lhd_{cc}$. As mentioned above, we define $\lhd_{cc}$ in the spirit of Dill (cf. Definition 5.16) where now a systems works well in an environment when both are I/O-compatible, defined as follows.

Definition 5.19 (Livelock-Free and I/O-Compatibility)

(1) An STG is *livelock-free* if there is no reachable marking which enables an infinite sequence of internal edges.

(2) Let N_1 and N_2 be two livelock-free STGs with $In_1 = Out_2$, $In_2 = Out_1$ and $Int_1 \cap Int_2 = \emptyset$. N_1 and N_2 are *I/O-compatible*, denoted $N_1 \leftrightharpoons N_2$, if there is a relation $\mathcal{R}$ between the markings of N_1 and N_2 such that $(i = 1, 2)$:

- (IO1) $(M_{N_1}, M_{N_2}) \in \mathcal{R}$
- (IO2) Receptiveness
  - (a) If $(M_1, M_2) \in \mathcal{R}$ and $M_1[x^{\pm}\rangle\rangle M_1'$ with $x \in Out_1$, then $M_2[x^{\pm}\rangle\rangle M_2'$ with $(M_1', M_2') \in \mathcal{R}$.
  - (b) vice versa for M_2
- (IO3) Internal Progress
  - (a) If $(M_1, M_2) \in \mathcal{R}$ and $M_1[u^{\pm}\rangle\rangle M_1'$ with $u \in Int_1$, then $(M_1', M_2) \in \mathcal{R}$.
  - (b) vice versa for M_2
- (IO4) Deadlock-freeness

(a) If $(M_1, M_2) \in \mathcal{R}$ and $\{a \in Sig_1 \mid M_1[a^{\pm}\rangle\rangle\} \subseteq In_1$, then $\{a \in In_2 \mid M_2[a^{\pm}\rangle\rangle\} \not\subseteq In_2$.

(b) vice versa for M_2 △

The idea is that the two circuits described by N_1 and N_2 are working together without failures or deadlocks, one producing the signals which are received by the other one as inputs. (IO2) requires that outputs must be matched immediately – without preceding internal signals – by the other circuit, and (IO3) means that the components can produce internal signals unobserved by the other. (IO4) forbids deadlocks, i.e. at least one circuit must activate an output or internal signal, and because they are livelock-free, eventually an output must be produced.

Definition 5.20 (of $\lhd_{cc}$)
Let N_1 and N_2 be livelock-free deterministic STGs. We write $N_1 \lhd_{cc} N_2$ if for all livelock-free and deterministic STGs N, $N_1 \leftrightharpoons N$ implies $N_2 \leftrightharpoons N$.

Theorem 5.21
For deterministic livelock-free STGs: $\lhd_{sv} \subseteq \lhd_{cc}$.

Proof. Let $N_1 \lhd_{sv} N_2$ due to STG-bisimulation $\mathcal{B}$ and let N be an arbitrary STG with $N \leftrightharpoons N_1$ due to relation $\mathcal{R}$. We will show that $N \leftrightharpoons N_2$ due to relation $\mathcal{R}' = \mathcal{R} \circ \mathcal{B}$. In the following $(M, M_2) \in \mathcal{R}'$ implies therefore the existence of some marking M_1 of N_1 with $(M, M_1) \in \mathcal{R}$ and $(M_1, M_2) \in \mathcal{B}$.

(IO1) Obviously, $(M_N, M_{N_2}) \in \mathcal{R}'$.

(IO2) Receptiveness

(a) Let $(M, M_2) \in \mathcal{R}'$ and $M[x^{\pm}\rangle\rangle\hat{M}$ for $x \in Out_N$. This implies $x \in In_{N_1}$ and $M_1[x^{\pm}\rangle\rangle\hat{M}_1$ with $(\hat{M}, \hat{M}_1) \in \mathcal{R}$; furthermore (N1) for $(\mathcal{B})$ implies $M_2[x^{\pm}\rangle\rangle\hat{M}_2$ with $(\hat{M}_1, \hat{M}_2) \in \mathcal{B}$. By definition of $\mathcal{R}'$, we also have $(\hat{M}, \hat{M}_2) \in \mathcal{R}'$.

(b) Let $(M, M_2) \in \mathcal{R}'$ and $M_2[x^{\pm}\rangle\rangle\hat{M}_2$ for $x \in Out_{N_2}$. This implies by (C1) for $\mathcal{B}$: $M_1[v\rangle\rangle M_1'[x^{\pm}\rangle\rangle\hat{M}_1$, $v \in (Int_{N_1}^{\pm})^*$ and $(\hat{M}_1, \hat{M}_2) \in \mathcal{B}$. By (IO3).(b) of $\mathcal{R}$ we know that $(M, M_1') \in \mathcal{R}$, and by (IO2)(b) for $\mathcal{R}$ we get $M[x^{\pm}\rangle\rangle\hat{M}$ and $(\hat{M}, \hat{M}_1) \in \mathcal{R}$; hence $(\hat{M}, \hat{M}_2) \in \mathcal{R}'$.

(IO3) Internal Progress

(a) Let $(M, M_2) \in \mathcal{R}'$ and $M[u^{\pm}\rangle\rangle\hat{M}$ for $u \in Int_N$. Then, we get immediately $(\hat{M}, M_1) \in \mathcal{R}$ and $(\hat{M}, M_2) \in \mathcal{R}'$.

(b) Let $(M, M_2) \in \mathcal{R}'$ and $M_2[u^{\pm}\rangle\rangle\hat{M}_2$ for $u \in Int_{N_2}$. By (C3) for $\mathcal{B}$ we get $M_1[v\rangle\rangle\hat{M}_1$, $(\hat{M}_1, \hat{M}_2) \in \mathcal{B}$ and $v \in (Int^{\pm}_{N_1})^*$. Then, repeated application of (IO3)(b) for $\mathcal{R}$ implies $(M, \hat{M}_1) \in \mathcal{R}$ and $(M, \hat{M}_2) \in \mathcal{R}'$.

(IO4) Deadlock-freeness

(a) Let $(M, M_2) \in \mathcal{R}'$ and $\{a \mid M[a^{\pm}\rangle\rangle\} \subseteq In_N$. If $M_2[u^{\pm}\rangle\rangle$ for some $u \in Int_{N_2}$ we are done. So assume this is not the case.
By (IO4)(a) for $\mathcal{R}$, we have that $\{s \mid M_1[s^{\pm}\rangle\rangle\} \not\subseteq In_{N_1}$. Let $v \in (Int^{\pm}_{N_1})^*$ be a maximal sequence (w.r.t. prefix) such that $M_1[v\rangle\rangle\hat{M}_1$; v exists because N_1 is livelock-free. (IO3)(b) for $\mathcal{R}$ implies $(M, \hat{M}_1) \in \mathcal{R}$ and (N3) for $\mathcal{B}$ implies $(\hat{M}_1, M_2) \in \mathcal{B}$ by assumption. As above (IO4)(a) implies $\{s \mid \hat{M}_1[s^{\pm}\rangle\rangle\} \not\subseteq In_{N_1}$; since v is maximal, there exists a signal $x \in Out_{N_1}$ with $\hat{M}_1[x^{\pm}\rangle\rangle$ and by (N2) and assumption we get: $M_2[x^{\pm}\rangle\rangle$.

(b) Let $(M, M_2) \in \mathcal{R}'$ and $\{a \mid M_2[a^{\pm}\rangle\rangle\} \subseteq In_{N_2}$ $(*)$.
This implies that no output signal is enabled at M_1, since otherwise a local signal would be enabled at M_2. As in the previous item, let $v \in (Int^{\pm}_{N_1})^*$ be a maximal sequence (w.r.t. prefix) such that $M_1[v\rangle\rangle\hat{M}_1$.
Then $(*)$ and (N3) for $\mathcal{B}$ imply $(\hat{M}_1, M_2) \in \mathcal{B}$, and (IO3)(b) for $\mathcal{R}$ implies $(M, \hat{M}_1) \in \mathcal{R}$. As above, in $\hat{M}_1$ no output signal is enabled and since v is maximal, no internal signal either, i.e. $\{a \mid \hat{M}_1[a^{\pm}\rangle\rangle\} \subseteq In_{N_1}$ and due to (IO4).(b): $\{a \mid M[a^{\pm}\rangle\rangle\} \not\subseteq In_N$. □

5.5.3 Strictness of Inclusions

Finally, we prove prove that the inclusions of Theorem 5.18 and 5.21 are strict.

Theorem 5.22
For deterministic STGs without internal signals (which are therefore livelock-free) we have: (1) $\lhd_{cc} \not\subseteq \lhd_{sv}$ (2) $\lhd_{Dill} \not\subseteq \lhd_{sv}$.

Proof. Consider the counterexample in Figure 5.2. To keep it simple, we do not consider signal edges, which makes no difference for the purpose of this proof.

(1) Obviously, $\neg(N_1 \lhd_{sv} N_2)$, because in the initial marking N_1 activates the output y which is not activated in N_2, violating Condition (N2).

On the other hand, $N_1 \lhd_{cc} N_2$: consider an STG N with $N \leftrightharpoons N_1$. In the initial state, N must be ready to receive the signals x and y as inputs due to condition (IO2). After x (y resp.) occurred, N_1 only activates the input a (b resp.) and (IO4) implies

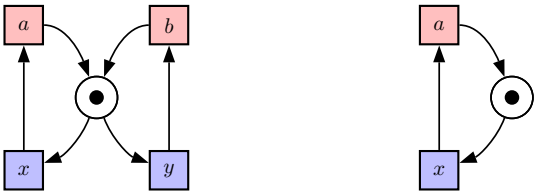


Figure 5.2: Counterexample for the proof of Theorem 5.22.

that N has to activate an output. This output can only be a (b resp.) because any other signal would violate condition (IO2). Therefore, the reachability graph of N is bisimilar to the one of N_1. It is therefore sufficient to show $\overline{N_1} \leftrightharpoons N_2$, where $\overline{N_1}$ denotes the *mirror* of N_1, i.e. the STG which is exactly the same as N_1 but inputs and outputs are exchanged.

In the initial state, N_2 only activates the output x which is expected by $\overline{N_1}$. Then $\overline{N_1}$ activates the output a which is expected by N_2. Observe that it is not relevant that $\overline{N_1}$ activates the input y in the initial state.

(2) The *Universal Do-Nothing Module* $\mathcal{U}$ is a system which accepts all inputs and produces no output. The corresponding canonical trace structure $\mathcal{U}$ over (In, Out) is given by $\mathcal{S}_\mathcal{U} = (In^\pm)^*$ for arbitrary In and Out. Clearly, $\neg N_2 \lhd_{sv} \mathcal{U}$. On the other hand, for any STG N it is valid that $N \lhd_{Dill} \mathcal{U}$ ([Dil88]): by Definition 5.15 we get $\mathcal{F}_\mathcal{U} = ((\mathcal{S}_\mathcal{U} \cdot In^\pm) - \mathcal{S}_\mathcal{U}) \cdot (Sig^\pm)^* = \emptyset \cdot (Sig^\pm)^* = \emptyset \subseteq \mathcal{F}_N$. Consider now a trace $p \in \mathcal{P}_\mathcal{U} = (In^\pm)^*$. If $p \in \mathcal{S}_N$, we are done. Otherwise, let $p = p' a^\pm p''$, such that p' is the longest prefix of p with $p' \in \mathcal{S}_N$. Since a is an input, $p' a^\pm$ and therefore also p are failure traces. □

For (1), it is essential that N_1 has a dynamic conflict between outputs, while for (2), it is essential that Dill requires an implementation to produce only *allowed* outputs, but does not prescribe to produce *all* of them.

Chapter 6

Advanced Decomposition Strategies

As it was mentioned in Chapter 3, the decomposition algorithm is non-deterministic and does not make any further specifications for an implementation. Hence, a technically simple implementation can be easily achieved, but the performance might not be so good.

In this chapter, we will investigate how the original decomposition algorithm – called Basic here – can be modified such that the efficiency is improved. This modifications are partially on the level of heuristics, i.e. in most – but not all – cases they will improve the efficiency, but in any case the results are correct.

An issue of non-deterministic algorithms is whether they are determinate; this question was answered positively for marked graphs in Chapter 4. Another issue is whether the worst-case performance of a non-deterministic algorithm can be improved by (partially) determinising it. The first section is devoted to this last question; there, a simple but efficient heuristic (Reordering) to speed up reduction is introduced.

In Section 6.3 an optimised backtracking algorithm (LazyBack) is presented which avoids the restarting of a reduction at the very beginning.

Decomposition reduces each component separately although for 'similar' components similar steps have to be performed. The third section shows how the reduction of all component can be performed together in order to exploit such similarities (Tree and Aggregation).

In Section 6.5, two improvements independent of these new strategies are presented.

The first decreases the runtimes of all strategies significantly, and the second has positive effects on the synthesisability of the components.

Another method of alleviating state space explosion are Petri net analysis techniques based on causal partial order semantics, in the form of Petri net unfoldings and their complete prefixes.[1]

The papers [KKY04, KKY06, Kho07] present a complete and efficient design flow for complex-gate logic synthesis based on Petri net unfoldings, which avoids generating the state graph at any stage. This unfolding-based approach can often synthesise specifications which are by orders of magnitude larger than those which can be synthesised by the state-space based techniques. However, this is still not enough for practical circuits. On the other hand, decomposition can introduce encoding conflicts in the components, thus causing problems for subsequent logic synthesis.

As the second contribution of this chapter, it is shown in Section 6.6 how to combine the decomposition and the unfolding approach.

6.1 Correctness of New Strategies

Since the correctness of the decomposition algorithm was proven for the non-deterministic version, the correctness proofs for the new strategies are quite simple: one only has to show that the result could have been generated by Basic, too. Furthermore the following lemma is needed.

Lemma 6.1
If the reduction operations are applied to some initial component, and this gives an STG C without structural auto-conflicts, then no structural auto-conflict encountered during this partial reduction is dynamic, and hence the reduction is correct so far.

Proof. The reduction operations are *auto-cc-preserving* [VK06], i.e. a reduction operation turns an STG with auto-concurrency or a dynamic auto-conflict into an STG which also has one of these properties.

Assume now that some intermediate STG C' has a dynamic auto-conflict. Then, auto-cc-preservation implies that C has to contain an auto-conflict or auto-concurrency. The former is impossible since C has no structural auto-conflicts; the latter implies that C is not consistent, which is also impossible, since consistency is preserved during reduction. □

[1] Confer the discussion on p. 40 why unfoldings are very suitable for the synthesis of asynchronous circuits.

In fact, auto-cc-preservation implies for consistent STGs that dynamic auto-conflicts are preserved.

6.2 Reordering Transition Contractions

Although reduction is meant to be performed automatically, it can be done with pen and paper. To keep this simple, one would contract those transitions first which generate the smallest number of new places. In the optimal case a divining transition has only one place in its pre- and postset, thus its contraction would generate one new place while removing both old ones. But the contraction of a transition, with for instance 4 places in its pre- and 6 places in its postset would increase the number of places by 14. These 14 places may be adjacent to other divining transitions and so on. Hence, contracting transitions in an unsuitable order can lead to an enormous increase in the number of places.

Contracting 'easy' transitions first turned out to be a good heuristic also for the automatic reduction of large STGs. In Reordering, the divining transitions are sorted by the number of additional places their contraction would generate in the initial component. Then reduction works as in Basic, following this precalculated list of transition contractions. In order to avoid repeated calculation after every reduction operation, this list is not updated during reduction; the benchmark results are good nevertheless.

Obviously, Reordering is correct in the sense of Lemma 6.1, because the chosen order of contractions is just a concrete instance of an arbitrary one.

6.3 Lazy Backtracking

In the original implementation, backtracking was performed by discarding all the operations performed so far and restarting the reduction for a modified initial component with more inputs, cf. also Figure 6.1. This way of backtracking plays an important part in the correctness proof in [VW02, VK06]. But it can obviously be rather inefficient, e.g. in extreme cases backtracking might occur for the last divining transition and result in repeating a large number of operations. To guarantee correctness of this approach, the deletion of redundant transitions has to be *restricted*: the deletion of a λ-labelled duplicate transition is only performed if the former labels of the respective transitions are equal. If this is not the case, the specification is in general ill-formed anyway, but this error is not visible in the components. Correspondingly, if the specification is guaranteed to be 1-live and consistent, this check can be omitted (see below).

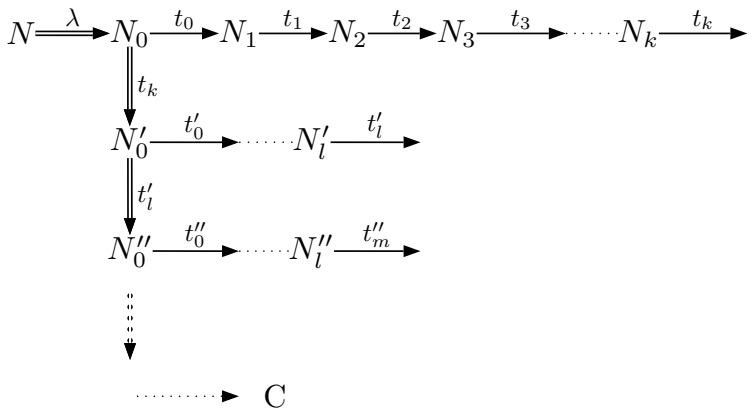


Figure 6.1: Ordinary Backtracking: in the specification N some signals are lambdarised ($\stackrel{\lambda}{\Rightarrow}$), then some transition contractions are performed ($\stackrel{t_0}{\rightarrow}$). If a transition t_k cannot be contracted the corresponding signal is delambdarised in N_0 ($\stackrel{t_k}{\Rightarrow}$) resulting in a new initial component N_0' and the reduction starts anew.

Naturally, if the reduction should not start anew from the beginning, one has to introduce *savepoints* for intermediate STGs. Since backtracking affects signals rather than single transitions, *lazy backtracking* (LazyBack) contracts all transitions of signal s_0, then all transitions of signal s_1 and so on. After a signal was successfully contracted (i.e. all corresponding transitions), the resulting intermediate STG is used as a savepoint.

If backtracking has to be performed, it is unnecessary now to start from the very beginning. Instead, it is possible to use the last suitable savepoint. While this basic idea is simple, there is a complication to consider.

Starting from N, all initially useless signals are lambdarised yielding the initial component N_0. Instead of contracting the corresponding transitions in an arbitrary order as in Basic, the divining transitions are contracted grouped by their former signals as described above and depicted in Figure 6.2.

If contracting all s_0-transitions is possible, i.e. all contractions are secure and no new structural auto-conflict is generated, save the resulting STG as N_1. Next, try to contract signal s_1 in N_1 and so on. This results in a sequence (N_i) of savepoints and a sequence (s_i) of contracted signals. If every signal contraction is possible, LazyBack is obviously correct.

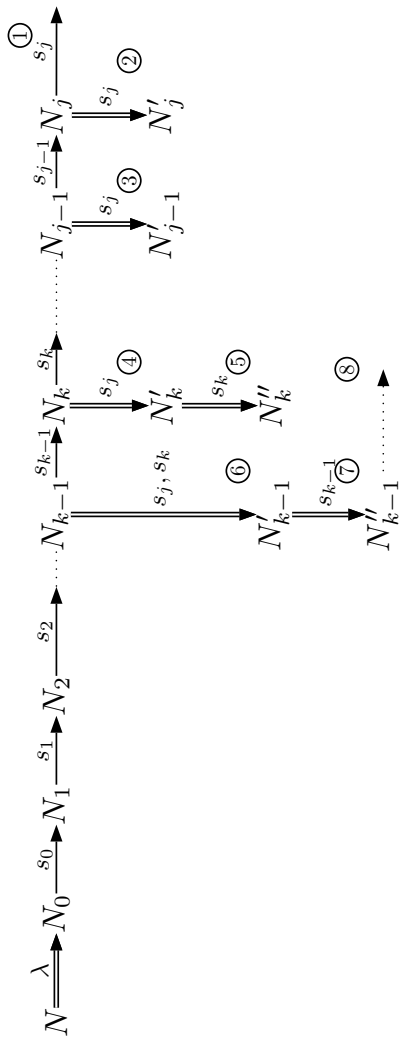


Figure 6.2: Backtracking of LazyMulti. Conventions as in Figure 6.1, except that $\stackrel{s_i}{\rightarrow}$ denotes the contraction of signal.

Assume now that backtracking has to be performed since the contraction of signal s_j is not possible in N_j (1). In BASIC, one would delambdarise s_j in N_0 and start anew from there. Instead, we delambdarise s_j in N_j resulting in N'_j (2); the critical point is that we have to check for a structural auto-conflict of s_j in N'_j now. (Such a conflict might exist, because conflicts between divining transitions are ignored during reduction.)

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

Proposition 6.2
Let N'_j be obtained from an intermediate savepoint N_j by delambdarising some set of signals S' not containing $s_0, \ldots, s_{j-1}$, and let N'_0 be obtained from N_0 by delambdarising S', too. If N'_j does not have any structural auto-conflicts, N'_j can be constructed during a correct reduction for the initial component N'_0.

Proof. We will argue inductively that actually the same operation sequence which reached N_j can be performed reaching N'_j, at least if we ignore structural auto-conflicts for the time being; during this, every original intermediate STG $\hat{N}$ is matched with some new intermediate STG $\hat{N}'$ obtained from $\hat{N}$ by delambdarising S'.

We clearly have this match before the first operation, i.e. for N_0 and N'_0. Assume we have reached some $\hat{N}'$ matching $\hat{N}$ in the original sequence.

If the operation applied to $\hat{N}$ is an implicit place deletion, then this can also be applied to $\hat{N}'$ with a matching result, since place redundancy does not depend on the labelling. If the operation is the contraction of transition t, we first note that the former signal of t is in $\{s_0, \ldots, s_{j-1}\}$ and that thus t is also a λ-transition in $\hat{N}'$. Secondly, the contraction is still secure since this is independent of the labelling. Hence, the contraction can be applied with a matching result (maybe generating a new structural auto-conflict).

If the operation applied to $\hat{N}$ is the deletion of a redundant transition t, there are several cases:

(1) t is not labelled with λ but a duplicate of some other transition with the same label. Then, it is also a duplicate of this transition in $\hat{N}'$ and can be deleted.

(2) t is labelled with λ in $\hat{N}$ and was removed because its former signal is in $\{s_0, \ldots, s_{j-1}\}$. In this case (same argumentation as above) it is a λ-transition in $\hat{N}'$ as well. We consider two cases:

(a) t is a loop-only transition in $\hat{N}$ and therefore also in $\hat{N}'$ and can be deleted.

