Hasso-Plattner-Institut für Softwaresystemtechnik
an der Universität Potsdam

Attacking Complexity in Logic Synthesis of Asynchronous Circuits

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
in der Wissenschaftsdisziplin "Digitaltechnik"

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Dominic Wist

Potsdam, den 13. 05. 2012

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2012/5970/
URN urn:nbn:de:kobv:517-opus-59706
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-59706

Abstract

Most of the microelectronic circuits fabricated today are synchronous, i.e. they are driven by one or several clock signals. Synchronous circuit design faces several fundamental challenges such as high-speed clock distribution, integration of multiple cores operating at different clock rates, reduction of power consumption and dealing with voltage, temperature, manufacturing and runtime variations. Asynchronous or clockless design plays a key role in alleviating these challenges, however the design and test of asynchronous circuits is much more difficult in comparison to their synchronous counterparts.

A driving force for a widespread use of asynchronous technology is the availability of mature EDA (Electronic Design Automation) tools which provide an entire automated design flow starting from an HDL (Hardware Description Language) specification yielding the final circuit layout. Even though there was much progress in developing such EDA tools for asynchronous circuit design during the last two decades, the maturity level as well as the acceptance of them is still not comparable with tools for synchronous circuit design. In particular, logic synthesis (which implies the application of Boolean minimisation techniques) for the entire system's control path can significantly improve the efficiency of the resulting asynchronous implementation, e.g. in terms of chip area and performance. However, logic synthesis, in particular for asynchronous circuits, suffers from complexity problems.

Signal Transitions Graphs (STGs) are labelled Petri nets which are a widely used to specify the interface behaviour of speed independent (SI) circuits – a robust subclass of asynchronous circuits. STG decomposition is a promising approach to tackle complexity problems like state space explosion in logic synthesis of SI circuits. The (structural) decomposition of STGs is guided by a partition of the output signals and generates a usually much smaller component STG for each partition member, i.e. a component STG with a much smaller state space than the initial specification.

However, decomposition can result in component STGs that in isolation have so-called irreducible CSC conflicts (i.e. these components are not SI synthesisable anymore) even

if the specification has none of them. A new approach is presented to avoid such conflicts by introducing internal communication between the components.

So far, STG decompositions are guided by the finest output partitions, i.e. one output per component. However, this might not yield optimal circuit implementations. Efficient heuristics are presented to determine coarser partitions leading to improved circuits in terms of chip area.

For the new algorithms correctness proofs are given and their implementations are incorporated into the decomposition tool DesiJ. The presented techniques are successfully applied to some benchmarks – including 'real-life' specifications arising in the context of control resynthesis – which delivered promising results.

Acknowledgements

I am thankful for finishing this thesis after a long journey. My thanks go to all the people who helped me to enjoy the traveling on this sometimes 'bumpy road'.

I would like to express my gratitude to Ralf Wollowski for providing a lot of inspirations not only for my research, but also for my personal development. Walter Vogler and Mark Schäfer deserve a lot of credit for providing guidance in the sometimes 'cold world' of formalisms. I am grateful to my supervisor Werner Zorn for making this thesis possible.

I would like to thank Victor Khomenko and Alex Yakovlev who gave me the chance to contribute to their research at the Newcastle University. Many thanks go to my former colleagues in Newcastle – especially Arseniy Alekseyev, Andrey Mokhov and Ivan Poliakov – as well as to Josep Carmona from the UPC in Barcelona for fruitful discussions and collaboration. Further thanks go to the DFG and the SOA research school of the Hasso Plattner Institute for their fundings.

Besides this professional support there were many people who gave me emotional support during the last years. Many thanks go to Stephan, Rami, Norman, Robert, Jakob, Johannes, Marcus, Fabian as well as to all my students who made my life at the Hasso Plattner Institute much more entertaining. I would like to thank all my friends in Newcastle, especially my flatmates Harriet, Pablo, David, Omar and Erik for having a very pleasant stay there. I am particularly grateful to Bütti, Claudi and Philipp for their support. Last but not least, I would like to thank Jakob and Anja for proofreading.

Contents

1 Introduction

The operation of modern microelectronic circuits is usually driven by one or several distributed periodic timing signals, so-called *clocks*. Despite the fact that it might be more natural to build such systems (consisting of many collaborating components) asynchronously, the introduction of a clock signal significantly simplifies their design[1] and test[2]. Furthermore, the insufficient maturity level of available EDA (Electronic Design Automation) tools prevents a breakthrough of the asynchronous technology. However, with higher integration levels and clock frequencies the impact of the following problems of synchronous systems increases, cf. also [Now93, vBJN99, Spa01]:

- *Clock Skew Problem.* The global clock signal must be distributed with minimal skew across the circuit, which gets more problematic with higher clock frequencies.