(b) t is a duplicate of another λ-transition t' in $\hat{N}$. If $l(t') \notin S'^{\pm}$, t' is still a λ-transition in $\hat{N}'$ and t can be deleted.

But if $l(t') = s^{\pm} \in S'^{\pm}$, t is no longer a duplicate of t', but a *structural duplicate.*, i.e. the structural preconditions are fulfilled but the labels differ. This case will not happen if – as mentioned at the beginning of this section – a duplicate λ-transitions is only deleted if the respective former labels coincide.

Otherwise, the specification and hence N'_0 are guaranteed to be consistent and 1-live. Furthermore, the reduction operations preserve consistency and 1-liveness. Thus, in $\hat{N}'$, t and t' are 1-live and there is a reachable marking M which enables both with $M[t\rangle M'$ and $M[t'\rangle M'$. If t fires, the state vector of M and M' are equal, but if t' fires, they differ for signal s; this contradicts our notion of consistency.

This finishes the inductive proof – to conclude the main proof, we simply point out that, if the reduction of N'_0 is possible, the operation sequence reaching N'_j is correct according to Lemma 6.1. In particular, the newly generated structural conflicts in $N'_0 \dots N'_{j-1}$ are not dynamic. □

The last argument of the proof also shows that LazyBack can additionally help to keep the components small, because some unnecessary backtracking due to structural-but-not-dynamic auto-conflicts might be avoided.

We proceed with the description of LazyBack. We look now at the case where there is at least one structural auto-conflict for signal s_j in N'_j. Then we cannot proceed from this savepoint; instead, we have to find the signals whose contraction caused these conflicts. To do this, consider STG N_{j-1} with s_j delambdarised resulting in N'_{j-1} (3). If there is no conflict for s_j, it is clear that the conflicts were generated by the contraction of s_{j-1}.

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

If eventually a savepoint N_k is reached where the respective N'_k does not have a structural auto-conflict for s_j (4), it is clear that the contraction of signal s_k caused at least some conflict of s_j, which is visible in N'_{k+1}. Therefore, s_k has to be delambdarised in N'_k, too, resulting in N''_k (5).

At this point there are two possible substrategies:

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

- LazyMulti: If there is no structural auto-conflict for s_k in N''_k, proceed from there; differing from LazySingle, s_j is not lambdarised again. If there is a structural auto-conflict for s_k in N''_k, go back to savepoint N_{k-1}, delambdarise s_k *and* s_j (6) and check for structural auto-conflicts, and so on. In general, go back with *all signals delambdarised so far* until a suitable savepoint is found.

  In Figure 6.2, in N'_{k-1} the delambdarisation created a structural auto-conflict for signal s_{k-1} which is therefore also delambdarised, yielding STG N''_{k-1} (7). Now there are no more structural conflicts, and the reduction proceeds from there (8).

- LazySingle: If there is no structural auto-conflict for s_k in N''_k, lambdarise s_j in N''_k again and proceed from there with reduction. If there is a structural auto-conflict for s_k in N''_k, go back to savepoint N_{k-1}, delambdarise s_k (but not s_j) and check for structural auto-conflicts, and so on. In general, go back with the *last delambdarised signal* until a suitable savepoint is found.

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

Both strategies are correct due to Proposition 6.2. LazySingle is the more optimistic strategy: the hope is that preventing the 'initial' structural auto-conflict by making *one* signal visible might also prevent the resulting conflicts including the one for s_j. Backtracking in LazyMulti mimics Basic, which would restart with s_j delambdarised after a structural auto-conflict was encountered, then restart with also s_k delambdarised, and so on.

The implementation of LazyBack in DesiJ differs slightly from the above description: if during backtracking a savepoint is found from which reduction can proceed, the corresponding signals are not delambdarised in the preceding savepoints in order to improve runtime. Instead, all signals which have been chosen for delambdarisation by LazyBack are stored separately, and they are delambdarised in the components on-the-fly when going back. It is in fact not hard to see that this is also correct due to Proposition 6.2.

6.4 Tree Decomposition

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

Considering example decompositions, it turned out that in most cases some components had many lambdarised signals in common. Therefore, there should be an intermediate STG C', from which these components could be derived: instead of reducing both components independently, it is sufficient to generate C' only once and to proceed separately with each component afterwards, thus saving a lot of work. If this principle is iteratively applied, it results in a *decomposition tree* which combines intermediate results to reduce the overall number of reduction operations. As in LazyBack, Tree performs the contractions signal-wise.

Tree itself has two phases: building the decomposition tree and actually performing *tree decomposition*. We demonstrate these phases by means of an example (see Figure 6.3): let N be an STG with the signal set $\{1, 2, 3, 4, 5\}$. Furthermore, let there be 3 components C_1, C_2, C_3, and $\{1, 2, 3\}$, $\{2, 3, 4\}$, $\{3, 4, 5\}$ be the signals which are lambdarised initially in these components. A possible intermediate STG C' for C_1 and C_2 would be the STG in which signals 2 and 3 have been contracted.

In (a) the initial situation is depicted. There are three independent leaves labelled with the signals which should be contracted to get a component. In (b) C' is introduced as a common intermediate result of C_1 and C_2. In (c) one can see nearly the same situation as in (b), but signals which were already contracted earlier are commented out; they are shown in brackets, and the actual labels of the respective leaves are $\{1\}$ and $\{4\}$. The following is a more operational view: each node u is labelled with the signals s_u which should be contracted when it is entered with some STG, see below. In (d) we added a common intermediate result for C' and C_3 with label $\{3\}$, yielding the final decomposition tree. In (i) there is another possible tree for the same components.

Tree decomposition according to a given decomposition tree works as follows: enter the root node with the initial STG N without lambdarised signals. If entering a node u with an STG N_u, lambdarise the signals s_u in N_u, perform reduction as usual and enter each child node with its own copy of the resulting STG. If u is a leaf, the resulting STG is a final component.

From this use of a decomposition tree, it is clear that in an *optimal* decomposition tree the sum of all $|s_u|$ should be minimal. Because of this, a decomposition tree is the same as a *preset tree* defined in [KK01]. There it is shown that finding an optimal preset tree is NP-complete, and a heuristic bottom-up algorithm is described which performs reasonably well and works roughly as in the example above. We use this algorithm for the automatic calculation of decomposition trees.

But there is a twist in our setting: since this tree is precalculated from the initial feasible partition, it is likely that not all signal contractions are possible. If signal $s \in s_u$ cannot be contracted in N_u, we have to backtrack and s will be visible in the

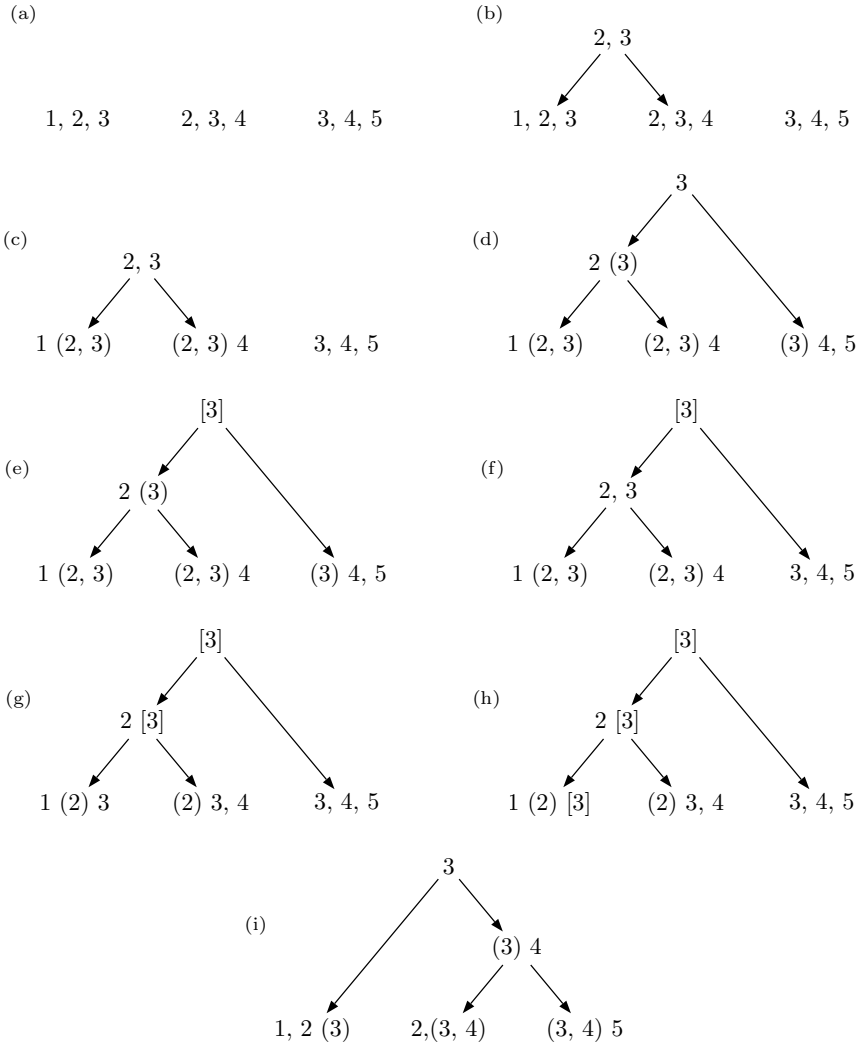


Figure 6.3: Tree Decomposition: the numbers denote signals, (1) denotes that signal 1 is removed from a node during the construction, [1] denotes that signal 1 cannot be contracted and is postponed.

intermediate component resulting from N_u; now the easiest thing would be to leave the tree as it is (and making s visible for the whole subtree of u).

But there is a way to obtain better results: we *postpone* s, i.e. we add s to every child node of u (if there are any). This is promising for the following reason: the contraction of s may have caused a structural auto-conflict for a signal s', which is lambdarised deeper in this subtree. When s' is eventually contracted, the contraction of s may become possible, making at least some of the final components smaller.

In our example, assume that the contraction of signal 3 in the root node is not possible, because its contraction causes a conflict for signal 4, see Figure 6.3(e). Signal 3 is therefore added to the inner node and the rightmost leaf in (f). In the rightmost leaf the contraction of signal 3 becomes eventually possible after the contraction of signal 4, but not in the inner node so that 3 is added to the leftmost and middle leaves (g). In the first one the contraction is again not possible, but in the latter one it finally is (h). Therefore, the components C_2 and C_3 were generated as prearranged, only component C_1 has the additional signal 3, considered as an additional input.

Observe that – in contrast to LazyBack– once the decomposition of a node is finished, it is not necessary to come back to this node and to delambdarise additional signals. Since signals are lambdarised just in time when entering a node, there are no λ-transitions left after the reduction in a node is finished and every potential auto-conflict has become visible.

Backtracking with or without postponing changes the precalculated decomposition tree, possibly decreasing its quality. In future work we will study how to interleave the generation of the decomposition tree with the calculation of the components: in [KK01] also a top-down algorithm for generating preset-trees is given, which starts at the root node of the tree. After this algorithm has calculated the root node, Tree is applied just to this node and maybe postpones some signals, thus changing the root node. Then the children of this modified node are calculated (by the preset tree algorithm) and reduced by Tree and so on. With this method postponing is taken into account and the resulting tree might be better than for the current approach.

We now argue why Tree is a correct strategy. Consider the final decomposition tree resulting from postponing some signals, cf. Figure 6.3(h); consider a final component C (possibly having additional input signals due to postponing) and the path from the root to the corresponding leaf. The reduction operations on this path can be performed in the same order without backtracking by Basic. The only difference is that in Tree signals are lambdarised 'just in time' and not at the beginning. This could only result in more structural auto-conflicts, but Tree does not encounter any, which implies correctness by Lemma 6.1.

6.4.1 Component Aggregation

An open problem of decomposition is how to find a good partition of the output signals of an STG N, i.e. one that would result in components which are small enough to be synthesised and which at the same time would yield small and efficient circuits. A natural partition is of course the finest partition (cf. Section 3.3), whose members usually contain only one output signal. However, this finest partition is not always the best one, since it may be difficult to resolve CSC conflicts in it, and even when this is not the case, having more output signals in a component can simplify the circuit. Hence, it often makes sense to merge a number of small components (or components sharing many signals) into one, provided that the resulting STG is not too big for the synthesis tool to handle. Using Tree, this can be accomplished by *aggregating subtrees* of a decomposition tree. We perform Tree as described above for the finest partition, but with one difference: after a node u is reduced yielding STG C, we check if we should stop at this point. If so, instead of generating all the components for the leaves of u in the original decomposition tree, only C is returned, which produces all the output signals of the leaves of this subtree.

By this method we get a reduced decomposition tree corresponding to a new coarser partition, and clearly the correctness of Aggregation follows from the correctness of Tree. Observe that the reduced decomposition tree corresponds to a feasible partition, since all other feasible partitions can be derived from the finest one as described on page 59.

It remains to explain under which conditions a subtree u should be aggregated. Since the main purpose of decomposition is to make the components small enough for the synthesis tool to handle, we propose two sensible criteria:

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

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

6.5 Undo Stack and Self-Triggering

During decomposition, one often has to copy large STGs as savepoints for backtracking; this actually has quite an influence on runtimes. Also, some of the described strategies store quite a number of copies, which increases the memory usage. The following simple implementation idea considerably improves the efficiency: during reduction, one keeps an *undo stack* of the operations performed; instead of copying a previous STG, one recomputes it by undoing the reduction operations as needed, see Chapter 8 for a detailed explanation. In the result section, we show experimentally that this idea can decrease the runtimes of decomposition by factor 12, depending on the size of the specification.

The second idea concerns a specific issue of circuit synthesis: for synthesis, the STG must have CSC. Unfortunately, decomposition might easily introduce new CSC conflicts. A special type of CSC conflict is *self-triggering*, which means that, for some reachable marking M, we have $M[t\rangle M'[t'\rangle M''$ such that $\neg M[t'\rangle$ and t and t' are labelled with complementary edges of the same input signal and M and M'' enable different outputs. CSC conflicts have to be resolved by insertion of internal signals, but the ones resulting from self-triggering of input transitions are irreducible (in the speed-independent framework), thus self-triggering should be avoided.

However, checking for self-triggering requires a reachability analysis. The implementation of decomposition uses a conservative structural overapproximation: *structural self-triggering* means that there are transitions t and t' which are labelled with complementary edges of the same input signal and satisfy $t \in {}^\bullet({}^\bullet t')$. Analogously to new structural auto-conflicts, the algorithm checks whether a contraction would create a new structural self-triggering; such a contraction is not allowed and leads to backtracking. This approach also gives correct decompositions, since backtracking is allowed in Basic at any stage. In Section 6.7, we will show that this simple structural approach has a very beneficial effect on the synthesisability of the components.

6.6 CSC-Aware Decomposition

Often, one can assume that the original STG has CSC, e.g. if it was derived from a Balsa specifications. Unfortunately, decomposition can introduce new CSC conflicts; in such cases, it is preferable to delambdarise more signals than necessary for correctness, in order to preserve CSC rather than to resolve CSC conflicts with new internal signals.

In the previous section, it was discussed that already avoiding self-triggering can help to preserve CSC, but often this is not enough. In this section, it is described

how STG decomposition can be combined with unfolding-based STG synthesis – in particular with the tool MPSAT – to generate components which preserve CSC of the specification.

The strategy we adopted is as follows. A variation of TREE (see below) is used to detect signals which help to preserve CSC, and while the STGs are large, only structural conservative checks are made, as it may be computationally very expensive to perform the exact tests. After some reductions have been performed, it becomes feasible to check exact reachability-like properties using PUNF and MPSAT (logic synthesis is still not feasible at this stage). Eventually, when the components are small enough, logic synthesis is performed.

While DESIJ can handle and produce non-safe nets, PUNF and MPSAT need safe nets. Therefore, we accept only safe nets as specifications (which is no serious restriction) and perform only *safeness-preserving* contractions (Subsection 6.6.2) during decomposition.

During the decomposition process the decomposition algorithm checks from time to time the following reachability-like properties:

- The decomposition algorithm should backtrack if a new dynamic auto-conflict is produced. The corresponding conservative test is the presence of a new structural auto-conflict.
- It is also helpful to remove implicit places. The corresponding conservative test looks for redundant places, which are defined by a system of linear inequalities. Checking this condition with a linear program solver is not NP-complete but still quite expensive, and therefore DESIJ looks only for the subset of shortcut places, cf. Section 3.1.2.
- In order to apply MPSAT, the STG must be safe. In general, a transition contraction can transform a safe STG into an non-safe (2-bounded) one. The corresponding conservative structural conditions guaranteeing that a contraction preserves safeness are developed below.

All of the mentioned dynamic properties can be checked with a reachability analysis, which can be performed by MPSAT. Since we only consider safe nets here, reachability-like properties can be expressed as Boolean expressions over the places of the net. For example, a net fulfils the property $p_1 \wedge p_2 \wedge \neg p_3$ iff some reachable marking has a token on p_1 and p_2 and no token on p_3. (Such properties can be checked by MPSAT.) Below we give Boolean expressions and a conservative test for safeness-preservation.

In the next subsection the general approach of CSC preservation is introduced. In Subsection 6.6.2 it is described how safeness is preserved during reduction, since

non-safe STGs cannot be analysed at all, or only with a greatly reduced efficiency. Unfortunately, transition contractions do not preserve safeness in general, and here it is shown how this can be guaranteed by an efficient reachability analysis as well as by proper structural preconditions. In Subsection 6.6.3 the detection of implicit places and dynamic auto-conflicts with a reachability analysis is described.

6.6.1 CSC-Aware Decomposition

On the base of tree decomposition, we now introduce *CSC-aware* decomposition. Our aim is to reduce the number of CSC conflicts in the components generated by the decomposition algorithm. Ideally, if the original specification is free from CSC conflicts then this should be the case also for the components.

First, we describe how CSC conflicts can be detected on unfoldings. Due to its structural properties (such as acyclicity) the unfolding can represent the reachable states of an STG using *configurations*. A configuration C is a downward-closed set of events (i.e. transitions) without conflicts. Being downward-closed means that if $e \in C$ and f is a predecessor of e then $f \in C$; without conflicts, means that for all distinct events $e, f \in C$, ${}^\bullet e \cap {}^\bullet f = \emptyset$. Intuitively, a configuration is a partially ordered firing sequence, i.e. a firing sequence where the order of firing of some of its events (viz. concurrent ones) is not important.

A CSC conflict can be represented in the unfolding prefix as an unordered *conflict pair* of configurations $\langle C_1, C_2 \rangle$ whose final states are in CSC conflict, as shown in Figure 6.4. It was shown in [KKY04] that the problem of checking if there is such a conflict pair is reducible to SAT, and an efficient technique for finding all CSC conflict pairs was proposed.

Let $\langle C_1, C_2 \rangle$ be a conflict pair. The corresponding *complementary set* $\mathcal{CS}$ is defined as the symmetric set difference of C_1 and C_2. $\mathcal{CS}$ is a *core* if it cannot be represented as the union of several disjoint complementary sets. For example, the core corresponding to the conflict pair shown in Figure 6.4 is $\{e_4, \ldots, e_8, e_{10}\}$.

An important property of complementary sets is that for each signal $s \in Sig$, the differences between the numbers of s^+– and s^-–labelled events are the same in the two parts $C_1 \setminus C_2$ and $C_2 \setminus C_1$ (and are 0 if $C_1 \subset C_2$). This suggests that a complementary set can be eliminated (resolving thus the corresponding encoding conflict), e.g. by introduction of a new internal signal *csc*, and insertion of a csc^+ labelled transition into one part, as then the stated property would be violated. (Note that the circuit has to implement this new signal, and so for the purpose of logic synthesis it is regarded as an output, though it is ignored by the environment.) To preserve the consistency of the STG, the transition's counterpart, csc^-, must also be inserted *outside the core,* in such a way that it is neither concurrent to nor in structural conflict with csc^+.

Another restriction is that an inserted signal transitions must not trigger an input signal transition in order to preserve input-properness.

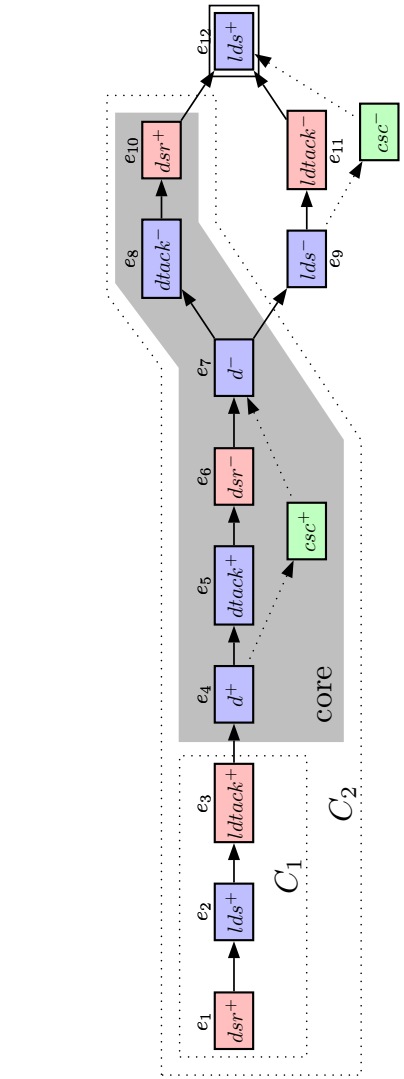
For example, the core in Figure 6.4 can be eliminated by inserting a new signal edge csc^+, somewhere in the core, e.g. concurrently to e_5 and e_6 between e_4 and e_7, and by inserting its complement outside the core, e.g. concurrently to e_{11} between e_9 and e_{12}. (Note that the concurrent insertion of these two transitions avoids an increase in the latency of the circuit, where each transition is assumed to contribute a unit delay.) After adding this signal to the STG, the CSC property is satisfied.

Now, we present CSC-aware decomposition, cf. Figure 6.5. The new algorithm traverses the decomposition tree in a depth-first order starting at the root node. When entering a node u from above, i.e. coming from its parent, the respective reduction operations are performed as in TREE, generating an STG N_u. Additionally, when entering a node from below, i.e. coming back from one of its children, additional operations are performed to resolve CSC conflicts (if any). These operations are encapsulated as *CSC-jobs*, which are created in the leafs.

When a leaf is reached, MPSAT checks the corresponding component for CSC conflicts. If there are none, the component is immediately saved as final result. Otherwise, a CSC-job is created, containing the signature of the respective component as well as a conflict pair for each CSC conflict generated by MPSAT. Additionally, the job contains an initially empty set of signals D; in general, D contains signals which should be added to the component to destroy CSC conflicts. The leaf does not know about any additional signals, so it cannot know about signals destroying the conflicts detected in the leaf; consequently, D is empty. This CSC-job is stored within the parent node of the leaf.

If the algorithm enters an inner node u from below, all CSC-jobs present there are processed: for each CSC-job J, a new one J' is created as a copy of J but with an empty set of conflict pairs. Then, the algorithm checks for each CSC conflict pair in J, if the corresponding conflict is also present in the STG N_u associated with u, as follows. The firing sequences of the conflict pair are mapped to firing sequences of N_u by *inverse projection* (see below). If these new firing sequences do not lead to a CSC conflict in N_u, analysing the inverse projection shows which signals helped to destroy the conflict, and one of them is added to D' in J'. Otherwise, the new firing sequences are added as conflict pair to J'.

If, after handling all conflict pairs of J, J' does not contain any, then one adds D' to the inputs of J' and generates the final component with this modified signature from N_u with reduction (using strategy REORDERING); this component may contain new CSC conflicts, for which a new CSC-job is created and stored in the parent of u. Otherwise, J' is stored in the parent of u in the hope that more signals might help to destroy the remaining conflicts.



[lds]=$\overline{\text{csc}}\cdot(\overline{\text{ldtack}}\cdot\text{dsr}+\text{lds})+\text{d}$ [dtack]=d
[d]=$\text{lds}\cdot\overline{\text{csc}}\cdot\text{ldtack}+\text{d}\cdot\text{dsr}$ [csc]=$\text{d}+\text{lds}\cdot\text{csc}$

Figure 6.4: Unfolding prefix of VME bus controller: the unfolding prefix with the corresponding conflict core, and a way to resolve it by adding a new signal *csc* (top), and a complex-gate implementation (bottom).

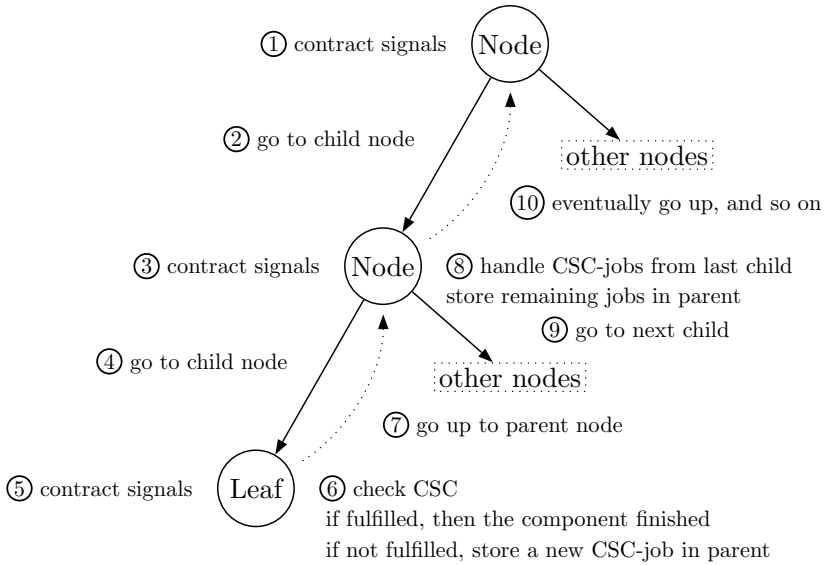


Figure 6.5: Outline for CSC-aware decomposition. Step 8 is repeated every time a node is entered from a child, step 9 possibly includes the generation of new CSC-jobs.

If all CSC-jobs stored in u are processed, the depth-first traversal is continued and the next child of u is entered. This may result in new jobs stored in u, which are processed before its next child is entered, and so on. If all children of u have been considered and the last CSC-job was handled, the decomposition algorithm returns to the parent of u and deals with the respective CSC-jobs, etc.

To sum it up, for each final component containing CSC conflicts, a CSC-job is created containing the necessary information to produce a component where these conflicts have been resolved with signals contained in the specification.[3] For this, additional inputs are determined by 'pushing' the CSC-job upwards in the tree and determining helpful signals in each node. This CSC-job handling is interleaved with the actual tree decomposition.

We now explain inverse projection: let N and N' be two STGs such that N' is obtained from N by a sequence of reduction operations. If v' is a firing sequence of N', we call a firing sequence v of N an inverse projection of v' if v' is the projection of v on the transitions of N'.

[3] In contrast, ordinary CSC conflict resolution adds *new* internal signals.

For a conflict pair (v'_1, v'_2) of N' the corresponding signal change vectors $b^{v'_1}$ and $b^{v'_2}$ coincide. If the inverse projection (v_1, v_2) of this pair is such that $b^{v_1} = b^{v_2}$, then also these sequences lead to a CSC conflict; otherwise, the corresponding conflict is destroyed by delambdarising any of the signals s for which $b^{v_1}(s) \neq b^{v_2}(s)$.

Hence, let N' be obtained from N by the secure contraction of t. (The case of deletion of an implicit place or a redundant transition is trivial.) The inverse projection v of v' is obtained by firing the transitions of v' in N, one by one, while possible. If, at some point, a transition of v' cannot be fired then t is fired (it is guaranteed to be enabled in such a case due to Corollary 3.11).[4] This process is continued until all the transitions of v' are fired, yielding v. One can see that a shortest inverse projection is computed by the described procedure.

A sequence of contractions can be processed in the reverse order one-by-one. However, for efficiency the inverse projection is computed in one step in this case: if a transition t of v' cannot be fired in N, a breadth-first search is started at the corresponding marking of N to get a shortest sequence of transitions from $N \setminus N'$ which enables t.

This algorithm is complete, i.e. it guarantees for a specification with CSC that each component has CSC, too. This is due to the fact that a CSC-job can be pushed up to the root node, where CSC is fulfilled initially. In practice, however, one should stop this after a fixed number of levels and try to resolve the remaining CSC conflicts with new internal signals instead. Therefore, the algorithm is also applicable to specifications which have CSC conflicts initially.

Furthermore, if some conflict of a CSC-job can be destroyed with any of several signals, one could determine a minimal set of signals which destroys all conflicts of this job. This has not been investigated for the time being and is left as future research.

6.6.2 Safeness-Preserving Contractions

A transition contraction preserves boundedness, but, in general, it can turn a safe net into a non-safe one, as well as introduce weighted arcs. However, since unfolding techniques are not very efficient for non-safe net, we assume that the initial STG is safe, and perform only *safeness-preserving* contractions, i.e ones which guarantee that if the initial STG was safe then the transformed one is also safe. (Note that the transitions with weighted arcs must be dead in a safe Petri net, and so we can assume that the initial and all the intermediate STGs contain no such arcs.)

We now give a sufficient structural condition for a contraction being safeness-preser-

[4] A detailed proof would have to consider the subtle case that the simulation of Corollary 3.11 could have 'fired' t earlier than necessary; such a firing of t can be moved to the end of v reaching the same marking.

ving. Then we will show how this can be checked with a partial reachability analysis and also how a single unfolding prefix can be used for checking if a sequence of contractions is safeness-preserving.

Lemma 6.3
Let N be a safe net and let N' be obtained from N by the secure contraction of a transition t. Then all places which are not generated by the contraction (i.e. all places from $P' \setminus ({}^\bullet t \times t^\bullet)$) are safe.

Proof. Let $\mathcal{S}'$ be the transition simulation between N' and N from Corollary 3.11, and let M' be a reachable marking with $M_{N'}[v'\rangle M'$. Then $M_N[v\rangle M$ with $v' = v|_{T'}$ and $(M', M) \in \mathcal{S}'$. If $p' \in P'$ is not generated by the contraction, it is of the form $(p, \star)$ for some $p \in P$. The marking equality now implies $M'(p') = M'((p, \star)) = M(p) + M(\star) \leq 1$. □

Theorem 6.4 (Structural safeness-preservation)
The secure transition contraction of t in a net N is safeness-preserving if

(1) $|{}^\bullet t| = 1$ *or*

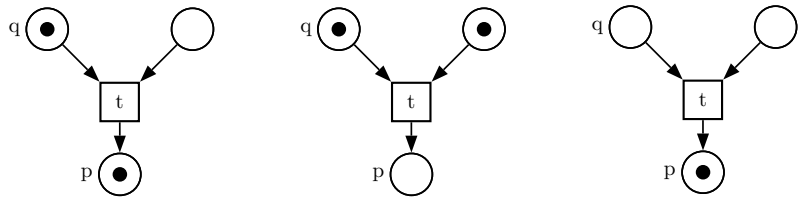
(2) $|t^\bullet| = 1$, ${}^\bullet(t^\bullet) = \{t\}$ *and*

- *(a)* *N is live and reversible*
  or
- *(b)* $M_N(p) = 0$ *with* $t^\bullet = \{p\}$
  (or equivalently: the contraction is type-2 secure)

Proof. Let $\mathcal{S}$ and $\mathcal{S}' \subseteq \mathcal{S}$ be the corresponding transition simulations from Corollary 3.11. Observe that a transition contraction turns a safe net into a 2-bounded one, which does not have to be safe.

(1) $|{}^\bullet t| = 1$: let ${}^\bullet t = \{p\}$. If N is safe but N' is not, one of the places generated during contraction is non-safe (Lemma 6.3), e.g. (p, q) with $q \in t^\bullet$. Hence, a marking M' of N' exists with $M_{N'}[u\rangle M'$ and $M'((p, q)) = 2$. Corollary 3.11 implies that there is a marking M of N with $(M', M) \in \mathcal{S}'$. Hence $M(p) + M(q) = 2$, and so due to the safeness of N, $M(p) = M(q) = 1$. Therefore t is enabled due to $M(p) = 1$, and firing it puts a token on q which already contained one, contradicting the safeness of N.

(2) $|t^\bullet| = 1$ and ${}^\bullet(t^\bullet) = \{t\}$: let N be safe and N' be non-safe, and let $t^\bullet = \{p\}$. Analogously to the first case, Lemma 6.3 implies that there is a reachable marking M_1 of N such that $M_1(q) = M_1(p) = 1$ for some place $q \in {}^\bullet t$; the picture below shows a corresponding fragment of N. Observe that, by safeness of N, some places of ${}^\bullet t$ must be empty.



M_1 M_2 M_3

If (a) holds, i.e. N is live, there must be a reachable marking M_2 which enables t; since N is safe, M_2 puts exactly one token in every place in ${}^\bullet t$ and no token on p. The marking M_3 which is reachable from M_2 by firing t puts one token in p and no tokens in the places in ${}^\bullet t$. Since N is reversible, M_1 is reachable from M_3.

If (b) holds, i.e. p is initially unmarked, M_1 can only be reached via markings M_2 and M_3, since only t can put a token on p.

In both cases there are transition sequences v_1 and v_2 such that

$$M_N[v_1\rangle M_2[t\rangle M_3[v_2\rangle M_1\,.$$

Moreover, without loss of generality, one can assume that v_2 does not contain t; indeed, if $v_2 = v_2' t v_2''$ with v_2'' not containing t, one finds that $M_N[v_1 t v_2'\rangle M_2'[t\rangle M_3'[v_2''\rangle M_1$ with the required properties.

Since v_2 does not contain t, the token on p cannot be removed by the transitions in v_2 because only t can put it there again. Hence $M_2[v_2\rangle M$ with $M(q) = 2$, because the firing of v_2 increases the marking of q by 1. This contradicts N being safe. □

Figure 6.6 shows two counterexamples: the leftmost net violates the condition that either the pre- or postset of t has to contain a single place; one can see that, the contraction of t generates an non-safe net. The net in the middle violates the condition ${}^\bullet(t^\bullet) = \{t\}$ of the second case of Theorem 6.4 (i.e. that the place in the postset of t must not have incoming arcs other than from t); the rightmost net is obtained by contracting t in the net in the middle.

In practice, the decomposition algorithm checks the condition 2(b) which makes no assumptions about the net which are difficult to verify. This is important since there exist STGs which are neither live nor reversible, e.g. ones which have some initialisation part which is executed only once in the beginning.

If the specification is guaranteed to be live and reversible, it is also possible to use condition 2(a); then Proposition 3.13 is needed to apply such contractions repeatedly.

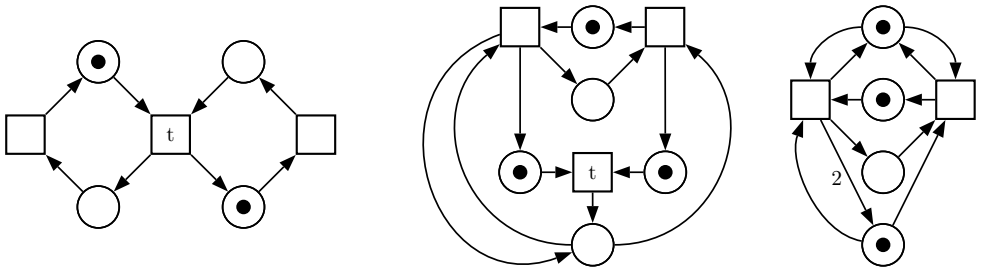


Figure 6.6: Examples of non-safeness-preserving contractions

Theorem 6.4 gives us a sufficient *structural* condition to check whether a contraction is safeness-preserving. To be sure one has to perform a reachability analysis; the rest of this section shows this can be achieved on the unfolding of a net.

Theorem 6.5
Let N be a safe STG and $t \in T$ be such that the contraction of t is secure. The contraction of t is safeness-preserving iff the following property does not hold:

$$\left(\bigvee_{r \in {}^\bullet t} r\right) \wedge \left(\bigvee_{s \in t^\bullet} s\right) . \qquad (*)$$

Proof. Let N' be the resulting STG, $\mathcal{S}$ be the transition simulation between N and N', and $\mathcal{S}' \subseteq \mathcal{S}^{-1}$ be the transition simulation between N' and N from Corollary 3.11.

($\Rightarrow$) Suppose there is a reachable marking M of N fulfilling the Boolean expression. Then there are two places $r \in {}^\bullet t$ and $s \in t^\bullet$ with $M(r) = M(s) = 1$. Since the contraction is defined, $r \neq s$. Since M is reachable, $M_N[v\rangle M$ for some firing sequence v of N. Hence, $M'_N[v|_{T'}\rangle M'$ with $(M, M') \in \mathcal{S}$. Since t was contracted, $M'((r,s)) = M(r) + M(s) = 2$, i.e. N' is not safe.

($\Leftarrow$) Suppose now that N' is non-safe due to a place p' and a reachable marking M' of N'. Lemma 6.3 implies that p' is newly generated by the contraction, i.e. $p' \equiv (r,s)$ with $r \in {}^\bullet t$ and $s \in t^\bullet$. Since M' is reachable, $M'_N[v'\rangle M'$ for some firing sequence v' of N'. Hence, $M_N[v\rangle M'$ with $v' = v|_{T'}$ and $(M', M) \in \mathcal{S}'$. Then, the marking equality implies $M'((r,s)) = M(r) + M(s) > 1$. Since N is safe, $M(r) = M(s) = 1$ and M fulfils the Boolean expression. □

To check the safeness of a sequence of contractions on a single unfolding, one has to build expressions over the original net, which are derived from the intermediate nets.

For this, we use place-projection (Definition 3.7) and the extended marking equality (Proposition 3.12).

Theorem 6.6
Let N be a safe STG and let N' be an STG obtained from it by a sequence of secure safeness-preserving transition contractions. Then the contraction of a transition t in N' is safeness-preserving iff the following property does not hold in N:

$$\left(\bigvee_{r\in\Phi_{\bullet t}} r\right) \wedge \left(\bigvee_{s\in\Phi_{t\bullet}} s\right) \qquad (**)$$

With $\Phi_X = \bigcup_{p\in X}[\Phi_N^{N'}(p)]$ with $X \subseteq P$; recall that $[.]$ denotes the support of a multiset.

Proof. We will show that the equation $(**)$ can be fulfilled in N if and only if the equation $(*)$ from Theorem 6.5 can be fulfilled in N'. Let $\mathcal{S}$ be the transition simulation between N and N' and $\mathcal{S}' \subseteq \mathcal{S}^{-1}$ be the transition simulation between N' and N, whose existence is implied by Corollary 3.11.

($\Leftarrow$) Suppose $(**)$ is fulfilled for the marking M reached by some firing sequence v of N. Then there are two places $r \in \Phi_{\bullet t}$ and $s \in \Phi_{t\bullet}$ with $M(r) = M(s) = 1$. Moreover, by Proposition 3.12 and safeness of N', $M'_N[v|_{T'}\rangle M'$ with $(M, M') \in \mathcal{S}$ and $M'(q') = \sum_{q\in P} \Phi_N^{N'}(q')(q) \cdot M(q) \leq 1$ for each place $q' \in P'$. Since $M(r) = 1$, $r \in [\Phi_N^{N'}(q')]$ then implies $M'(q') = 1$ for $q' \in P'$ (†). In particular, since $r \in \bigcup_{p\in{}^\bullet t}[\Phi_N^{N'}(p)]$ there is a place $r' \in {}^\bullet t$ with $r \in [\Phi_N^{N'}(r')]$, and then by (†) $M'(r') = 1$.
Analogously there is a place $s' \in t^\bullet$ with $M(s') = 1$ and therefore $(*)$ is fulfilled for M'.

($\Rightarrow$) Suppose now that $(*)$ is fulfilled for the marking M' reached by some firing sequence v' of N', i.e. there are two places $r' \in {}^\bullet t$ and $s' \in t^\bullet$ with $M'(r') = M'(s') = 1$. Then there is a firing sequence v of N such that $v' = v|_{T'}$ and $M_N[v\rangle M$ with $(M', M) \in \mathcal{S}'$. Moreover, by Proposition 3.12, $M'(q') = \sum_{q\in P} \Phi_N^{N'}(q')(q) \cdot M(q)$ for each place $q' \in P'$. In particular, $1 = M'(r') = \sum_{q\in P} \Phi_N^{N'}(r')(q) \cdot M(q)$. This implies that there is a place $r \in [\Phi_N^{N'}(r')]$ with $M(r) = 1$. Analogously, one can show that there is a place $s \in [\Phi_N^{N'}(s')]$ with $M(s) = 1$, and thus $(**)$ is fulfilled for M. □

The last result makes it possible to use one unfolding prefix to check that a given sequence of contractions is safeness-preserving. This works as follows:

(1) Starting with N, check if the first contraction is safeness-preserving using Theorem 6.5.

(2) Perform the contraction resulting in a new STG.

(3) To check if the next contraction is safeness-preserving in the new STG, build an expression over N using Theorem 6.6.

(4) Repeat steps 2 and 3 until all the desired contractions are performed.

Observe that all the new nets are generated and used to build an expression over the original net, thus the original unfolding prefix can be used in step (3), which avoids the expensive generation of several unfolding prefixes.

6.6.3 Implicit Places and Dynamic Auto-Conflicts

As it was already mentioned, the deletion of implicit places is important for the success of decomposition. As a conservative condition and as discussed in previous chapter, DesiJ only looks for shortcut and loop only places. Unfolding-based reachability analysis makes it possible to check exactly whether a place is implicit: a place p of N is implicit iff the following property does not hold:

$$\neg p \wedge \left(\bigvee_{t \in p^\bullet} \bigwedge_{q \in {}^\bullet t \setminus \{p\}} q \right) , \qquad (***)$$

i.e. there is a reachable marking such that only p being unmarked prevents the firing of a transition in $p^\bullet$.

It is possible to detect all implicit places of a net with a single unfolding: observe first that the unfolding of a net in which an implicit place was deleted can be obtained from the original unfolding by deleting all occurrences of this place[5], because the sets of reachable markings and firing sequences do not change. In fact, it is not even necessary to modify the original unfolding to check the implicitness of another place, since previously deleted implicit places will simply not occur in the corresponding instantiation of $(***)$.

It is furthermore sufficient to check $(***)$ *once* for each place p (and to delete p where applicable) in order to delete as much implicit places as possible. To see this, observe that the deletion of an implicit place cannot turn a non-implicit place into an implicit one. Indeed, suppose p_1 is implicit and deleted in N, yielding N_1, and p_2 is implicit and deleted in N_1, yielding N_2. Then $FS(N) = FS(N_1) = FS(N_2)$ by definition of implicit places. Suppose now that p_2 is deleted first in N, yielding N_1', and p_1 is deleted in N_1', yielding N_2 again. Then $FS(N) \subseteq FS(N_1') \subseteq FS(N_2) =$

[5]This is in general not true for the deletion of arbitrary places.

$FS(N)$, since deleting places can only increase the set of firing sequences. Therefore $FS(N) = FS(N_1') = FS(N_2)$, which shows that p_2 is implicit in N.

On the other hand, deletion of an implicit place may render a different implicit place non-implicit, e.g. if two places are duplicates of each other (a common situation during decomposition). Hence, it is not correct to calculate all implicit places first and to delete them all afterwards.

Additionally, it is possible to detect dynamic auto-conflicts with unfoldings. A conservative test for the presence of an auto-conflict is the presence of a structural auto-conflict. Unfolding-based reachability analysis makes it possible to check exactly for the presence of an auto-conflict as follows. In a safe STG, distinct transitions t_1 and t_2 in structural auto-conflict are in dynamic conflict iff the following property holds:

$$\bigwedge_{p \in {}^\bullet t_1 \cup {}^\bullet t_2} p \; .$$

Using this exact test for each suitable pair of transitions can reduce the number of times the decomposition algorithm has to backtrack, which ultimately can result in an improved runtime and smaller final components.

However, in our experience, structural auto-conflicts are in most cases also dynamic ones (cf. also the discussion for the risky strategy in the next section). Hence, we think the additional computational effort of performing the dynamic check is lost, and the dynamic test for dynamic auto-conflicts is not used in DesiJ.

6.7 Conclusion

6.7.1 Results

The strategies and other implementation improvements of the previous sections have been implemented in the tool DesiJ. This section presents the experimental results for a number of benchmark examples circulating in the STG community as well as for some newly generated benchmarks based on Balsa [EB02] handshake components. In [SVWK06], similar benchmarks for DesiJ were presented; however, the results there differ very much from the present ones. The reason is that the undo stack was not implemented then and also several other smaller optimisations have been implemented since then.

The algorithm for the CSC-aware approach to decomposition is denoted by DeMpSy (DEcomposition and MPsat for SYnthesis); it uses the undo-stack, forbids self-triggering and includes the dynamic checking of implicitness and safeness-preservation.