- *Higher Power Consumption.* The perpetual clock signal requires every component to consume power even if the component is not active in the current computation.

- *Worst-Case Performance.* The operating speed of synchronous circuits is determined by global worst-case latencies rather than actual local latencies.

[1] The introduction of a clock signal is a crucial step for avoiding malfunction of a circuit caused by so-called hazards, cf. [Ung69].

[2] Synchronous circuits have two features that simplify testing dramatically: they can be stopped during each clock cycle, and it is both simple and cheap to include a scan-chain through all flip-flops [vBJN99, HBB95].

- *Electro-Magnetic Emission.* If one component emits electro-magnetic radiation at the frequency of its clock (or higher harmonic frequencies) a radio receiver component might mistake this radiation for a valid radio signal.

- *Modularity and Composability.* If a component of a synchronous system should be replaced by another one (working at a different speed) or several components are combined to realise a new system, the clock speed for the (entire) system must be adjusted to the slowest component; otherwise the system may malfunction.

Large companies such as ARM, Boeing, Epson, Intel, Infineon, IBM, Sun and Philips as well as smaller start-ups like Achronix, Elastix, Fulcrum[3], Tiempo and GreenArrays, Inc. already exploit the advantages of asynchronous technology today. First entirely asynchronous microprocessors have already hit the market, e.g. [vGvBP+98] and [KNI08]. Even fully asynchronous multi-computers (i.e. an array of independent complete computers – each with its own memory – on a single chip) have recently been developed, cf. the SEAforth 40C18 array processor [Int] and the GA144 [Gre]. However, large portions of these systems were designed without tool support.

A driving force for a widespread use of asynchronous technology is the availability of mature EDA tools. During the last two decades there was much progress in developing such tools, but the number and the maturity level of them is still incommensurable with EDA tools for synchronous system design. In particular, the application of logic synthesis[4] for the system's control path is important in order to build efficient circuits (e.g. in terms of area and performance), but logic synthesis suffers from complexity problems.

1.1 Scope

The aim of this thesis is to enable logic synthesis of asynchronous circuits from *complex* specifications. A decompositional approach will be used for the synthesis of speed independent (SI) circuits [MB59], a robust subclass of asynchronous circuits. Figure 1.1 compares systems for pure SI logic synthesis and *decomposition-based* SI logic synthesis.

For specifying the behaviour of SI circuits *Signal Transition Graphs (STGs)* [Chu87, Wen77] are widely used. STGs are labelled Petri nets which model the occurrence of rising and falling edges of circuit signals by transition firing. PETRIFY [CKK+97,CKK+02]

[3] Recently, Intel agreed to acquire Fulcrum Microsystems Inc. which could signal a significant change of direction for the world's biggest chip maker toward asynchronous technology.

[4] During logic synthesis a desired circuit behaviour specification is turned into an efficient design implementation in terms of logic gates by exploiting the power of Boolean minimisation techniques.