All experiments were performed on a Pentium 4 HT with 3 GHz and 2 GB RAM. The runtimes in the tables are given in seconds (rounded). We start with a table showing the basic parameters of these benchmarks. Then a comparison of our decomposition strategies using the newly developed undo stack in contrast to copying STGs at the respective savepoints is presented in Table 6.2. Then we compare the detection of implicit places with unfolding and structural methods in Table 6.3. Table 6.4 presents a more extensive comparison of the different decomposition strategies, which shows that Tree is the fastest one. This is followed by a presentation of the impact of Aggregation on the *size* of a decomposition in Table 6.5. In Table 6.6 we compare Tree (with and without allowing self-triggering) with DeMpSy. Finally, in Table 6.7 the results of the decomposition of large STGs with DeMpSy are presented.

The risky auto-conflict detection was tested with Basic, Reordering, LazyBack and Tree, but turned out to be not very successful in general: for most decompositions, there are some final components with at least one auto-conflict; this is also the case for nearly every benchmark of the `dup.x` series (see below) even when using the conservative approach, but risky worsens this problems. Furthermore, the runtimes are not much smaller than for the conservative conflict detection. Nevertheless, if at least most of the components are free from auto-conflicts, the risky approach might be useful for semi-automatic decomposition. Here we only present results for the conservative approach. Also, there is no difference between the results of LazySingle and LazyMulti and therefore only the results for the former are given.

All benchmarks were performed using safeness-preserving contractions (Section 6.6.2) to keep the comparison fair and to allow for the synthesis with Mpsat. In particular, DeMpSy has to use safeness-preserving contractions to allow the dynamic checking of properties (like implicitness) with Mpsat, and also safeness-preservation is checked dynamically there. The other strategies only use the respective structural tests.

In Table 6.1, all the benchmarks we used are presented together with the respective number of signals, places, transitions and arcs. The series `2pp.x`, `3pp.x` and `dup.x` were used by other authors before, whereas the STGs from the `seqpartree.x` series are newly generated. Observe that the latter series contains some very large STGs. They are derived from Balsa specifications as described below, and we produced them in two variants: with and without CSC initially. Similar benchmarks were used before in [CC06]. The benchmark `seqpartree(21,10)` from there is nearly the same as `seqpartree.05` here; the difference is that we did not hide the internal handshake signals between the components.

Name	$\|Sig\|$	$\|P\|$	$\|T\|$	$\|W\|$
2pp.arb.nch.03.csc	11	40	24	84
2pp.arb.nch.06.csc	17	64	36	132
2pp.arb.nch.09	23	86	48	176
2pp.arb.nch.12.csc	29	112	60	228
2pp.wk.03.csc	7	24	14	48
2pp.wk.06.csc	13	48	26	96
2pp.wk.09.csc	19	72	38	144
2pp.wk.12.csc	25	96	50	192
3pp.arb.nch.03.csc	16	59	36	126
3pp.arb.nch.06.csc	25	95	54	198
3pp.arb.nch.09.csc	34	131	72	270
3pp.arb.nch.12.csc	43	167	90	342
3pp.wk.03.csc	10	36	20	72
3pp.wk.06.csc	19	72	38	144
3pp.wk.09.csc	28	108	56	216
3pp.wk.12.csc	37	144	72	288
dup.mst.mod.1	22	129	100	296
dup.mst.mod.2	21	113	88	264
dup.mst.mod.3	22	134	98	308
dup.mst.mod.3.1	23	140	100	321
dup.mst.mod.3.3	22	135	98	310
dup.mst.mod.3.4	26	145	107	330
dup.mst.mod.3.5	28	153	115	346
dup.mst.mod.3.6.1	28	153	115	346
dup.mst.mod.3.6	28	153	115	346
dup.mst.mod.3.7	29	159	119	359
dup.mst.mod.3.8	29	159	119	369
dup.4.ph.dt.pl.1	27	133	123	286
dup.4.ph.dt.pl.2	27	135	123	290
dup.4.ph.dt.pl.3	27	136	123	292
dup.4.ph.dt.pl.mst.3	26	114	105	242
dup.4.ph.dt.pl.mst.4.a	23	109	96	234
dup.4.ph.dt.pl.mst.4	26	113	100	242
dup.4.ph.dt.pl.slv.3	26	121	112	258
dup.4ph.csc	27	135	123	290
dup.4ph	27	133	123	286
dup.4ph.mtr.csc	26	114	105	242
dup.4ph.mtr	23	109	96	234
dup.mtr.mod.csc	28	153	115	346
dup.mtr.mod	22	129	100	296
seqpartree.02	14	38	28	76
seqpartree.02.csc	19	48	38	96
seqpartree.03	30	86	60	172
seqpartree.03.csc	29	104	78	208
seqpartree.04	62	190	124	380
seqpartree.04.csc	87	240	174	480
seqpartree.05	126	382	252	764
seqpartree.05.csc	167	464	334	928
seqpartree.06	254	798	508	1596
seqpartree.06.csc	359	1008	718	2016
seqpartree.07	510	1566	1020	3132
seqpartree.07.csc	679	1904	1358	3808
seqpartree.08	1022	3230	2044	6460
seqpartree.08.csc	1447	4080	2894	8160
seqpartree.09	2046	6320	4092	12604
seqpartree.09.csc	2727	7664	5454	15328
seqpartree.10	4094	12958	8188	25916
seqpartree.10.csc	5799	16368	11598	32736

Table 6.1: The benchmark STGs used in this section together with their number of signals, places, transitions and arcs.

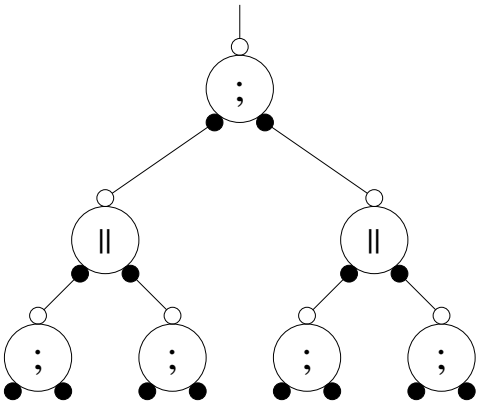


Figure 6.7: `seqpartree.03`. Filled dots denote active handshake ports (they can start a handshake), blank nodes denote passive ones. Each port is implemented by two signals, *req* and *ack*. If two ports are connected then the parallel composition merges these four signals into two outputs.

The `seqpartree.x` STGs are generated from two basic Balsa handshake components: 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, see also the discussion of handshake circuits in Section 2.2.3 and Example 5.1. The benchmark examples `seqpartree.x` are complete binary trees with alternating levels of sequencers and parallelisers, as illustrated in Figure 6.7 (x is the number of levels of the tree). They 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 made of handshake components (e.g. trees of parallelisers only); the results did not differ much, so we exemplarily present `seqpartree.x` only. Due to their construction, the resulting STGs contain a lot of implicit places which were not contracted in advance, cf. also the discussion of Table 6.3.

The first experimental results can be found in Table 6.2. For them, we performed decomposition with the four variants Basic, Reordering, LazyBack and Tree, when using the undo stack and when using copying of savepoint STGs. (Only a representative selection of benchmarks is presented.) As expected, the undo stack increases the efficiency of decomposition a lot, especially for large STGs; in particular,

in every case the first approach is faster. Hence, the undo stack was used in all further experiments. Observe also that in nearly every case, Tree with copying is faster than the other approaches with the undo stack. It turns out that the Tree strategy benefits most from this heuristic (up to factor 12). Somewhat surprisingly, the Basic and Reordering strategies also benefit from it (factors up to 7.5 and 6.5, respectively), even though their only savepoint is the original STGs. This can be explained by the fact that backtracking might be performed early in the reduction process, when undoing changes is much more efficient then copying the entire STG. It was however unexpected that LazyBack benefits least from the undo stack (the factor of only about 1.5); an explanation of this phenomenon is yet to be found.

This results already indicate that it is most worthwhile to study Tree in greater detail.

	Basic		Reorder		Lazy		Tree	
Name	Undo	Copy	Undo	Copy	Undo	Copy	Undo	Copy
2pp.wk.12.csc	< 1	2	< 1	2	3	3	< 1	1
2pp.arb.nch.12.csc	2	4	1	4	4	5	< 1	1
3pp.arb.nch.12.csc	3	13	3	13	12	15	1	2
3pp.wk.12.csc	2	7	2	7	9	11	1	1
seqpartree.05	2	8	2	8	38	49	1	3
seqpartree.05.csc	3	10	3	9	70	96	1	5
seqpartree.06	18	99	39	212	476	623	1	13
seqpartree.06.csc	25	187	26	171	1294	1998	2	24

Table 6.2: Comparison between undo stack and copying STGs.

In Table 6.3 the detection of implicit places with unfoldings (see Section 6.6.3) is compared to the structural detection. The `seqpartree.x` series is especially useful for this comparison, because these STGs are marked graphs, for which it was shown in Section 4.1 that all their implicit places are shortcut or loop-only places. The latter ones are also detected by the structural approach, hence the results are the same in both cases. As mentioned above, the `seqpartree.x` STGs contain a lot of implicit places, and their number is printed in the second column. Note also that these runtimes are for the detection of the implicit places only and do not include decomposition.

Only for the first two benchmarks the dynamic detection with Mpsat is faster (when considering the unrounded runtimes for several runs). As a consequence, DeMpSy uses the structural approach for STGs with more than 100 nodes and the dynamic one otherwise. The same applies to the structural/dynamic detection of safeness-preserving contractions.

The dynamic detection of implicit places might be more advantageous for STGs which

Name	impl. Places	unfolding	structural
seqpartree.02	8	< 1	< 1
seqpartree.03	24	< 1	< 1
seqpartree.04	56	< 1	< 1
seqpartree.05	120	2	< 1
seqpartree.06	248	4	< 1
seqpartree.07	504	11	< 1
seqpartree.08	1016	45	2
seqpartree.09	2040	164	7
seqpartree.10	4088	745	26

Table 6.3: Detection of implicit places with unfoldings and structural methods. (Just detection without decomposition.)

are not marked graphs, where implicit places might lead to unnecessary backtracking if they are not deleted.

Now, we come to the main benchmarks in Table 6.4. In columns two to five, the decomposition time of the respective strategy is printed. The resulting components are the same in every case, and hence the synthesis time of the components with MPSAT (including CSC resolution if needed), the number of successfully generated components (denoted as successful/all components) and the overall number of new internal signals (for achieving CSC in the components) are only printed once in the last three columns.

All STGs (except the `seqpartree.x` ones) were decomposed using the finest possible partition, i.e. usually each component produces one output. Since REORDERING turned out to be faster then BASIC, it is used as the reduction algorithm in the intermediate stages of LAZYBACK and TREE, i.e. when a set of signals has to be contracted in an intermediate STG. For these benchmarks, self-triggering was forbidden and backtracking was performed in this case.

In contrast to the decomposition method of [CC03, CC06] we also allow components with more than one output. This was utilised for the `seqpartree.x` series, where the initial partitions were chosen such that each component corresponds to one handshake component. Other partitions of the outputs might lead to further speedups. (This approach conforms to the intended use for *resynthesis* [CC06], cf. also the discussion in Chapter 9.)

Using BASIC as a reference point for the other strategies is somewhat problematic: due to its 'random' order, the results for different benchmarks may vary. However, this seems to be not the case here. LAZYBACK does not meet the expectations concerning the runtime. For small STGs, its performance does not differ much from that of

Name	Basic	Reord.	Lazy	Tree	Synth.	Comp.	Int.
2pp.arb.nch3.csc	< 1	< 1	< 1	< 1	< 1	10 / 10	0
2pp.arb.nch6.csc	< 1	< 1	< 1	< 1	< 1	16 / 16	0
2pp.arb.nch9.csc	1	< 1	2	< 1	< 1	22 / 22	0
2pp.arb.nch.12.csc	1	1	4	< 1	< 1	28 / 28	0
2pp.wk3.csc	< 1	< 1	< 1	< 1	< 1	7 / 7	0
2pp.wk6.csc	< 1	< 1	< 1	< 1	< 1	13 / 13	0
2pp.wk9.csc	< 1	< 1	1	< 1	< 1	19 / 19	0
2pp.wk.12.csc	< 1	< 1	3	< 1	< 1	25 / 25	0
3pp.arb.nch3.csc	< 1	< 1	< 1	< 1	< 1	14 / 14	0
3pp.arb.nch6.csc	3	1	2	< 1	< 1	23 / 23	0
3pp.arb.nch9.csc	8	2	5	< 1	< 1	32 / 32	0
3pp.arb.nch.12.csc	3	3	12	1	< 1	41 / 41	0
3pp.wk3.csc	< 1	< 1	< 1	< 1	< 1	10 / 10	0
3pp.wk6.csc	< 1	< 1	1.3	< 1	< 1	19 / 19	0
3pp.wk9.csc	1	1	4	< 1	< 1	28 / 28	0
3pp.wk.12.csc	2	2	9	< 1	< 1	37 / 37	0
dup.mst.mod.1	2	1	2	1	15	1 / 10	1
dup.mst.mod.2	1	1	1	1	11	1 / 9	1
dup.mst.mod.3	2	2	2	1	147	1 / 11	1
dup.mst.mod.3.1	3	2	2	2	3	3 / 11	0
dup.mst.mod.3.3	3	2	2	2	53	2 / 11	0
dup.mst.mod.3.4	5	3	5	2	1	4 / 14	0
dup.mst.mod.3.5	5	4	6	3	1	6 / 15	0
dup.mst.mod.3.6.1	5	4	6	3	1	6 / 15	0
dup.mst.mod.3.6	5	4	6	3	1	6 / 15	0
dup.mst.mod.3.7	6	4	7	3	1	6 / 16	0
dup.mst.mod.3.8	6	4	7	3	1	6 / 16	0
dup.4.ph.dt.pull.1	3	2	2	2	< 1	13 / 15	0
dup.4.ph.dt.pull.2	3	2	2	2	< 1	15 / 15	0
dup.4.ph.dt.pull.3	3	2	2	2	< 1	12 / 15	0
dup.4.ph.dt.pull.mst.3	2	2	2	1	< 1	14 / 16	0
dup.4.ph.dt.pull.mst.4.a	1	1	1	1	< 1	8 / 11	1
dup.4.ph.dt.pull.mst.4	2	2	2	2	< 1	10 / 13	0
dup.4.ph.dt.pull.slv.3	2	2	2	2	1	12 / 16	0
dup.4ph.csc	3	2	2	2	< 1	15 / 15	0
dup.4ph	3	2	2	2	< 1	13 / 15	0
dup.4ph.mtr.csc	2	2	2	1	< 1	14 / 16	0
dup.4ph.mtr	1	1	1	1	< 1	8 / 11	1
dup.mtr.mod.csc	5	4	6	3	1	6 / 15	0
dup.mtr.mod	2	1	2	1	16	1 / 10	1
seqpartree5	2	2	36	< 1	1	31 / 31	41
seqpartree5.csc	3	3	70	< 1	< 1	31 / 31	0
seqpartree6	11	11	442	3	2	63 / 63	105
seqpartree6.csc	17	17	738	2	2	63 / 63	0
seqpartree7	73	72	–	14	4	127 / 127	169
seqpartree7.csc	92	96	–	14	4	127 / 127	0

Table 6.4: Comparison of the decomposition strategies.

the other methods, but for the large ones it deteriorates; there is no satisfactory explanation for this yet. As expected, the runtimes of Tree are always minimal, and this effect is most obvious for the `seqpartree.x` series at the end of Table 6.4.

Regarding the synthesis, the `dup.x` series seems to be unsuitable for decomposition: in every case there are components which could not be synthesised, and sometimes this even holds for the majority of the components. Furthermore, the large synthesis times for some STGs indicate that the decomposition was also not successful when regarding the size of the components. For the other series, the decomposition is successful, and only for the `seqpartree.x` STGs without initial CSC internal signals had to be added during synthesis. Comparing the overall synthesis times of decomposition (i.e. decomposition plus synthesis of the components with Mpsat) with the direct synthesis of the specification with Petrify and Mpsat we get the following result: Petrify is always slower – from a few seconds for the small STGs from `2pp.x` and `3pp.x` up to 6 hours (aborted) for the large ones of these series, and more than 12 hours (aborted) for the `seqpartree.x` STGs. Mpsat is faster for the `2pp.x` and `3pp.x` series (less than 1 second) but much slower for the `seqpartree.x` series (also aborted after 12 hours).

We had hoped that Aggregation would reduce the decomposition runtimes since it saves some reduction steps. Actually, the decomposition runtimes do not differ much from the ones of Tree and are therefore omitted. Furthermore, it turned out that Aggregation also has little impact on the synthesis time and therefore we present only benchmark results for the impact on the size of the state graphs of the components, see Table 6.5. Aggregation was performed with the bounds 3 to 15 for the number of signals a component is allowed to have, i.e. a subtree was aggregated when the corresponding intermediate STG contained not more than this number of signals. In the table the results are given for the value 3, the value 15, and for the value which has resulted in the smallest components. In each case, the total number of reachable markings over all components is given under 'size', and the number of components under '#C'. For the smallest result, also the values for which this result was achieved are given under 'best'.

Aggregation might also have an impact on the *quality* of the components: in general, larger components allow for better optimisation during synthesis, and therefore it might be better to produce components which are just small enough to be synthesised in a reasonable time. This topic is left for further research.

Since Tree turned out to be the best decomposition strategy, the remaining benchmarks are now performed only with Tree, DeMpSy resp.

In Table 6.6 the effects concerning the synthesisability of the components for three approaches are presented: we compared Tree with allowing self-triggering, Tree

	Agg. 3		Agg. best			Agg. 15	
Name	Size	#C	Size	#C	best	Size	#C
2pp.arb.nch.03.csc	123	8	92	3	7	960	1
2pp.arb.nch.06.csc	171	14	166	9	4	2048	2
2pp.arb.nch.09.csc	219	20	214	12	4	16384	2
2pp.arb.nch.12.csc	267	26	262	16	4	12160	3
2pp.wk.03.csc	40	5	40	3	3, 4	128	1
2pp.wk.06.csc	88	11	88	6	3, 4	8192	1
2pp.wk.09.csc	136	17	136	9	3, 4	4608	2
2pp.wk.12.csc	184	23	184	14	3, 4	36864	2
3pp.arb.nch.03.csc	309	11	273	5	7	2248	2
3pp.arb.nch.06.csc	381	20	381	14	3, 4	18688	2
3pp.arb.nch.09.csc	453	29	453	19	3, 4	14536	4
3pp.arb.nch.12.csc	525	38	525	25	3, 4	24776	4
3pp.wk.03.csc	64	7	64	4	3 - 5	1024	1
3pp.wk.06.csc	136	16	136	10	3, 4	10752	2
3pp.wk.09.csc	208	25	208	15	3, 4	9728	3

Table 6.5: Impact of Aggregation on the size and number of the components.

without allowing self-triggering and DeMpSy (viz. CSC-aware decomposition); here, the finest partition was used for the `seqpartree.x`, too. For every approach, the decomposition time, the synthesis time and the number of synthesisable components are given. All specifications have CSC initially, but when allowing self-triggering, there are components which could not be synthesised in each case, even when introducing new internal signals. Forbidding self-triggering helps to perform the `2pp.x` and `3pp.x` series successfully and reduces the number of failures for `seqpartree.x`.

DeMpSy is able to generate synthesisable components also for the `seqpartree.x` series. However, this success is somewhat flawed since it depends on the decomposition tree: to speed up the calculation of the latter, the presented bottom-up algorithm is partially randomised (see Section 8.1.2) resulting in a different result for every run of DesiJ, and only for some runs all components where synthesisable; but in every run the number of failures was smaller than for first two variants. The respective results are marked with *.

One can observe that decomposition and synthesis are slightly slower for the second and third approach. This is due to more frequent backtracking and the CSC conflict detection and resolution (by Mpsat) for DeMpSy. The synthesis time is increased since the final components are larger and more of them have to be synthesised. However, this increase is not critical, especially when considering the success of these approaches.

	Self-Triggering Allowed			Self-Triggering Forbidden			DEMPSY		
Name	dec.	syn.	suc.	dec.	syn.	suc.	dec.	syn.	suc.
2pp.arb.nch.03.csc	< 1	< 1	9 / 10	< 1	< 1	10 / 10	< 1	< 1	10 / 10
2pp.arb.nch.06.csc	< 1	< 1	15 / 16	< 1	< 1	16 / 16	< 1	< 1	16 / 16
2pp.arb.nch.09.csc	< 1	< 1	21 / 22	< 1	< 1	22 / 22	1	< 1	22 / 22
2pp.arb.nch.12.csc	< 1	< 1	27 / 28	< 1	< 1	28 / 28	< 1	< 1	28 / 28
2pp.wk.03.csc	< 1	< 1	7 / 7	< 1	< 1	7 / 7	< 1	< 1	7 / 7
2pp.wk.06.csc	< 1	< 1	13 / 13	< 1	< 1	13 / 13	< 1	< 1	13 / 13
2pp.wk.09.csc	< 1	< 1	19 / 19	< 1	< 1	19 / 19	< 1	< 1	19 / 19
2pp.wk.12.csc	< 1	< 1	25 / 25	< 1	< 1	25 / 25	< 1	< 1	25 / 25
3pp.arb.nch.03.csc	< 1	< 1	13 / 14	< 1	< 1	14 / 14	< 1	< 1	14 / 14
3pp.arb.nch.06.csc	< 1	< 1	22 / 23	< 1	< 1	23 / 23	1	< 1	23 / 23
3pp.arb.nch.09.csc	< 1	< 1	31 / 32	1	< 1	32 / 32	1	< 1	32 / 32
3pp.arb.nch.12.csc	1	< 1	40 / 41	2	< 1	41 / 41	2	< 1	41 / 41
3pp.wk.03.csc	< 1	< 1	10 / 10	< 1	< 1	10 / 10	< 1	< 1	10 / 10
3pp.wk.06.csc	< 1	< 1	19 / 19	< 1	< 1	19 / 19	< 1	< 1	19 / 19
3pp.wk.09.csc	< 1	< 1	28 / 28	< 1	< 1	28 / 28	< 1	< 1	28 / 28
3pp.wk.12.csc	< 1	< 1	37 / 37	< 1	< 1	37 / 37	2	< 1	37 / 37
seqpartree.04.csc	< 1	1	50 / 70	< 1	1	65 / 70	2	1	70 / 70
seqpartree.05.csc	2	2	114 / 134	2	2	126 / 134	4	2	134* / 134
seqpartree.06.csc	5	6	210 / 294	6	6	267 / 294	8	7	294* / 294
seqpartree.07.csc	73	15	466 / 550	83	16	520 / 550	94	16	550* / 550
seqpartree.08.csc	732	67	850 / 1190	791	69	1109 / 1190	834	74	1190* / 1190

Table 6.6: Comparison of methods for preserving CSC.

Name	dec.	syn.	int.
seqpartree.05	2	3	41
seqpartree.06	11	7	105
seqpartree.07	15	12	169
seqpartree.08	138	37	425
seqpartree.09	610	79	681
seqpartree.10	4639	263	1705
seqpartree.05.csc	1	< 0	0
seqpartree.06.csc	3	2	0
seqpartree.07.csc	12	3	0
seqpartree.08.csc	87	10	0
seqpartree.09.csc	649	32	0
seqpartree.10.csc	4543	123	0

Table 6.7: Results of DeMpSy for large benchmarks.

Finally, we present the results of DeMpSy for the `seqpartree.x` series with and without CSC in Table 6.7, and the results demonstrate the real power of the combined approach. The corresponding STGs are very large, and we consider it as an important achievement that the proposed combined approach could synthesise them so quickly. As one can see, an STG with more than 4000 signals is synthesised in about 80 minutes, whereas, as mentioned above, stand-alone Petrify and Mpsat cannot synthesise either of these benchmarks within 12 hours.

6.7.2 Application to other Decomposition Approaches

In this section, we discuss how the new strategies could be used to improve the decomposition methods of Carmona and Cortadella [CC03, Car03] and Yoneda, Onda and Myers [YOM04].

Both of these decomposition methods start with an STG which initially has CSC, and guarantee that each final component will also have CSC. They also assume that each component produces exactly one output (for this reason, Aggregation is not considered in the rest of this section).

In contrast to our decomposition method, in the method of Carmona and Cortadella all relevant signals are determined before reduction: starting with the syntactical triggers, integer linear programming problems are solved to repeatedly add additional relevant signals until CSC can be guaranteed for this component, cf. also Section 5.4.

When these signals are determined, all other ones are lambdarised, and a restricted subset of our reduction operations is applied. If there are non-contractible dummy

transitions, they are removed later in the reachability graph with automata-theoretic methods. As a consequence, backtracking is not needed for this method, and therefore LazySingle and LazyMulti cannot be applied.

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

Tree can also be applied, and since postponing does not occur (as backtracking does not occur), there is no need to change the hopefully optimal pre-calculated decomposition tree during reduction. The application of Tree will therefore definitely increase the efficiency of this decomposition method.

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

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

6.7.3 Conclusion

The prototype implementation of the decomposition algorithm of [VW02] was very successful compared to the former direct synthesis approach. Nevertheless, the improved DesiJ implementation demonstrated that there are enough possibilities to improve performance. Especially Tree in combination with Reordering for the reduction in the nodes turned out to be an excellent strategy for saving time and memory, and DeMpSy additionally has a positive effect on the synthesise of the final components.

As mentioned above, the pre-calculated decomposition tree is not necessarily optimal for the final components, since signals might be moved from nodes to their children. Future work in this direction will be to consider the top-down algorithm for building

preset trees in [KK01]. This strategy starts at the root node – as the tree decomposition does – and adds branches iteratively to the tree. The idea is to interleave this building process with decomposition itself – including postponing – in order to get a better decomposition tree.

The purely structural decomposition approach of [VW02, VK06] can handle large specifications, but it does not take into account the properties of STGs related to synthesisability, such as the presence of CSC conflicts. In contrast, MPSAT can resolve CSC conflicts and perform logic synthesis, but it is inefficient for specifications with more than 40 to 50 signals. In Section 6.6 we demonstrated how these two methods can be combined to synthesise large STGs very efficiently.

One of the main technical contributions was to preserve the safeness of the STGs throughout the decomposition, because MPSAT can only deal with safe STGs. This is not just an implementation issue or a compensation for a missing MPSAT feature, but it is also far more efficient than working with non-safe nets, for which unfolding techniques seem to be inefficient. We also showed how dynamic properties like implicitness and auto-conflicts can be checked with unfoldings and how these checks can be combined with cheaper conservative structural conditions.

Future research is required for the decomposition tree, the calculation of which takes a cubic runtime and memory usage in the number of signals and exceeds the memory usage for decomposition and synthesis by far, see also Section 8.1.2. Here, heuristics are needed which explore the tradeoff between the quality of the decomposition tree and the amount of memory needed for its calculation.

Furthermore, we consider the handling of large STGs resulting from BALSA handshake components as very important, see also Chapter 9. The BALSA system can handle very large specifications due to its approach of syntax directed translation as discussed in Section 2.2.3, but the resulting circuit is inefficient due to a large number of unnecessary state-holding elements, which increase the circuit area and latency. Our combined approach can help here by performing so called *resynthesis*: instead of implementing each handshake component separately, one can combine several such components, hide the internal handshake signals, and synthesise one circuit implementing this network of components. As shown in [CC06], resynthesis can halve the area of the circuit and improve its latency. Finally, AGGREGATION might help here to detect which components should be combined.

Chapter 7

Output-Determinacy

When a circuit is synthesised from an STG, it is often assumed that the specification is deterministic (in the sense of automata theory), and that 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:

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

OR-causality When modelling a situation with a safe Petri net, where the system has to respond to any of several possible stimuli in the same way, non-determinism naturally arises,[1] as shown in Fig. 7.1. OR-causality has been studied in [YKKL94, YKK$^+$96]. For the importance of safe STGs see the discussion in the previous section.

Lambdarising of Signals Non-determinism naturally arises when in a deterministic specification some of the signals are lambdarised, as illustrated in Fig. 7.2.

So far, no satisfactory formal semantics of non-deterministic STGs and in particular for dummy transitions[2] has been given (we will show below that the language is not a

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

[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

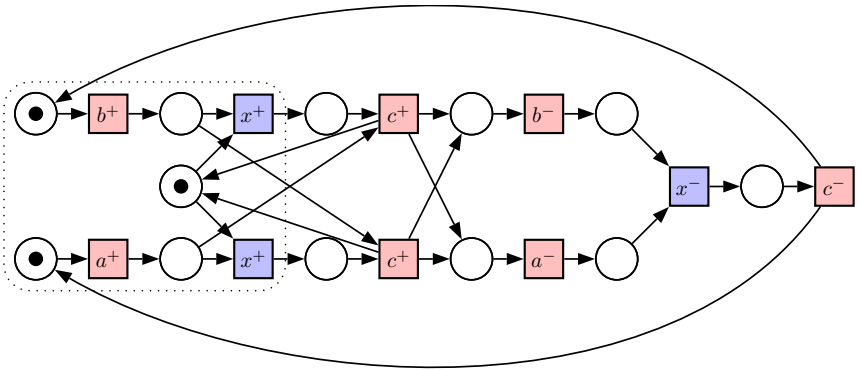


Figure 7.1: OR-causality (the 'interesting' part of the STG is highlighted): a^+ and b^+ are concurrent inputs, and the output x^+ can be produced upon arrival of either of them. Note that the two transitions labelled x^+ are in dynamic auto-conflict, i.e. the specification is non-deterministic. However, it still can be implemented by the deterministic circuit $[x] = a \vee b$.

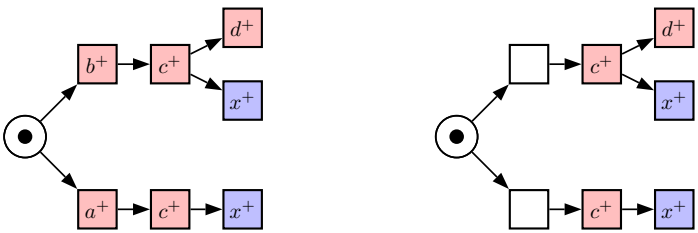


Figure 7.2: Non-determinism due to lambdarisation. After lambdarising signals a and b, the STG becomes non-deterministic, but it is output-determinate and can be implemented (the system can simply wait for c, and produce x upon receiving it; input d can be ignored). Note that the two branches after the non-deterministic choice are not entirely symmetric, as the upper one has an input d which is not present in the lower one.

satisfactory semantics in the non-deterministic case). In this chapter, a formal semantics of non-deterministic STGs is proposed and justified. 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.

As an important application of the developed theory of output-determinacy, we will generalise the decomposition algorithm of Chapter 3 and prove its correctness with the new theory.

Our approach also allows to make the decomposition algorithm more efficient. Each component is obtained from the original STG by lambdarising some of the signals in it, and then contracting the corresponding transitions. The success of this algorithm depends on the ability to securely contract *all* such transitions. If this is not possible, the decomposition algorithm has to backtrack and re-introduce some of the signals into the component, even if they are not really needed for the implementation. In our new version of the algorithm, one can leave such non-contracted lambdarised 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, components can be non-deterministic now; to guarantee correctness, they have to be checked for output-determinacy in the end.

The correctness proof for the new version is essentially just language-based, and might be easier to grasp than the 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 find further valid transformations. Indeed, also a new transformation is described and proven valid.

This chapter is organised as follows: in the next section, the new notion of output-determinacy is introduced and justified. In the following section, we present the new STG-decomposition algorithm and prove its correctness, and give a list of semantics-preserving transformations. In Section 7.3, we show that violations of output-determinacy cannot be resolved while preserving the behaviour. We close with some experimental results and a conclusion.

7.1 Definitions

In this section, we define when a deterministic STG can be regarded as a correct implementation of a specification STG N; we only consider deterministic implemen-

dummy transition can disable other transitions.

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

For the class of output-determinate STGs we show that their language is an adequate semantics, and re-formulate the notion of correct implementation purely in terms of the language; this notion will play an important role as part of the invariant in the proof of correctness of our STG decomposition algorithm described in Section 7.2, 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.

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

The implementation may actually have fewer input signals than N, keeping only those that are relevant for producing the required outputs. In this case, the environment may provide irrelevant inputs, but the implementation simply ignores them — and in this sense, they are always allowed (e.g. in the STG in Fig. 7.2, inputs a and b are irrelevant for producing x and can be ignored). See also the discussion after Definition 3.4.

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 7.1 (Correct Implementation – Language Based)
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) $w\big|_C$ 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.

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,[3] and so an implementation cannot know whether its current state corresponds to M_1 or M_2. Hence, a deterministic implementation must behave consistently with the specification *no matter in which of these markings it is*.

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

7.1.1 Output-Determinacy

Definition 7.2 (Output-Determinacy)
An STG N is called *output-determinate* if $M_N[w\rangle\rangle M_1$ and $M_N[w\rangle\rangle M_2$ implies for every $x \in Out_N$ that $M_1[x^{\pm}\rangle\rangle$ iff $M_2[x^{\pm}\rangle\rangle$. △

For example, the STG in Fig. 7.2 is output-determinate after lambdarising a and b. Clearly, a deterministic STG is also output-determinate; note also that – in contrast to a deterministic STG – an output-determinate STG may contain λ-transitions.

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

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

First of all, the naïve approach consisting in determinisation of a non-deterministic specification using the usual procedure for finite state automata and then proceeding with the synthesis is not always correct. In the context of STGs and circuit synthesis, the result of determinisation can manifest some problems, e.g. non-output-persistency, as illustrated in Fig. 7.3; Fig. 7.4 illustrates a much more dangerous scenario, where the determinised STG contains no apparent problems but the resulting circuit is incorrect according to Definition 7.1. 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.

This rule can be justified by Proposition 7.3.

Proposition 7.3
Let C be a correct implementation of N; in particular, C is deterministic. Then N is output-determinate.

Proof. Assume 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 7.1, $w|_C$ 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 7.1, $M'[x^{\pm}\rangle\rangle$ due to $M_1[x^{\pm}\rangle\rangle$, and, on the other hand, $\neg M'[x^{\pm}\rangle\rangle$ due to $\neg M_2[x^{\pm}\rangle\rangle$, a contradiction. □

Observe that a non-output-determinate STG always has CSC conflicts, as, according to Definition 7.2, 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 is shown in Section 7.3 that such a CSC conflict is *irreducible*.

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:

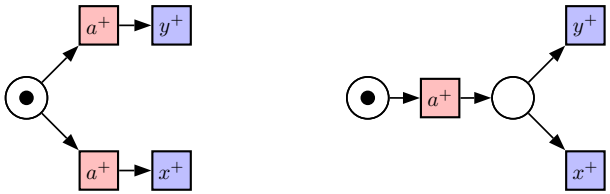


Figure 7.3: Non-output-persistency due to determinisation. A output-persistent but not output-determinate STG (left) and the non-output-persistent STG (due to the choice between the outputs x and y) obtained from it by determinisation (right). Note that determinisation can also result in a choice between an input and an output (this would be the case if y were an input).

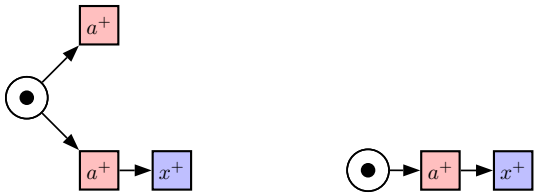


Figure 7.4: Incorrect determinisation: a non-output-determinate STG before (left) and after (right) determinisation. The latter STG, though implementable, is not a correct implementation of the original specification, since it can cause a failure in the environment by producing x when the environment does not expect it.

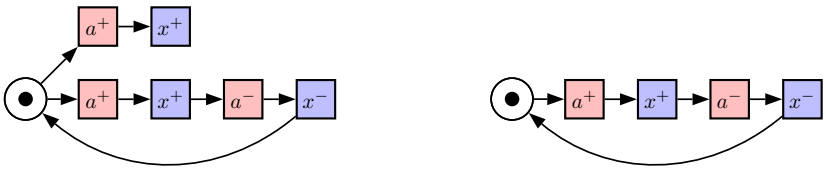


Figure 7.5: Determinisation: an output-determinate STG N with a deadlock (left) and the deadlock-free STG obtained from N by determinisation (right). The latter STG is a correct implementation of N; intuitively, the execution of x^- is correct, since it only occurs when the environment signalled with a^- that the system is in the 'lower' branch of N. The circuit $[x] = a$ implements either of these two STGs.

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 7.4
Let N be output-determinate and C be the deterministic automaton $DA(N)$ obtained by determinisation of the reachability graph of N. Then C is a correct implementation of N.

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

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

The proposed semantics has interesting consequences, in particular, a specification with deadlocks can be 'equivalent' (cf. Definition 7.13) to one without them, as illustrated in Fig. 7.5. Hence, arbitrary language-preserving transformations of output-determinate specifications are allowed, as long as the resulting STG is still output-determinate. That is, there is no need to preserve stronger equivalences such as bisimulation. We present valid transformations in Section 7.2.1.

In view of Semantic Rule 2, one would expect that the notion of correct implementation given in Definition 7.1 can be re-formulated purely in terms of the language if the specification and the implementation are known to be output-determinate. In fact, we generalise the definition to allow a non-deterministic implementation, as long as it is output-determinate.

Definition 7.5 (Trace-Correct Implementation)
An output-determinate STG C is a *trace-correct implementation* of an output-determinate STG N if $In_C \subseteq In_N$, $Out_C = Out_N$, and for every trace w of N the following hold:

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

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

This definition can be viewed as a *denotational* notion of correctness, as opposed to the *operational* one given in Definition 7.1. 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 7.1 if the implementation is deterministic and the specification is output-determinate.

Proposition 7.6 (Justification of the notion of trace-correct implementation)
Let N be an output-determinate STG and C be a deterministic STG such that $In_C \subseteq In_N$ and $Out_C = Out_N$. Then C is a correct implementation of N iff it is a trace-correct implementation of N.

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

At the end of this section we quote the following theorem from [KSV07] about the complexity of checking output-determinacy:

Theorem 7.7
Checking output-determinacy is PSpace-complete for safe and bounded STGs, and it is ExpSpace-hard for unbounded STGs.

7.2 Decomposition into Output-Determinate Components

In this section, we describe how the developed theory of output-determinacy can be applied to derive a new 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.

In contrast to the decomposition algorithm from Section 3.3, this new algorithm allows non-deterministic specifications (if they are output-determinate).

Now, 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 TCOD-transformations described in this chapter. 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 7.1 to families of components, additionally taking care of computation interference as explained after the previous correctness definitions.

Definition 7.8 (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 correct decomposition) of N, if C is a correct implementation of N (cf. Definition 7.1) 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*). △

This definition is equivalent to Definition 3.4 for general N.

(C4) forbids computation interference; here the same arguments and considerations as after Definition 3.4 are valid. In particular, Definition 7.8 is a generalisation of Definition 7.1: 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.

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

Definition 7.9 (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 7.5 hold 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). △

Observe that the components $(C_i)_{i\in I}$ have to be output-determinate rather than their parallel composition C, since they will be synthesised separately and not C.

Again, this definition can be viewed as a *denotational* notion of correctness, as opposed to the *operational* one given in Definition 7.8. The result below shows that this notion is equivalent to Definition 7.8 if the implementation is deterministic and the specification is output-determinate. Proposition 7.6 is obtained as a special case of this theorem by considering $I = \{1\}$ and $C = C_1$.

Theorem 7.10 (Justification of trace-correct distributed implementation)
Let N be an output-determinate STG and let $C = \|_{i\in I} C_i$ 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.

Proof. First we prove that, if $(C_i)_{i\in I}$ is a correct distributed implementation of N, then it is also a trace-correct distributed implementation of N. We consider each requirement of Definition 7.9 in turn, and show that they follow from Definition 7.8.

(TC1) Coincides with (C1).

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

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

Now we show that if C is a trace-correct distributed implementation of N then it is also a correct implementation of N. We consider each requirement in Definition 7.8 in turn, and show that it follows from Definition 7.9.

(C1) Coincides with (TC1).

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

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

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

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

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 (cf. also Section 5.2).

Theorem 7.11 (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 $K = (I \setminus \{i'\}) \cup J$.

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 [SV07]. 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 ∉ J Then, (TC3/C) implies $w|_C x^\pm \in L(C)$, and by (TC2/C) $wx^\pm \in L(N)$.

k ∈ 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 7.12
The relation 'trace-correct implementation' (i.e. the relation $\{(N, N') \mid N'$ is a trace-correct implementation of $N\}$) is a pre-congruence for parallel composition.

Proof. Reflexivity is trivial. Transitivity follows from the above theorem if I and J each consist of a single component. Now precongruence follows when considering that in a parallel composition a component is replaced by another single component. □

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

7.2.1 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 7.11 (for $|J| = 1$) suggests, it is sufficient to preserve trace-correctness. However, it is also 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 notions.

Definition 7.13 ($\approx_{lod}$, $\preceq_{tcod}$ and LOD/TCOD-transformations)
Two STGs N and N' are *LOD-equivalent*, denoted $N \approx_{lod} N'$, if

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

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

- N and N' are both non-output-determinate, or
- N' is a trace-correct implementation of N and both are output-determinate.

An STG transformation is an *LOD/TCOD-transformation* if the original and the transformed STG are LOD-equivalent / in the TCOD relation.

△

Obviously, 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. 7.6. Moreover, any transformation preserving the language and output-determinacy can be made into an LOD-transformation if its domain is restricted to output-determinate systems.

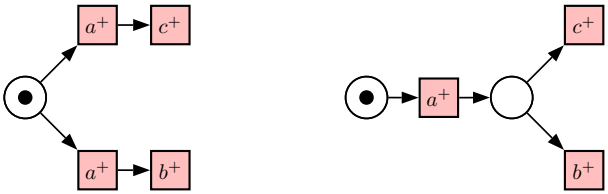


Figure 7.6: Two LOD-equivalent STGs which are not bisimilar.

Below we list the TCOD-transformations which will be used for our decomposition algorithm. For one of the transformations and for further use, we first introduce some notions.

Definition 7.14
For transitions t, t' of some STG, t is a (syntactic) *trigger of* t' or *triggers* t' if $t^\bullet \cap {}^\bullet t' \neq \emptyset$. A λ-transition t is a *weak trigger* of t', if it triggers t' or another weak trigger of t'. A transition t with $l(t) \neq \lambda$ is a *signal trigger* of t', if it triggers t' or a weak trigger of t'.

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

List of 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 (Definition 7.16). This operation was not used in the context of decomposition before; it is discussed below.

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 an 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 an LOD-transformation. Indeed, 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.

Theorem 7.15
If N' is obtained from some STG N by LOD-SecTC2 or SecTC2', then N and N' are LOD-equivalent.

Proof. A secure contraction gives a language-equivalent result in any case by Theorem 3.6.

Now, we consider an output-determinate N and show that N' is also output-determinate. If $M_{N'}[w\rangle\rangle M_1'[x^{\pm}\rangle\rangle$ and $M_{N'}[w\rangle\rangle M_2'$ ($w \in (Sig^{\pm})^*$, $x \in Out$), then $M_N[w\rangle\rangle M_1$ and $M_N[w\rangle\rangle M_2$ with $(M_1', M_1), (M_2', M_2) \in \mathcal{S}'$ for the ready simulation $\mathcal{S}'$ of Theorem 3.6(1). Furthermore, $M_1[x^{\pm}\rangle\rangle$ due to simulation, $M_2[x^{\pm}\rangle\rangle$ due to output-determinacy, and $M_2'[x^{\pm}\rangle\rangle$ 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 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 3.6(1); to get a result on the level of transitions, observe that this relation also is a simulation if the labelling of N is λ for t and the identity otherwise. This consideration implies that e.g. u is simulated by $u\big|_{-t}$, obtained by deleting all occurrences of t in u. Thus, we get $M_{N'}[u\big|_{-t}\rangle[x^{\pm}\rangle\rangle$ and $M_{N'}[v\big|_{-t}\rangle M_1'$.

Since N' is output-determinate and $\bar{l}(u\big|_{-t}) = \bar{l}(v\big|_{-t})$, we have $M_1'[x^{\pm}\rangle\rangle$. Therefore, we can take some $t' \in \overline{T}$ and a minimal $w \in \overline{T}^*$ such that $M_1'[wt'\rangle$, $\bar{l}(t') = x^{\pm}$ and $\bar{l}(w) = \lambda$. 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 $w'\big|_{-t} = wt'$. As induction base, we have $M_1[\lambda\rangle$. So assume $M_1[w''\rangle M$ and $M_1'[w''\big|_{-t}\rangle M'$, where $w''\big|_{-t}$ is a proper prefix of wt' with $M_1'[w''|_{-t}\rangle M'$ 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 $M'[t_1\rangle$, that M and 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 $M(p) + M(p_1) = M'((p, p_1)) \geq \overline{W}((p, p_1), t_1) = W(p, t_1) + W(p_1, t_1)$ (where the inequality follows from $M'[t_1\rangle$), 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 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, $M[t^{m_1} t_1\rangle$. □

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.