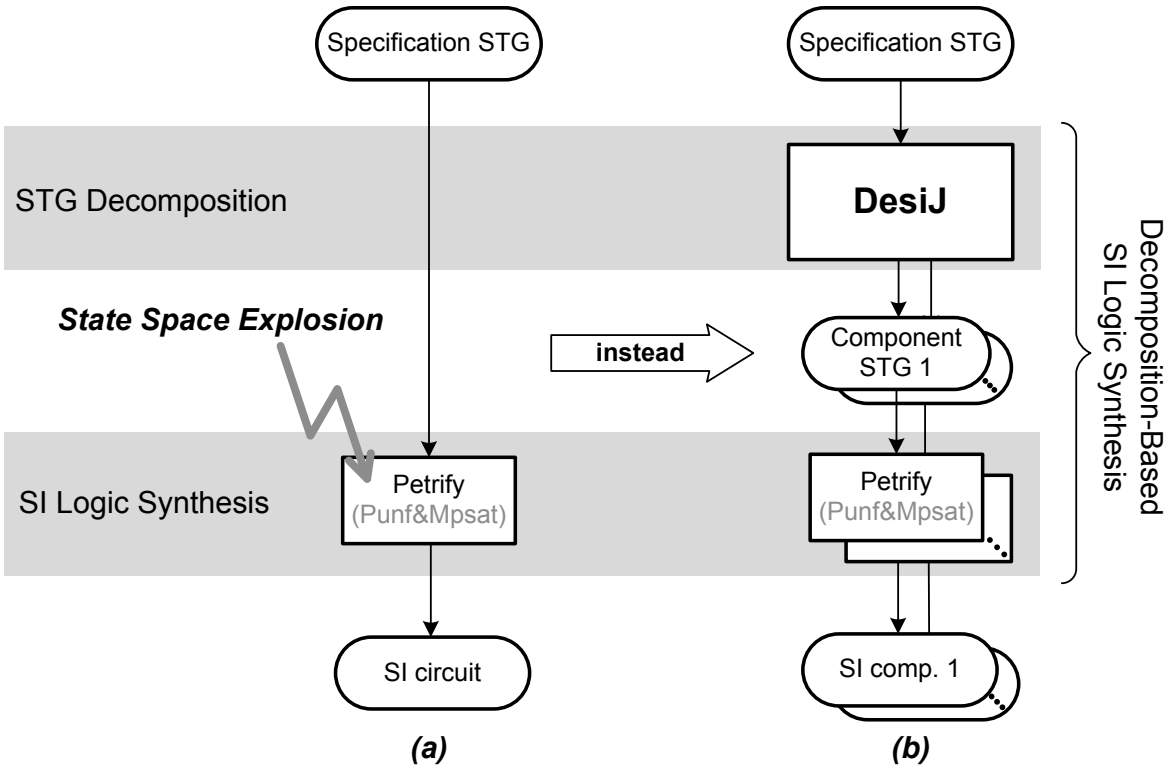


Figure 1.1: SI logic synthesis systems: pure SI logic synthesis (a) vs. decomposition-based SI logic synthesis (b)

These FMC architecture diagrams consist of active system components (called *agents*) as well as passive ones (*storages*) [KGT06]; agents and storage elements are depicted as rectangles and ovals resp.

as well as PUNF&MPSAT [KKY04] are commonly used tools for logic synthesis of SI circuits; both tools use STG *specifications* as design entry. For logic synthesis, PETRIFY must explore the entire state space of the STG, and hence suffers from the *state space explosion* problem [Val98], cf. Figure 1.1(a). The unfolding based synthesis via MPSAT has also complexity problems when solving huge SAT problems[5] (cf. [KMY06]).

To cope with the complexity issues of logic synthesis, *decomposition-based* logic synthesis was proposed in [Chu87, KKT93, Wol97]. By using fast structural techniques, a large STG specification (having a complex state space) can be decomposed according to a partition of the output signals into several smaller component STGs (having a smaller state space each), one for each member of the partition. After decomposition, logic synthesis will be applied to each smaller component, cf. Figure 1.1(b). Thus, the essential motivation for the application of decomposition is that pure logic synthesis is not applied to a large specification, but to smaller components (with less complexity). Finally, the interacting component circuits in combination form the desired asynchronous system.

More recent STG decomposition approaches are presented in [VW02, VK07, CCCGV06, YM07]. The approaches from [CCCGV06, YM07] need specifications satisfying the so-called CSC (Complete State Coding) property which is necessary for SI logic synthesis. Then, after decomposition CSC is preserved for the components. However, specifications do not satisfy CSC in general and optimal CSC solving is a complex computational task. A CSC solving technique for preprocessing *large* specifications was proposed in [CC06, CC08], which in combination with the method of [CC03, CCCGV06] can synthesise efficient circuit implementations. Both [CCCGV06, CC08] are based on solving $\mathcal{NP}$-complete ILP[6] problems, which entails restrictions on the specification size. Furthermore, both decomposition techniques [CCCGV06] and [YM07] only yield total decompositions (i.e. one output per component) which may restrict the optimisation space for the final implementations.

In this thesis, STG decomposition according to [VW02, VK07] is used which formalises the approach from [Wol97], in particular a correctness notion for STG decomposition is defined and the decomposition algorithm is proven correct. In [Sch08] generalisations and advanced decomposition strategies have been discussed. The applied STG decomposition method allows the application of coarser output partitions which yields improved circuit implementations. Additionally, specifications without CSC can be decomposed; this enables the handling of more complex STGs than with the approaches from [CCCGV06, YM07].