Definition 7.16 (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 syntactical 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. Figure 7.7):

- $T' = T$ 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 $t \in T'$,
  $W'(p_1^a, t_1) = W'(p_1^b, t_2) = W'(t_1, p_3^a) = W'(t_2, p_3^b) = 1$,
  $W'(t, p_1^a) = W'(t, p_1^b) = W(t, p_1)$ 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 $M'(p_3^a) = M'(p_3^b) = 0$.

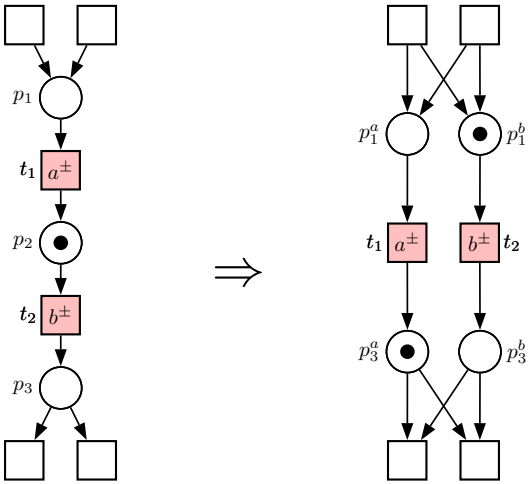


Figure 7.7: Increasing input concurrency for the case $M(p_2) = 1$.

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.) △

Theorem 7.17 (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' preserves consistency of N.

(6) 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 $M'[t\rangle M_1'$. By definition of W', $M_1'|_{P \cap P'} = M_1|_{P \cap P'}$. If $p_1 \in t^\bullet$, observe that $M(p_1) = M(p_2) = M(p_3) = 0$ 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 thus, $M_1' = iic(M_1)$.
- $t \in {p_3}^\bullet$: 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 $M[t_1 t_2\rangle M_2$, 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 sv be the consistent state encoding of N. We define a state encoding sv of N' as follows: if $M' \in \mathcal{M}$, there is a (unique) marking M of N with $(M, M') \in \mathcal{S}$, and $sv'_{M'} = sv_M$. If $M'_1 \in \mathcal{M}'$, there is a (unique) marking $M' \in \mathcal{M}$ with $M'[t_2\rangle M'_1$, and $sv_{M'_1} = sv'_{M'} \oplus t_2$, where the latter denotes the state vector obtained from $sv'_{M'}$, either by adding 1 to the signal s if $l(t_2) = s^+$, or by subtracting 1 if $l(t_2) = s^-$.

Let $M'[t\rangle M'_1$ be a firing in N', such that $M' \in \mathcal{M}$ due to $(M, M') \in \mathcal{S}$. For $t \neq t_2$ or $M'(p_1^a) = 0$, the consistency conditions for sv' are fulfilled due to (1).

So let $M'[t_2\rangle M'_1$ with $M'(p_1^a) = 1$. In this case, $M'_1 \in \mathcal{M}'$ and $sv'_{M'_1} = sv'_{M'} \oplus t_2$ by definition of sv', and the consistency condition is trivially true.

Now, let $M'_1[t_1\rangle M'_2$. Then there is a marking M_2 of N as in the proof of (2) with $M[t_1t_2\rangle M_2$ such that $(M_2, M'_2) \in \mathcal{S}$. Therefore, $sv'_{M'_2} = sv_{M_2} = sv_M \oplus t_1 \oplus t_2 = sv_M \oplus t_2 \oplus t_1 = sv_{M'} \oplus t_2 \oplus t_1 = sv_{M'_1} \oplus t_1$.

Finally, let $M'_1[t\rangle M'_3$ with $t \neq t_1$. Then, there are markings M'_4 and M'_3 of N' as in the proof of (2) with $M'[t\rangle M'_4[M'_3\rangle$. Now, a similar argumentation as above shows that $sv_{M'_3} = sv_{M'_1} \oplus t$.

(6) 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 $M[x\pm\rangle\rangle$, which proves (TC2). □

Theorem 7.17(4,6) shows that IIC is indeed a TCOD-transformation. Furthermore, this result can easily be extended to more general cases:

- Three or more sequential transitions $t_1, \ldots, t_n$ (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 more complicated. In the last case, the proof stays the same since it never uses 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 7.8 demonstrates.

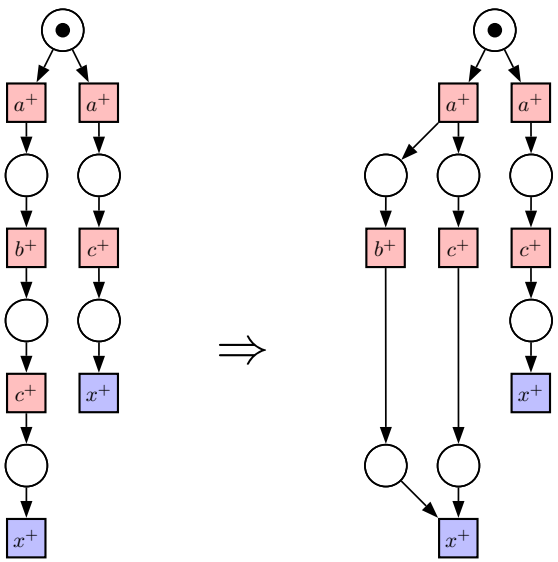


Figure 7.8: Counterexample for application of IIC to an output-determinate but non-deterministic STG. The STG on the left hand side is output-determinate, but the STG resulting from applying IIC for b^+ and c^+ is not, i.e IIC is not a TCOD transformation in this case.

7.2.2 New Algorithm

The new decomposition algorithm works nearly as the old one described in Section 3.3. There are only a few modifications:

- The specification does not have to be deterministic any more.
- The condition (F2) for a feasible partition is modified such that the signal trigger of an output have to be included in the component.

  (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 initial components are built as usual.
- Every TCOD-operation can be applied.
- Eventually, check each component for output-determinacy. If the check fails, perform backtracking for some lambdarised signal or, if no lambdarised signal is left, report that N is not output-determinate. Otherwise, the component is constructed.

Observe that a non-output-determinate STG cannot be synthesised, thus checking output-determinacy can be omitted in principle. However, since the final components are (hopefully) small, we advise to perform this check anyway, since synthesis can also fail for other reasons.

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.

The algorithm of this paper is a generalisation of the decomposition algorithm of Section 3.3, 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 lambdarised 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 weight greater one. Since backtracking applies to all transitions of a signal, one had to delambdarise 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. Now, we have the chance to avoid this, which is an important contribution.

If a transition contraction generates a new dynamic auto-conflict, then – as explained in Section 3.3.2 – 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 if the latter is indeed the case, one has 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 Section 3.3 is guaranteed to find a correct decomposition without any final check.

When using the risky strategy, 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 the new algorithm of this section: in the end, components had to be checked for determinism, which is more restrictive than our check. The experience is that the hope is most often in vain, cf. Section 6.7.

Consequently, a *conservative strategy* is recommended for the new algorithm: whenever the contraction of a dummy 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, where backtracking is not possible.

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

Before we present the correctness proof, observe that notions like signal trigger and weak syntactic conflict (Definition 7.14) 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 component-dummies as well as spec-dummies.

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

Lemma 7.18
Let N be an STG with an initial decomposition $(C_i)_{i\in I}$ where all components are output-determinate. Then $(C_i)_{i\in I}$ is a trace-correct distributed implementation of N.

Proof. (TC1) Let $w \in L(N)$ due to $u \in T^*$. Then, for one transition of u after the other, we can fire all copies of the respective transition in the C_i. In more detail, all copies with label not equal to λ are synchronised in the parallel composition and fire as one transition; the other copies fire one after the other. This shows that $w\big|_C \in L(C)$.
(TC2) & (TC3) Again, let $w \in L(N)$ due to $u \in T^*$. To show (TC3), consider $j \in I$ such that $x \in Out_j$ and $w\big|_{C_j} x^{\pm} \in L(C_j)$ due to the firing sequence vt with $l(t) = x^{\pm}$. Since $l_j(v) = w\big|_{C_j} = l_j(u)$ and C_j is output-determinate, we have a firing sequence $uu't'$ of C_j with $l_j(u') = \lambda$ and $l_j(t') = x^{\pm}$; choose such a u' with minimal length. By minimality, each transition in u' triggers a succeeding transition in $u't'$; thus, if u' contained a lambdarised transition, we could consider the last one, which would be a signal trigger of t', a contradiction to (F2) for C_j. We conclude that all transitions in u' are spec-dummies. Thus, firing $uu't'$ in all C_i as in the first part of this proof, we get that $w\big|_C x^{\pm}$ is a trace of C.
Whenever $w\big|_C x^{\pm}$ is a trace of C, we have $w\big|_{C_j} x^{\pm} \in L(C_j)$. So from the above argument, we also see that (TC2) holds since we can fire $uu't'$ in N as well, showing $wx^{\pm} \in L(N)$. □

Lemma 7.19
Let N be a non-output-determinate STG with an initial decomposition $(C_i)_{i\in I}$. Then some C_i is not output-determinate.

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

Theorem 7.20
Consider the application of the decomposition algorithm to an STG N.

(1) 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 even a correct distributed implementation of N.

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

(3) If only the TCOD-transformations from the list in Subsection 7.2.1 are applied, the algorithm terminates.

Proof. (1) 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 7.19, 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 initial components are output-determinate; hence, by Lemma 7.18 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 7.11 then implies also that the set of final components is a trace-correct distributed implementation of N.

Also determinisation of the C_i is an TCOD-transformation, and the third claim about the deterministic automatons $(DA(C_i))_{i\in I}$ follows from Theorem 7.10.

(2) The algorithm reports that N is non-output-determinate only if there is some component without hidden signals which is non-output-determinate. In this case, the respective initial component is also non-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.

(3) This is a result from [VK06]: backtracking delambdarises a lambdarised 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. □

It should also be noted that for a consistent N only consistent components are produced, cf. Section 3.5 and Theorem 7.11(5).

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]. 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 operations 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.

7.3 Output-Determinacy and Internal Signals

In this chapter, we considered only STGs without internal signals so far. Usually, such signals are introduced automatically into the STG during the synthesis process, mainly in order to resolve CSC conflicts, but also to perform logic decomposition (i.e. splitting large gates into smaller ones) or – as a recent application – to resolve CSC during decomposition [WW07].

There are several possible interpretations for internal signals, depending on the role the STG 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 circuit is synthesised. Hence, the 'distance' from the STG to the circuit 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 ignored by the environment anyway.) 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 a non-output-determinate STG is implementable by a deterministic one, as illustrated in Fig. 7.9. On the other hand, the proposed treatment allows us to lift Proposition 7.3 to the case of STGs with internal signals, stating that only output-determinate STGs can be deterministically implemented.

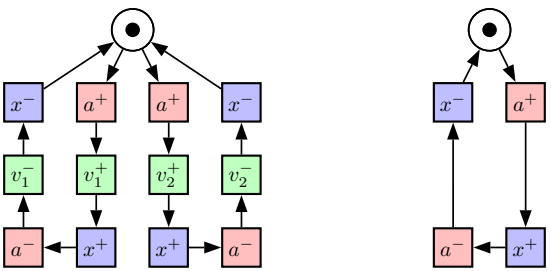


Figure 7.9: Counterexample: the non-output-determinate STG on the left with internal signals v_1, v_2 has the same external behaviour as the right STG.

Now, we generalise the notion of correct implementation given in Definition 7.1 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 7.21 (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 $v_C \in (Int_C^{\pm})^*$.

△

In this definition, the items (IC1)/(IC2)/(IC3) correspond to (C1)/(C2)/(C3) from Definition 7.1 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, in particular, 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 the discussion after Definition 5.2.

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: a non-output-determinate specification cannot be implemented by a deterministic STG with internal signals. Thus, behaviour-preserving insertion of internal signals into a non-output-determinate STG always results in a non-output-determinate STG, and the latter still has CSC conflicts. The result below is an extension of Proposition 7.3 to implementations with internal signals.

Proposition 7.22
Let N be an STG without internal signals and C (with internal signals) be a correct implementation of N. Then N is output-determinate.

Proof. For $x \in Out_N$, let $M_N[w\rangle\rangle M_1[x^{\pm}\rangle\rangle$ and $M_N[w\rangle\rangle M_2$. Then, by (IC1) of Definition 7.21, 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), $M_2[x^{\pm}\rangle\rangle$. □

7.4 Results

As described in the previous sections, it is now possible to leave λ-transitions in the final components as long as they are output-determinate. Moreover, the new approach can also be used to speed up existing decomposition strategies, in particular Tree (Section 6.4) (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.

In the new approach (TreeNew), such postponing of signals can be avoided 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 not all transitions are contractible due to technical reasons only (e.g. due to a violation of safeness-preservation), no backtracking is performed and the remaining non-contractible transitions are simply left as dummies in the intermediate STG. As mentioned above, most of them will be contracted at later stages, and otherwise these dummies will remain in the final component.

We applied TreeNew to the `seqpartree.x` series from the previous chapter; again the initial partition was chosen in such a way that each component of the decomposition corresponds to one handshake component.

We applied four variants of tree decomposition to these benchmarks, as well as standalone 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 performed on a PC with Pentium 4 HT/3GHz processor and 2GB RAM.

TreeOld is compared with TreeNew for ordinary contractions as described in Subsection 3.3.2 as well as for safeness-preserving contractions, see Section 6.6.2 Essentially, the preservation of safeness is another condition which can prevent some

	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 7.1: 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.

contractions and thus increases the runtime. This resulted in the mentioned four series of experiments, see Table 7.1.

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, non-output-determinism is a special case of a CSC conflict, *which cannot be resolved* as discussed in the previous section. Hence, the resulting components are indeed output-determinate, and Theorem 7.20(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 7.1.

The synthesis with stand-alone Petrify or Mpsat has not terminated within 12 hours, even for SeqParTree-05, as the corresponding STGs are very large. We consider it as a notable achievement that the proposed approach could synthesise them so quickly – e.g. `seqpartree.05` 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.

Chapter 8

DesiJ – A Tool for STG-Decomposition

Premature optimization is the root of all evil
– or at least most of it – in programming.
Donald Knuth, 1974

In this chapter, the implementation DesiJ (decomposition Java) of the decomposition algorithm is described. Based on [VW02,VK06], B. Kangsah implemented a prototype version Desi in C, which covered only the Basic strategy. Since this implementation was simply not extendable, we decided to start a new implementation from scratch. This time, Java was chosen instead of C++ (or even C as for Desi), in order to focus on the implementation itself instead of technical details of the programming language, as this often happens for C/C++. Furthermore, performance was not the prime objective for the following reasons:

- The decomposition algorithm works only structurally, i.e. it is explicitly avoided to perform expensive operations in particular building reachability graphs.
- The main focus was the easy extensibility with the new features developed during the STG decomposition project.
- Additionally, the performance of Java is not so bad compared to C/C++ anymore, due to the availability of Java virtual machines with just-in-time-compiling.

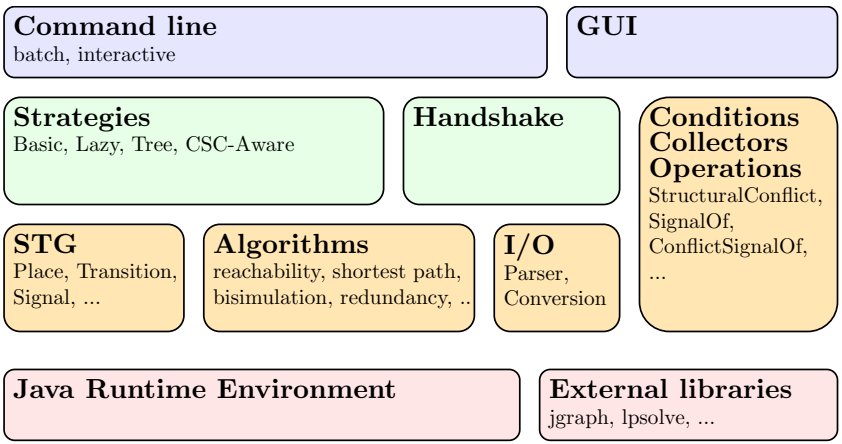


Figure 8.1: Structure of DesiJ

To guarantee the extensibility, DesiJ was developed in three layers, based on the Java runtime environment (JRE) and some external libraries, see Figure 8.1:

- The STG layer provides basic functionality for STG handling independent of decomposition.
- The decomposition layer implements the various decomposition strategies and provides features for the generation and handling of handshakes circuits (cf. also Chapter 9)
- The interface layer provides three access possibilities for the implemented functionality: a batch command line mode, which is the most powerful mode with the most options, an interactive command line mode and a graphical user interface for decomposition. The features of the command line mode are presented in Section 8.2.
- The Conditions, Collectors and Operations block is something special: these operations are independent of decomposition, but provide useful features for it which belong to the STG layer on a functional level, nevertheless. They are described in Section 8.1.3.
- Additionally, there are independent blocks with prototype implementations of new ideas and concepts (not shown).

This chapter is organised as follows: in the first section, some implementation issues are discussed and justified. In Section 8.2 an overview of the implemented functionality of DesiJ is given in form of its commented command line options and parameters.

8.1 Implementation

8.1.1 STG Undo Implementation

In the first implementation of the decomposition algorithm, backtracking was realised by generating copies of the respective component STGs. This is inefficient for LazyBack, Tree and Aggregation, because between two savepoints, decomposition-tree nodes resp. often only small parts of a component are modified. Undoing few changes is clearly more efficient than to copy all the unchanged parts, especially for a large STG. In Basic and Reordering, backtracking means to restore the initial component. If this is performed for a small intermediate STG, copying might be more efficient: in particular, during reduction a lot of redundant places are produced and deleted afterwards. When undoing, the places are first introduced and then deleted again. Nevertheless, it turned out that even these strategies benefit from undoing, see Section 6.7.

Furthermore, copying the STGs increases the memory usage very much, because there are a lot of intermediate results which have to be stored – this is especially important for Tree, where all components are created at the same time and even more intermediate results have to be stored.

Instead, in DesiJ an undo mechanism is implemented which can restore previous versions of an STG very efficiently. Functionally, it works as a stack like in most applications (e.g. text or image editors), i.e. only the latest performed operation can be undone.

There are only three supported operations: adding/removing nodes with their incident arcs and changing the signature. All high-level operations like transition contraction are reduced to these ones; at the moment, there is no operation for changing arc weights, since it is not needed so far.[1]

For every change in an STG the corresponding undo operation (`UndoOperation`) is generated and pushed on the undo stack (`STG.undoStack`). When an undo is performed, the last operation is removed from the undo stack and is applied, i.e. the corresponding change is undone. (`UndoOperation.apply()`).

[1] Arcs with a weight greater than 1 can be produced during decomposition, but their weight does not change afterwards.

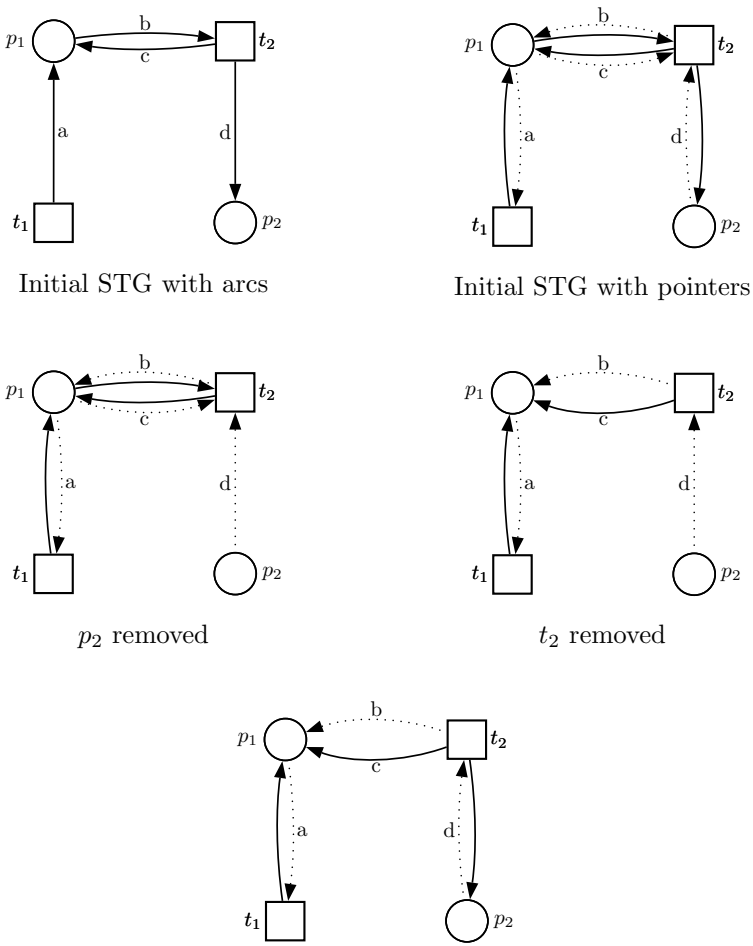


Figure 8.2: STG undo example. The internal representation (upper right) of the initial STG (upper left) contains two pointer for every arc – a forward pointer and a backward one (dotted). Here, the arc labels denote the arc names.

To understand the undo mechanism, we will first have a short look at the data structure of an STG in DesiJ, cf. also the first row in Figure 8.2. Essentially, an STG is a graph (and the undo stack can also applied to them in principle), and it is stored with adjacency lists kept in the nodes. The nodes are double-linked, i.e. for every arc from node x to node y, two pointers are stored: one *forward pointer* in x pointing to y, and one *backward pointer* in y pointing to x.[2] Furthermore, all nodes of a net are stored in a list.

Undoing the adding of a node is quite easy: the respective node and incident arcs are simply removed from the STG. Undoing the removal of a node x is a bit more complicated and harnesses that the nodes are double-linked: if x should be removed from a net, it is not deleted completely, but only removed from the list of nodes and all pointers *pointing to* x are deleted in the respective nodes, i.e. the STG 'forgets' about x. On the other hand, the *outgoing pointers* of x are kept, i.e. x itself still 'knows' about its former connections.

For an example, have a look at Figure 8.2. The initial STG (upper left) contains four nodes and four arcs (a, b, c and d). In the internal representation (upper right), each of these arcs is represented by a forward pointer and a backward pointer (dotted). If the nodes p_2 and t_2 are removed in this order, we get the internal representations in the middle row. As one can see, only the respective outgoing pointers remain. If the operations are undone, the states are encountered in the reverse order.

It depends on the direction of an arc which of the corresponding pointers is deleted, e.g. in the STG in Figure 8.2, t_2 lies on a loop with p_1 formed by the arcs b and c. When t_2 is deleted, the backward pointer of c but the forward pointer of b is deleted in p_1. If the deletion of x should be undone, all ingoing pointers to x are restored in the respective nodes from the outgoing counterparts in x, and x is added again to the list of nodes.

In principle, it is possible to apply undo operations out of order as in the last row, where p_2 was added before t_2. Observe, that this STG is not well-formed: the backward pointer of arc d is *dangling*, i.e. it points to a node which is not contained in the list of nodes. Therefore, only STGs corresponding to a proper stack state[3] are guaranteed to be well-formed, but it is also possible that independent operations are undone, resulting in well-formed data structures, e.g. if the respective nodes are not adjacent. Undoing out-of-order might allow to perform undo operations concurrently on multi-processor systems.

[2] Java does not have pointers in the same sense as C++, but all variables in Java are actually references and therefore some kind of pointer, although there is no pointer arithmetic.

[3] We can imagine that non-top-elements removed from the stack are replaced by dummy elements, which will be removed automatically if they become the top element. Then, a stack is proper if it contains no dummy elements, and this is exactly the case if this state was encountered previously when applying the operations which are now undone.

The undo operations for the modification of the signature just keep track of the old state and restore it if needed. For example, the STGs keeps record of each occurring signal (also for lambdarised signals the signal name is kept – the signal type is just changed to dummy). If the last transition of a signal is deleted, this signal is removed from the list of signals and re-inserted if the deletion of the transition is undone.

Additional features of the undo stack are *combined undo operations* which encapsulate associated modifications; at the moment, this is only used for transition contractions. Furthermore, one can push special parametrised *undo markers* onto the stack. They are used to simplify backtracking: the Basic strategy puts a marker on the stack before the reduction starts. If it is necessary to backtrack, the method `STG.undoToMarker(Object)` is called to undo all operations performed after the respective marker was added. Different strategies use different markers, which makes it possible that the other strategies can use Basic to perform reduction in an intermediate STG: Basic only undoes its own changes when backtracking, and other strategies can undo to their own markers ignoring the Basic marker.

A typical stack for Tree might look as in Figure 8.3 (undo markers are shaded). At the bottom of the stack, there is a 'entered node' marker, which corresponds to the begin of the decomposition algorithm when the root node of the decomposition tree is entered with the specification STG. Then Basic is performed for the signals to be contracted in this node. A 'basic marker' is set to enable backtracking to this point. As one can see, directly on top of this marker there is another 'entered node' marker, i.e. in the root node no contractions took place. Then, the first real change happened, viz. the type of some signals was changed to dummy (i.e. they were lambdarised), and the contractions of the corresponding transitions by Basic took place. Each undo operation encapsulating a contraction is built from several smaller undo operations as described above: first the new combined places are added, then the old ones are removed, then the signature is updated (it is possible that the transition was the last of a signal, and finally the contracted transition is removed. In the lower contraction, the respective transition has exactly one place in its pre- and postset; in the upper contraction, both, the preset and the postset contains two places.

8.1.2 Calculation of Decomposition Trees

For the decomposition strategy Tree (see Section 6.4), it is needed to precalculate a decomposition tree which is the plan for the decomposition process. Every node stores a set of signals which should be lambdarised and contracted when entering it with an STG, and decomposition starts at the root node with the initial STG. Therefore, every leaf corresponds to a final component, in which all signals on the path from the root to this leaf are contracted.

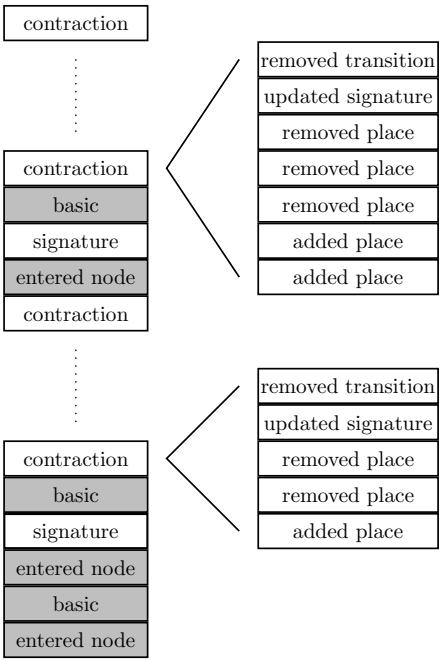


Figure 8.3: Undo stack during TREE (growing to the top). Shaded entries are undo markers, contractions are combined undo operations consisting of several operations.

Since the purpose of this tree is to minimise the absolute number of contractions while generating all components, we consider a tree as optimal if the overall number of signals in the nodes is minimal. This is a slightly relaxed assumption, since the number of transitions is not the same for each signal, but in practical STGs most signals have only two transitions. Furthermore, as it was mentioned in Section 6.4, the tree will be changed during decomposition due to postponing contractions.

In general calculating optimal decomposition trees is NP-complete (see [KK01], decomposition trees are called preset trees there). Also in [KK01], a heuristic algorithm is given, which performs reasonably well: it starts with a set of trees – one for each component, containing only one node with all signals to be contracted in the respective component. Then the algorithm works bottom-up, combining root nodes which share the most signals, until only one tree remains.

For these calculations, a data structure which is a combination of an associative array and a sorted tree is used for storing all pairs of root nodes together with the

intersection of their signal sets. It supports the following operations (n is the number of signals):

- Given two sets of signals, adding the entry for the intersection of them is done in $O(n)$: $O(n)$ for generating the intersection and $O(\log n)$ for adding.
- Retrieving the largest intersection is done in $O(1)$.
- Given two sets of signals, removing all corresponding entries and adding new entries for their intersection is done in $O(n^2)$.

Initially, this data structure contains all possible intersections between the signals of all root nodes. If n is the number of signals of the specification, setting this up takes $O(n^3)$ time, $O(n^2)$ entries of $O(n)$ size. Obviously, also the initial memory usage is in $O(n^3)$. If one wants to generate the finest partition, the memory usage as well as the runtime is in $\Omega(n^3)$.

The main loop iteratively retrieves the largest intersection of two signal sets, combines the corresponding trees T_1 and T_2 and updates the data structure. T_1 and T_2 are combined by generating a new tree node T which stores the intersection and which has T_1 and T_2 as children. Furthermore, the intersection is removed from the signal sets stored in T_1 and T_2. Since in every step the number of trees is decreased by 1, the main loop is executed exactly $n - 1$ times resulting in a $O(n^3)$ overall runtime.

To make the implementation more efficient, the signal sets are stored as fixed length bitsets, which improves the runtime and memory usage by a constant factor. If this is still not efficient enough, the algorithm can be partially randomised: the set of leafs is (randomly) partitioned into sets of a fixed size and the presented bottom-up algorithm is applied to each of them separately, resulting in a number of preset trees. The root nodes of these trees are considered as leafs for a new iteration and so on until only one tree remains.

For a large STG like `seqpartree.10` (cf. Section 7.4), the decomposition tree can be calculated in less than a minute with the partially randomised version; the resulting tree reduces the number of contractions to be performed with Tree to about 15% compared to Basic, if backtracking is not considered. In random benchmarks which mimic the characteristics of STG decomposition trees[4], the bottom up algorithm is about 2 percentage points better than the partially randomised version. The completely random strategy, which combines in each step two arbitrary trees, is of course even more faster – runtime and memory usage are in $O(n^2)$ – and reduces the number of contractions to about 20%. This algorithm will possibly be needed for larger specifications.

[4]The size of the initial sets as well as their number is related to each other as it is the case for STGs, where e.g. each initial set contains nearly all signals.

8.1.3 Conditions, Collectors and Operations

The condition/collector/operation concept was introduced to enable the fast (sometimes prototype) implementation of the necessary algorithms.

Conditions (`stg.traversal.Condition`) check if a given object fulfils some properties, for instance, if the signal of a transition is in a given set, or if a node is the child of another node. Furthermore, there are special classes which allow to combine predefined conditions to more complex conditions: the `NotCondition` just negates a condition, the class `MultiCondition` combines a set of conditions with AND, OR or XOR.

Collectors (`stg.traversal.Collector`) derive certain information from objects, e.g. the children or syntactical triggers of a node. An overview of the currently implemented conditions and collectors can be found in Tables 8.1 and 8.2. Additionally, operations (`stg.traversal.Operation`) can perform certain tasks for all elements of a given set; there are no predefined operations.

In the following, `List<T>` denotes a list with elements of Type `T`, `Condition<T>` denotes a condition which can be applied to objects of type `T` and `Collector<T,R>` denotes an operation which is applied to objects of type `T` and returns objects of type `R`. A `Collection` is the most general superclass of the `List` class (except of the `Object` class itself); it defines the basic interface for representing a bunch of objects.

Although each condition or collector could be used stand-alone or for just one object, they are usually used in combination with one of the following methods.

- `List<T> getElements(Collection<T> c, Condition<T> cond)`

  Returns the elements from `c` for which `cond` is true.

- ```
  List<R> collect(Collection<T> c, Condition<T> cond,
                      Collector<T,R> col)
  ```

  Returns the results of `cond` for the elements of `c` for which `cond` is true.

- ```
  void modify(Collection<T> c, Condition<T> cond,
              Operation<T> op)
  ```

  Applies the operation `op` to all elements of `c` for which `cond` is true.

In some cases, the usage of a condition or collector was replaced with a specialised implementation, either for efficiency or due to growing complexity. In particular, the `Contractible` condition was moved to `STG.isContractable(STG stg, Transition transition)`:

checking if a contraction is applicable and to determine the exact reason when this is not the case, has become quite complex, since there are several optional reasons for backtracking, e.g. safeness-preserving contractions.

In general, during the development of DesiJ, the condition/collector/operation concept turned out to be very useful. It allows to implement pseudo-code style descriptions of an algorithm in a very straightforward way, because it separates different layers of the implementation from each other.

8.1.4 Verification of the Implementation

In most cases, scientific programming is inherently more complex than business programming: the data structures are rather complex and the algorithms are not well-known but newly developed.

Therefore, it is even more important to ensure the correct functionality of the developed software. In particular, DesiJ works on large input files for which the results cannot be checked by hand. While this is possible for small benchmarks, it is easily possible that new effects and errors arise only for large STGs.

To guarantee[5] the correctness of the decomposition algorithms, the following approach was chosen: an independent package of DesiJ tries to find a proper STG-bisimulation for an STG and its decomposition. If this is successful for a sufficient number of runs of several benchmarks (in particular for large ones), the implementation of the respective algorithm is considered correct. This introduces redundancy: since the two parts of DesiJ only share the low level implementations of the `STG` layer, it is rather unlikely that both parts work incorrect in such a way that an incorrect decomposition is reported as correct.

The correctness checking package is divided into two parts:

- A general-purpose simulation engine (`Simulation`), which tries to build a relation between the states of two systems according to certain rules.

- A set of `RelationPropagator`s which define the corresponding rules. Rules are of the form: *if this element is in the relation, then also this* or *this* or ... *element has to be.* A rule also defines which elements have to be in the relation unconditionally, e.g. the element $(M_N, M_{C'})$ for an STG-bisimulation.

  In particular, the `STGBisimulationPropagator` defines the rules of an STG-bisimulation.

[5] Of course, this is no 100% guarantee in the sense of verification. The implemented algorithms itself are proven correct anyway – this is a check of their correct implementation.

Name	Description	Comb.
`Activated`	If a transition is activated	
`Adjacent(Set<Node> n)`	If a node is adjacent to a node from `n`	•
`All`	Always true	
`ArcWeight(int n)`	If a node is incident to arcs with weight $> n$	
`ChildOf(Node n)`	If a node is a child of `n`	•
`Contractible`	If a transition can be contracted. Not used any more, cf. text.	•
`DuplicateTransition (Transition t)`	If a transition is a duplicate of t	
`EqualTo(Object o)`	If an object equals `o` (in the sense of Java)	
`LoopOnly`	If a node is a loop-only node	
`LoopWith(Node n)`	If a node forms a loop with node `n`	•
`MarkedGraphPlace`	If place has only one incoming and one outcoming arc, both with weight 1	
`MultiNodes(Set<Node> n)`	If a node is contained in `n`	
`NewAutoConflict`	If a transition contraction creates a new autoconflict pair	
`NumberOfChildren(int n)`	If a node has more than `n` children	
`NumberOfParents(int n)`	If a node has more than `n` parents	
`ParentOf(Node n)`	If a node is a parent of `n`	•
`RedundantPlace`	If a place is redundant. This is the most complex condition.	•
`RedundantTransition`	If a transition is redundant	
`SafeContractable`	If a contraction preserves safeness	
`SecureContraction`	If a contraction is secure	•
`SelfTriggeringPlace`	If a place connects two transitions labelled with the same signal	•
signal/signature	Several conditions related to the labelling of transitions	
`StructuralConflict (Transition t)`	If a transition is in structural conflict with `t`	

Table 8.1: Implemented Conditions. The 'Combined' column is marked if the respective condition is a `MultiCondition` or `NotCondition`, or if its implementation mainly uses other conditions. The `All` condition may seem a bit pointless, but it is used if `collect` or `modify` should be applied without any preconditions.

Name	Description
`AutoConflict`	Returns the signals for which a place constitutes a structural auto-conflict
`Children`	Returns the children of a node
`ConflictSignal`	Returns all signals which are in structural conflict for a transition
`Identity`	Returns the object itself
`Marking`	Returns the marking of a place
`NewAutoConflictPair`	Returns all signals for which a contraction will create new structural auto-conflicts
`Parents`	Returns the parents of a node
`Signal`	Returns the signal of a transition
`String`	Returns a string representation of an object
`SignalTrigger`	Returns all weak syntactical triggers of a transition
`SyntacticalTrigger`	Returns all syntactical triggers of a transition

Table 8.2: Implemented Collectors.

8.2 Command Line Options and Parameters

In this section, the various command line options of DesiJ are listed, in order to give an overview of the implemented features and to give an impression of the complexity of the whole system.

There are command line options and command line parameters. The first ones can be given in a long and in a short form. For instance, the help option can be given as `--help` or just as `-h`. The short forms can be combined, i.e. instead of `-m -Z`, just `-mZ` can be used. Furthermore, options have a default value, either true or false, which is switched by the presence of the option. The default value is given in brackets on the right. Parameters are given in the form `parameter=value`. If there is a default value, it is also given in brackets.

The default values for options and parameters are chosen such that in most cases just typing `desij some_stg` results in a fast and sufficient decomposition of `some_stg`.

`operation=[bisim|check|cl|convert|create|decompose|info|info1| info2| killdummies|rg|reddel|show]` (decompose)

Determines the working mode for DesiJ as follows:

`bisim <specification> <components>`

Finds and prints an STG bisimulation between the STG in <specicfication> and the parallel composition of the STGs in the <components> files.

`check <specification> <components>`

Same as for `bisim`, but only checks if a STG-bisimulation exists. It is also faster than `bisim`.

`cl <file1>`

Opens the interactive command line mode.

`convert <file1> <file2> ...`

Converts the input STG to a different format. See `format` parameter for more details.

`create`

Creates a predefined STG model (e.g. `seqpartree.x`). When using this, the `model` parameter is mandatory.

`decompose <file1> <file2> ...`

Decomposes the given STGs. This operation is affected by various other parameters and options.

`info,info1,info2 <file1> <file2> ...`

Prints several information of the given STGs. `info2` gives the most detailed ones.

`killdummies <file1> <file2> ...`

Contracts all dummy transitions, if possible.

`rg <file1> <file2> ...`

Creates the reachability graph.

`reddel <file1> <file2> ...`

Deletes all redundant places.

`show <file1> <file2> ...`

Converts the STG to PS and opens a viewer.

`--help` `-h` (false)

Shows the help message.

`--values` `-V` (false)

Shows the values of all command line options and parameters.

`--stat-server` `-E` (true)

Starts the statistic server, which is a simple TCP/IP server providing information about the decomposition process and allows a basic control over running DesiJ processes.

`--silent` `-Z` (false)

Suppresses the starting message.

`verbose=[0|1|2|3]` (1)

Writes detailed information of what is happening at the moment to the command line. 0: no information, 3: nearly every operation.

`--punf-mpsat-output`	`-m`	`(false)`
Shows PUNF and MPSAT output for debugging.		

`--gui`	`-G`	`(false)`
Starts the graphical user interface.		

`--productive`	`-v`	`(true)`
Several optimisations take place. Essentially, the places are just numbered after contractions. This makes the result more readable and accelerates the computations. Otherwise, a reduction combines the name of the resp. places, which is helpful for debugging.		

Control of the Decomposition Algorithm

`version=[basic|csc-aware|lazy-multi|lazy-single|tree]`	`(csc-aware)`
The decomposition strategy to use.	

`--aggregation`	`-a`	`(false)`
Performs tree aggregation. Works only for tree decomposition and CSC-aware decomposition.		

`--leave-dummies`	`-l`	`(false)`
Do not backtrack for dummy transitions which are not contractible due to structural reasons, cf. Chapter 7.		

`max-csc-backtracking=[1..MAX_INT]`	`(3)`
When CSC aware decomposition is enabled, this value determines how often a CSC-job is pushed upwards.	

`mcs=[1..MAX_INT]`	`(10)`
The maximum number of signals a component can have to perform component aggregation.	

`--recontract`	`-#`	`(true)`
After solving CSC with known signals, the signals not destroying CSC cores are contracted again.		

`--stop-when-backtracking`	`-A`	`(false)`
Stops decomposition if backtracking is about to be performed for the first time. Mainly for debugging.		

`--risky`	`-Y`	`(false)`
When true, structural auto conflicts are ignored.		

`conflict-signal-exception`
Which signals are treated different in case of an auto-conflict (see conflict-strategy). Turns conservative to risky and vice versa.

`--incomplete-partition` `-i` `(false)`
Checks the given partition only for conflicts between outputs and not if it is complete. i.e. all outputs occur within.

`--remove-redundant-transitions` `-T` `(true)`
Redundant transitions are removed during decomposition.

`partition` `(finest)`
Output signals of the components, the inputs are determined according to the requirements of a feasible partition.
For instance, `partition=x:y_z1:z2_ack` will create the three components `(x,y)`, `(z1,z2)` and `(ack)`. `partition=finest` automatically determines the finest possible partition.

`--order-dummy-transitions` `-o` `(true)`
Enables Reordering.

`--postpone-contractions` `-p` `(true)`
If a transition contraction is not possible it is tried later again, as long as there are other contractions possible.

`--forbid-self-triggering` `-s` `(true)`
Consider transitions as non-contractible if their contraction leads to a self-triggering situation and the respective places are not redundant.

`--safe-contractions` `-f` `(true)`
Performs only safeness preserving contractions.

`--safe-contractions-unfolding` `-x` `(true)`
Check safeness preserving contractions with Mpsat.

`deco-tree=[combined|random|top-down]` `(combined)`
The method used to generate the decomposition tree.

`max-unfolding-size=[0..MAX_INT]` `(100)`
The maximum size of an STG (#Transition + #Places) for which properties are checked on the unfolding. See also Section 6.7.

Handling of Redundant Places

When the deletion of redundant places is activated, it is checked for (in this order):

- easy special cases (e.g. a place with empty preset is always redundant)
- loop only places
- duplicate places
- shortcut places
- implicit places (with unfoldings)

The latter two possibilities are only checked if they are enabled (see below), the last one only if the STG is small enough, see also `max-unfolding-size`.

`--remove-redundant-places` `-P` `(true)`
Redundant places are removed during decomposition.

`--shortcutplace` `-u` `(true)`
Check also for shortcut places when looking for redundant places.

`shortcut-length=[2..MAX_INT]` `(MAX_INT)`
Maximal path length for shortcut places.

`--red-unfolding` `-X` `(true)`
Use MPSAT to check for implicit places. When this option is enabled, every implicit place is found.

`--check-red-often` `-O` `(false)`
It is checked for redundant places (and transitions) before every contraction. Reduces performance significantly.

`max-place-increase=[1.0..MAX_FLOAT]` `(1.1)`
If during the contraction of some transitions the number of places exceeds the original number multiplied by this value, redundant places are deleted before it is proceeded with the contraction. Prevents the 'explosion of places' during contraction.

Synthesis of Components

`--synthesis` `-y` `(false)`
When enabled DesiJ tries to synthesise the components with an external tool.

`equations` `(equations)`
The file in which the equations are stored if synthesis is enabled.

`syn-tool=[mpsat|petrify]` (mpsat)
The given tool is used for synthesis of the components.

`syn-param=[arbitrary]`
The following parameters are forwarded to the synthesis tool.

Various Options

`hide`
The set of hidden signals when checking for bisimilarity.

`model=[art|seq|par|multipar|seqpartree|parseqtree]`
When the operation is `create`, this parameter defines the generated STG model.

`--hide-internal-handshakes` `-H` (false)
When creating models out of handshake components, the internal handshakes are hidden, i.e. labelled with lambda in this case.

`--handshake-component-csc` `-c` (false)
When creating models out of handshake components, the components are generated with CSC.

Output Options

`--write-logfile` `-L` (true)
Writes a logfile.

`logfile` (desij.logfile)
Name of the logfile.

`--intermediate-results` `-I` (false)
Intermediate STGs are written to disk. Mainly for debugging.

`format=[g|dot|ps]` (g)
Fileformat of the written STGs. `*.g` is the standard exchange format for STGs, `*.dot` is a dot (from GraphViz) source file for the generation of graphical representations, `*.ps` is a PostScript file.

`--save-all-places` `-S` (false)
When saving in `*.g` format, all places are saved explicitly.

`--reddel-before-save` `-b` `(false)`
Removes redundant places and transitions before an STG is saved. Not needed for decomposition.

`--reachability-graph` `-R` `(false)`
Writes the reachability graph of an STG instead of the STG itself.

`label=`
User defined label/caption for output in graphic formats.

`outfile=`
Name of output file for operations different from `decompose`.

Chapter 9

Conclusion and Future Research

This thesis dealt with the STG decomposition algorithm of [VW02, VK06]. A variety of properties and improvements of this algorithm as well as of related topics from the field of asynchronous circuits and STGs have been presented.

In Chapter 4, it was shown that the STG decomposition algorithm presented in [VW02] is determinate if applied to live marked graphs, a subclass of considerable interest in the area of circuit design. The proof of this result is based on several statements, and only one of them could be shown for general Petri nets. It would be clearly interesting to generalise at least some other partial results to other net classes.

Related to the determinacy result, but also of independent interest, is the conceptionally and algorithmically easy characterisation of redundant places in live marked graphs, for which we provided a new easy proof. Again, we would like to generalise this result; it is clear that in S-Systems [DE95] — which coincide with finite automata — no place can be redundant.[1]

The next more general class are *free-choice nets*, in which two transitions either have the same preset, or their presets are disjoint; clearly, marked graphs are a subclass of free-choice nets. It is still unclear how redundant or implicit places could be characterised structurally for this class, and a determinacy result is rather unlikely since a contraction does in general not preserve the free-choice property.

In Chapter 5, we have generalised the correctness definition of [VW02, VK06], Definition 3.4 resp. to decompositions of STGs with internal signals and proven that speed-independent

[1] In S-Systems, every place must have a transition in its postset. This is allowed for finite automata, and such places/states would be indeed implicit.

CSC-solving as performed e.g. by PETRIFY is correct. We have shown that the new correctness is preserved in a top-down decomposition, and this result has a number of consequences: now we can use step-wise decomposition in the decomposition algorithm of [VW02, VK06] to improve efficiency as it is described in Section 6.6, and we know that this algorithm in combination with speed-independent CSC-solving gives correct results. Applying the correctness definition to compare two STGs, we get an implementation relation, and consequences of our result are that this is a preorder and, with a small restriction, a precongruence for parallel composition, relabelling and hiding.

As another application of the correctness definition, it was shown that a decomposition method based on integer linear programming [CC03] is correct. It remains an open problem whether a related method in [YOM04] is correct: while the first method checks on the original STG to be decomposed whether a set of signals is a CSC-support and in the positive case removes the other signals, the related method removes some signals and checks CSC on the remaining STG; this is in general not sufficient, but it might be sufficient under the specific circumstances of the algorithm in [YOM04].

Finally, we compared our implementation relation with the one derived from the notion of I/O-compatibility in [CC02] and the one defined by Dill [Dil88]. We have shown that our implementation relation is strictly stronger than the latter ones.

While the previous chapters are concerned with theoretical findings, in Chapter 6 practical strategies for the implementation of the decomposition algorithm were introduced. These improvements increased the performance of decomposition very much. Especially TREE and DEMPSY turned out to be excellent strategies for saving runtime and memory.

As mentioned in Section 6.4, the pre-calculated decomposition tree is not necessarily optimal for the final components, since signals might be moved from nodes to their children. Further work in this direction will be to consider the top-down algorithm for building preset trees in [KK01]. This strategy starts at the root node – as the tree decomposition does – and adds branches iteratively to the tree. The idea is to interleave this building process with decomposition itself — including postponing — in order to get a better decomposition tree.

Furthermore, for the time being, DESIJ looks only for so-called shortcut places (cf. also [SVJ05, STC98]), which are a subclass of redundant places. Improving this more algorithmic part of DESIJ would reduce backtracking (since undetected redundant places can prevent secure transition contractions) and therefore improve runtime and quality of the components.

This is partially done with the help of unfoldings, cf. Section 6.6, but only for comparatively small STGs; structural conditions like shortcut places in marked graphs could be more helpful.

The purely structural decomposition approach investigated in this thesis can handle large specifications, but it does not take into account the properties of STGs related to synthesisability, such as the presence of CSC conflicts. In contrast, MPSAT can resolve CSC conflicts and perform logic synthesis, but it is inefficient for large specifications. In Section 6.6,

we demonstrated how these two methods can be combined to synthesise large STGs very efficiently.

One of the main technical contributions of this chapter was to preserve the safeness of the STGs throughout the decomposition, because MPSAT can only deal with safe STGs. This is not just an implementation issue or a compensation for a missing MPSAT feature, but it is also far more efficient than working with non-safe nets, for which unfolding techniques seem to be inefficient. We also showed how dynamic properties like implicitness and auto-conflicts can be checked with unfoldings and how these checks can be combined with cheaper conservative structural conditions.

Further research is required for the calculation of the decomposition tree, the size of which is cubic in the number of signals and exceeds the memory usage for decomposition and synthesis by far. Here, heuristics are needed which explore the trade-off between the quality of the decomposition tree and the amount of memory needed for its calculation.

In Chapter 7 the new notion of output-determinacy is introduced. This concept is a relaxation of determinism and determinacy and fits very well into the speed-independent and STG synthesis context. It also gives a first formal semantics for the use of dummy transitions in STGs.

Based on this, it was also possible to give a simplified correctness notion, an extended algorithm with new reduction operations and a simplified correctness proof thereof. The new algorithm can also decrease runtime because it allows for dummy signals in the components which avoids unnecessary backtracking. Furthermore, increasing input concurrency was applied for the first time in the context of decomposition.

Additionally, all these results and algorithms were implemented in the tool DESIJ, which was presented in the previous chapter.

For every chapter there are some problems and unanswered questions, like the detection of redundant places, the optimisation of the decomposition strategies, the invention of new reduction operations or the handling of internal signals.

While all these problems are of some interest, it is more important that future research helps turning asynchronous design from art to engineering as mentioned in the introduction. One step in this direction is the handling of really large specifications in order to make asynchronous design attractive for industrial applications.

Unfortunately, the synthesis of large specifications – in particular of STGs – suffers from the state space explosion problem which makes synthesis impossible for interesting designs. A major advantage of decomposition is that it makes the synthesis of such large designs possible, as it was shown in Section 6.6 and 7. Improving these aspects of decomposition is essential for the future.

Another way to cope with the state space explosion problem is to use syntax-directed translation. This is essentially the idea behind BALSA [EB02] and TANGRAM as mentioned in Section 2.2.3. This technique, although computationally efficient, often yields circuits with large area and performance overhead compared with their synchronous counterparts. Due

to this, the resulting circuits are highly over-encoded, i.e. they contain many unnecessary state-holding elements.

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

In [CC06] NP-complete ILP-problems are solved for the full specification In contrast, with the presented decomposition algorithm, we follow a more scalable approach, which tries to avoid performing expensive operations (such as resolving encoding conflicts) on the original 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: this conflict was present already in the original STG, or this conflict was introduced by contracting a necessary signal. The technique described in Section 6.6 allows one to check which of these two reasons applies, and in the latter case 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.

Recently, it turned out that in order to decompose handshake STGs properly, it is probably needed to introduce internal communication signals between the components (Chapter 5). First experiments in [WW07] showed that this is indeed possible, but still a lot of research is necessary. It might also help to enhance STGs with timing information, which can be used to steer the decomposition algorithm, as it was proposed in [YMKM05, YM06].

Such improvements of the presented approach to STG decomposition may eventually help to reduce the complexity of large circuits derived from handshake description and to increase their working speed.

To conclude this chapter and the thesis, it was shown that STG decomposition is a useful and extensible concept linked to theoretically interesting questions, and it seems to be possible to combine it with other promising asynchronous design approaches in order to synthesise large real-world circuits.

Bibliography

[Ber87] G. Berthelot. Transformations and decompositions of nets. In *Petri Nets: Central Models and Their Properties*, LNCS 254, pages 359–376. Springer-Verlag, 1987.

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

[Bes87] E. Best. Structure theory of Petri nets: The free choice hiatus. In W. Brauer et al., editors, *Petri Nets: Central Models and Their Properties*, LNCS 254, 168–205. Springer, 1987.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, 1998.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

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

[Cav07] J. Cavanagh. *Sequential Logic, Analysis and Synthesis*. Taylor & Francis, Boca Raton, 2007.

[CC02] J. Carmona and J. Cortadella. Input/output compatibility of reactive systems. In *Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2002.

[CC03] J. Carmona and J. Cortadella. ILP models for the synthesis of asynchronous control circuits. In *Proc. ICCAD'03*, 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.

[CCJS94] J. Campos, J. M. Colom, H. Jungnitz, and M. Silva. Approximate throughput computation of stochastic marked graphs. In *IEEE Transactions on Software Engineering 20*, pages 526–535, 1994.

[Chu87a] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT, 1987.

[Chu87b] T.-A. Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *IEEE Int. Conf. Computer Design ICCD '87*, pages 220–223, 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. Inf. 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-Verlag, 2002.

[DE95] J. Desel and J. Esparza. *Free Choice Petri Nets*. Cambridge University Press, Cambridge, 1995.

[Dil88] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent circuits*. MIT Press, Cambridge, 1988.

[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 Comp. Prog.*, 18:223–245, 1992.

[ERV02] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.

[Esp98] J. Esparza. Decidability and Complexity of Petri Net Problems — an Introduction. In *Lectures on Petri Nets I: Basic Models*, LNCS 1491, pages 374–428. Springer-Verlag, 1998.

[GVC97] F. Garcia-Valles and J.M. Colom. Structural analysis of signal transition graphs. In *Petri Nets in System Engineering*, 1997.

[Huf64] D. A. Huffmann. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964.

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

[Kho07] V. Khomenko. Efficient automatic resolution of encoding conflicts using STG unfoldings. In T. Basten and S. Shukla, editors, *ACSD 2007*, pages 137–146. IEEE Computer Society Press, 2007.

[KK01] V. Khomenko and M. Koutny. Towards an efficient algorithm for unfolding Petri nets. In K.G. Larsen and M. Nielsen, editors, *CONCUR 2001*, LNCS 2154, 2001.

[KKY04] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state coding conflicts in STG unfoldings using SAT. *Fundamenta Informaticae*, 62(2):1–21, 2004.

[KKY06] V. Khomenko, M. Koutny, and A. Yakovlev. Logic synthesis for asynchronous circuits based on Petri net unfoldings and incremental SAT. *Fundamenta Informaticae*, 70(1–2):49–73, 2006.

[KS07] V. Khomenko and M. Schaefer. Combining decomposition and unfolding for STG synthesis. In *ATPN 2007*, 2007.

[KSV07] Victor Khomenko, Mark Schaefer, and Walter Vogler. Output-determinacy and asynchronous circuit synthesis. In *ACSD*, pages 147–156, 2007.

[MB59] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, 1959.

[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.

[Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut fr Instrumentelle Mathematik, 1962.

[Pet81] J.L. Peterson. *Petri Net Theory*. Prentice-Hall, 1981.

[Rei85] W. Reisig. *Petri Nets*. EATCS Monographs on Theoretical Computer Science 4. Springer, 1985.

[Ren95] J. Renegar. Linear programming, complexity theory and elementary functional analysis. *Math. Programming*, 70(3, Ser. A):279–351, 1995.

[Sch86] Alexandeer Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[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 Comp. Soc. Press, 1999.

[STC98] M. Silva, E. Teruel, and J.M. Colom. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. In *Lectures on Petri Nets I; Basic Models*, LNCS 1491, 309–373. Springer, 1998.

[SV07] M. Schaefer and W. Vogler. Component refinement and CSC solving for STG decomposition. *to appear in Theoretical Computer Science*, 2007.

[SVJ05] M. Schaefer, W. Vogler, and P. Jančar. Determinate STG decomposition of marked graphs. In G. Ciardo and P. Darondeau, editors, *ATPN 05*, LNCS 3536, 365–384. Springer, 2005.

[SVWK06] M. Schaefer, W. Vogler, R. Wollowski, and V. Khomenko. Strategies for optimised STG decomposition. In *Proc. ACSD'06*, 2006.

[Ung69] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[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 *Concurrency and Hardware Design*, LNCS 2549, pages 152 – 190. Springer-Verlag, 2002.

[Wen74] Siegfried Wendt. Petri-netze und asynchrone schaltwerke. *Elektronische Rechenanlagen*, 16(6):208–216, 1974.

[Wen77] S. Wendt. Using Petri nets in the design process for interacting asynchronous sequential circuits. In *Proc. IFAC-Symp. on Discrete Systems, Vol.2*, Dresden, 130–138. 1977.

[Wol97] R. Wollowski. *Entwurfsorientierte Petrinetz-Modellierung des Schnittstellen-Sollverhaltens asynchroner Schaltwerksverbünde.* PhD thesis, Uni. Kaiserslautern, FB Elektrotechnik, 1997.

[WW07] D. Wist and R. Wollowski. Avoiding irreducible CSC conflicts in component STGs. In *Proceedings of the 19th UK Asynchronous Forum.* Imperial College London, 2007.

[YKK+96] 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.

[YKKL94] A. Yakovlev, M. Kishinevsky, A. Kondratyev, and L. Lavagno. OR Causality: Modelling and Hardware Implementation. In *Proc. ATPN'94*, LNCS 815, pages 568–587. Springer-Verlag, 1994.

[YM06] Tomohiro Yoneda and Chris J. Myers. Effective contraction of timed stgs for decomposition based timed circuit synthesis. In *ATVA*, pages 229–244, 2006.

[YMKM05] Tomohiro Yoneda, Atsushi Matsumoto, Manabu Kato, and Chris J. Myers. High level synthesis of timed asynchronous circuits. In *ASYNC*, pages 178–189, 2005.

[YOM04] T. Yoneda, H. Onda, and C. Myers. Synthesis of speed independent circuits based on decomposition. In *ASYNC 2004*, pages 135–145. IEEE, 2004.

List of Figures

2.1 Simple circuit and corresponding truth table 16
2.2 Digital circuit classification 17
2.3 Flip-flop core — *NAND*-Flip-flop — operations of the latter 18
2.4 Basic structure of synchronous circuit 19
2.5 C-Element and non-atomic implementation 21
2.6 Setting $c = 0$ for a C-Element 22
2.7 Circuit with possible glitch 23
2.8 VME Bus Controller 24
2.9 8 bit Buffer out of handshake components 28
2.10 Textual description of the 8 Bit Buffer 28

3.1 Producer-consumer Petri net. 33
3.2 Example of a parallel composition 50
3.3 Laws of a circuit algebra 51
3.4 Transitivity of Simulations 67
3.5 Two *lbc* but not *sbc* STGs. The cloud does not contain any $x^{\pm}$ or $y^{\pm}$ labelled transitions. 71
3.6 Level place insertion. 72

4.1 For determinacy of reduction 84
4.2 For determinacy of reduction 86
4.3 For determinacy of reduction 88
4.4 Loop after Contraction 89
4.5 Two redundant places p_1, p_2 with $p_1 \notin Q_2, p_2 \in Q_1$ 91
4.6 Confluence of shortcut place deletion and transition contraction 93

5.1 Example of an event insertion . . . 109
5.2 Counterexample for the proof of Theorem 5.22. . . . 122

6.1 Ordinary Backtracking . . . 126
6.2 Backtracking of LazyMulti . . . 127
6.3 Tree Decomposition . . . 132
6.4 Unfolding prefix of VME bus controller . . . 139
6.5 Outline for CSC-aware decomposition . . . 140
6.6 Examples of non-safeness-preserving contractions . . . 144
6.7 `seqpartree.03` . . . 150

7.1 OR-causality . . . 162
7.2 Non-determinism due to lambdarisation . . . 162
7.3 Non-output-persistency due to determinisation . . . 167
7.4 Incorrect determinisation . . . 167
7.5 Determinisation and deadlocks . . . 167
7.6 Two LOD-equivalent STGs which are not bisimilar. . . . 174
7.7 Increasing input concurrency . . . 177
7.8 Counterexample for IIC . . . 180
7.9 Counterexample for output-determinacy and internal signals . . . 186

8.1 Structure of DesiJ . . . 192
8.2 STG undo example . . . 194
8.3 Undo stack during Tree. . . . 197

List of Tables

4.1 Structures of possible places after two transition contractions 84
4.2 All possible places after two transition contractions 85
4.3 Possible places after two transition contractions 87

6.1 Benchmark STGs 149
6.2 Comparison between undo stack and copying STGs. 151
6.3 Detection of implicit places 152
6.4 Comparison of the decomposition strategies. 153
6.5 Impact of Aggregation on the size and number of the components. 155
6.6 Comparison of methods for preserving CSC. 156
6.7 Results of DeMpSy for large benchmarks. 157

7.1 Results for the handshake benchmarks 189

8.1 Implemented Conditions 201
8.2 Implemented Collectors 202

List of Examples

3.1 Producer-Consumer 34
3.2 Producer-Consumer continued 35
3.3 Producer-Consumer continued 37
3.4 Unfolding Example 41
3.5 C-Element: State Graph and Input-Output Conflict 44
3.6 Reachability Graph of VME Bus Controller 46
3.7 Handshake Paralleliser: CSC Solving 48
3.8 Decomposition of VME Bus Controller 1 56
3.9 Decomposition of VME Bus Controller 2 58
3.10 Transition Contraction 62
3.11 Decomposition of VME Bus Controller 3 64
5.1 VME Bus Controller: CSC solving 112

Index

1-live, 35

active port, 27
adjacency list, 195
alphabet, 35
arbiter, 25, 99
arc, 32
asynchronous circuit, 15, 20
auto-cc-preserving, 124
auto-concurrency, 36, 124
auto-conflict, 36, 124
autonomous circuit, 20

backtracking, **63**, 163, 181, 193, 196
backward pointer, 195
balanced, 38
bipartite, 32
bisimulation, **36**, 54, 122, 168
    weak b., 99
bisumulation
    transition-b., 66
bounded, 33
    safe, 33
burst mode, 26

C-element, 20
circuit algebra, 51, 98
clock, 19
clock cycle, 19
clockless circuits, 15
code change, **45**, 114
collector, 199
combinatorial circuit, 16
combined undo, 196
complementary set, 137
Complete State Coding, *see* CSC
complex gate, 21
component, 53
component-dummy, 59, 182
computation interference, 49, **54**, 55, 98, 170
concurrently enabled, 35
condition, 199
configuration, 137
conflict, 25, 35
conflict pair, 137
conflict pair, new, 61
confluence, 81
conservative strategy, 182
consistency, **43**, 70, 125, 185
    language based, 43
contraction, 61
control path, 212
core, 137
correct decomposition, 53, 97
CSC, 45, 185
CSC conflict, 166, 185
CSC support, 114
CSC-conflict
    irreducible, 47
CSC-job, 138

dead signal, 70
deadlock, 168
deco-dummy, 59
decomposition operations, 55
decomposition tree, 196
delambdarise, 55, 63
delay-insensitive, 29, 175

DesiJ, 12, 191
determinacy, 163
determinate, 75
determinisation, 166
deterministic, 36, 52
diamond property, 82
digital circuit, 15, 16
Dill, 116
distributed implementation, 169, 170
distributed system, 31
double-linked, 195
dummy, 42
    component-dummy, 59
    deco-dummy, 59
    input-dummy, 59
    internal-dummy, 59
    output-dummy, 59
    spec-dummy, 59
dummy transition, 42
dynamic auto-conflict, 182
dynamic conflict, 35

edge, 16
EMI, 20
empty word, 35
enabled
    e. concurrently, 35
    transition, 32
    transition sequence, 33
energy, 19
environment, 42, 96, 164
event insertion, 108
extended duplicate, 76
extended marking equality, 68, 145
external isomorphism, 47
external signal, 42

feedback, 17
feedback loop, 16
final component, 55
fire, 32
firing rule, 33
firing sequence, 33, 65
flip-flop, 17
flow table, 26
forward pointer, 195
free-choice-net, 80, 209
fundamental mode, 26

gate, 15
glitch, 23

handshake channel, 27
handshake circuit, 27
    4-phase, 27
handshake component, 27
hazard, 21
hierarchical decomposition, 95, 100, 172
home state, 33
HS circuit, 27
Huffman mode, 26

I/O-compatibility, 119
IIC, 176
ILP, 13, 39, 114
implementation, 53, 163
    distributed implementation, 169
implicit place, 38, 136
inconsistent, 45
increasing input concurrency, 176
initial component, 55, 59
initial state vector, 70
input burst, 26
input proper, 96
input proper insertion, 109
input signal, 42
input-dummy, 59
input-output mode, 27
interference-free, 107, 108
interleaving semantics, 38
internal signal, 42, 95, **185**
internal-dummy, 59
inverse projection, 138
irreducible, 47, 135, 166
isochronic fork, 25

lambdarise, 59, 161
language, 164

lazy backtracking, 126
level place, 71
live, 35, 70
    signal-live, 70
livelock-free, 116, 119
local confluence, 82
local signal, 42
LOD-operation, 173
logical high, 15
logical low, 15
loop-only place, 76

marked graph, 76
marking, 32
    home state, 33
    initial m., 31
    reachable marking, 33
marking equality, 61
    extended m. e., 68
ME-element, 25, 99
meta-stability, 23
Mpsat, 13

next-state-function, 19
normal form, 82

occur, 32
operating mode, 26
operation, 199
optimistic strategy, 181
OR-causality, 161
output signal, 42
output-determinacy, 163, 165, 181
    complexity of, 169
output-dummy, 59
output-persistency, 25, 166

paralleliser, 48
passive port, 27
Petri net, 31
    labelled Petri net, 35
petrify, 13
place, 31
place projection, 63, 145
port, 27
postponing, 197
postset, 32
preorder, 96
preset, 32
producer-consumer, 34
Punf, 13

race, 17, 23, 25
reachability graph, 38
reachable, 33
    reachable from, 33
    reachable m., 33
ready simulation, 61
reduce, 55
reduction, 55
reduction rule, 81
reduction system, 81
redundancy, 136
redundant place, 38, 136
    reference set, 38
    valuation, 38
redundant transition, 60
reference set, 38
reversible, 33
risky, 64, 148, 182, 205
    optimistic strategy, 181

S-system, 209
safe, 33
safeness-preserving, 136, 141, 188
savepoint, 126
secure contraction, 61, 163
self-triggering, 135
semantics, 164
sequential circuit, 16, 17
shortcut place, 39, 64, 76, 210
SI, 25
signal, 16
    dead s., 70
signal change vector, 141
signal edge, 16, 42
signal insertion, 109
signal transition graph, 40

signal trigger, 181
signal-live, 70
simulation, 36, 65
skew, 25
spec-dummy, 59, 161, 182
specification, 53
speed-independent, 25, 43, 163, 165, 175
state assignment, 43
state duplication, 73
state vector, 43
    initial s. v., 70
STG, 27, 40
    timed, 13
STG-bisimulation, 53, 200
structural auto-conflict, 36, 182
structural auto-conflict, new, 61
structural conflict, 35
structural duplicate, 129
subcomponent, 100
synchronous circuit, 15, 18
syntactical trigger, 59, 114
syntax-directed, 29
synthesis, **12**

T-system, 76
TCOD-transformation, 173
technology mapping, 21
term rewriting, 81
terminating, 81
timed STG, 13, 212
timing assumptions, 24, 97
token, 32
trace, 36, 65, 164
trace structure, 13, 117
    canonical t. s., 117
trace-correct, 168
transition, 31
transition-bisimulation, 66
truth table, 16
type-1 secure, 61
type-2 secure, 61

unbounded gate delay model, 24
undo, 193
    combined u., 196
undo marker, 196
unfolding, 13, 40, 124
    u. prefix, 40
Unique State Coding, 45
Universal Do-Nothing module, 122
USC, 45

valuation, 38
VME, 24

weak bisimulation, 99
weight function, 31
wire, 15
worst-case-delay, 19