[5] Boolean **sat**isfiability problem [HMU06]
[6] Integer linear programming [STC98]

1.2 Contribution

The applied decomposition technique is more scalable than other approaches and aims at very large specifications, but in many cases some of the resulting component STGs might have *irreducible* CSC conflicts, i.e. conflicts that cannot be solved for the component in isolation, even if the initial specification has none. In [KS07], it was already attempted to control the decomposition algorithm in such a way that the component STGs satisfy CSC. However, this does not always succeed, and it can also result in too large component STGs.

Therefore, a new solution to this problem is proposed in this thesis. After decomposition, irreducible CSC conflicts will be avoided by introducing internal communication between the components, in such a way that the component size remains small and the overall behaviour of the resulting asynchronous system is preserved.

In particular, it will be shown how irreducible CSC conflicts in component STGs can arise and how the new internal communication signals can be introduced by using purely structural techniques. Such new structural techniques and an optimised *correct* algorithmic solution will be presented for both the handling of simple structured specifications and for a special (but often occurring) conflict type, so-called self-trigger. Furthermore, improvements for the handling of more general irreducible CSC conflicts will be provided as well as advanced structural techniques – and their correctness proofs – in order to handle specifications having a more general structure.

Thus, from another point of view this thesis contributes to coping with complexity problems in the important process of CSC solution for very complex STGs, by splitting this process into two steps:

1. During decomposition, internal communication signals are inserted to avoid irreducible CSC conflicts for the final components; for this, fast structural methods are only applied. (However, the resulting implementations could be less efficient in comparison to implementations resulting from a pure logic synthesis for the entire specification.)

2. Then Petrify inserts further internal signals to solve the remaining *reducible* CSC conflicts for each component; in fact, this signal insertion is a more complex computation than the aforementioned structural method, but it leads to more efficient final circuit implementations.

In order to counteract the potential loss in efficiency of the hardware resulting from the *structural* insertion of internal communication signals, the flexibility in building

the output partition can be exploited. This work starts to explore this optimisation space. In particular, for coarser partition members one can expect that CSC solving for one larger component leads to an improved final circuit implementation (as long as the resulting components can still be synthesised with PETRIFY), in comparison to the combination of the corresponding several smaller component implementations resulting from total decomposition. Furthermore, less irreducible CSC conflicts can be expected when generating larger components.

Four heuristics will be proposed to build improved output partitions to get component STGs which are SI-implementable and from which improved circuit implementations can be synthesised, in comparison to the application of total decomposition. In particular, the heuristics exploit the re-using of signals for more than one component, it will be shown how to get improved components that ('almost') satisfy CSC and how state explosion for coarser components can be prevented by restricting their increase of concurrency. To determine which output signals should form one partition member, merely structural techniques are used, either reasoning directly on the specification's structure or approximating its reachable markings by using efficient LP solving techniques.[7]

1.3 Organisation

This thesis is organised as follows: Chapter 2 presents related work. It gives an overview of the state-of-the-art design methods for asynchronous systems. In particular, logic synthesis of asynchronous circuits as well as decomposition-based logic synthesis will be introduced.

Chapter 3 defines the basics of (labelled) Petri nets, and STGs, introduces a number of operations which are important for STG decomposition and defines the decomposition operation as well.

The succeeding chapters present the main contribution of this thesis. They are based on articles in which the author of this thesis significantly participated.

Chapter 4 is crucial to adapt the STG decomposition method of [VW02, VK07] to SI logic synthesis. It will be demonstrated how irreducible CSC conflicts in component STGs can arise and how they can be avoided by introducing internal communication between the components. New structural techniques, optimisations and corresponding correctness proofs are given. Furthermore, promising experimental results are presented.

[7] Note that ILP is often preferred – and it can give better approximations as stressed in [EM00]; in contrast, one can achieve good results just with LP, and partially much faster than with ILP.

Chapter 5 introduces and discusses four partitioning heuristics to improve the resulting circuits and again successfully validates their applicability on benchmarks.

At the end of both chapters 4 and 5 the limitations and opportunities of the presented approaches are discussed. In particular, concrete ideas for further improvements will be sketched.

Chapter 6 presents the decomposition tool DesiJ that was initially developed by Mark Schäfer [Sch07]. It implements the decomposition technique from [VW02, VK07] as well as the improvements from [Sch08]. The author of this thesis extended DesiJ with the algorithms which will be discussed in chapters 4 and 5. All benchmark computations of sections 4.5 and 5.5 are performed using DesiJ.

In Chapter 7 conclusions are drawn and an outlook for future research is given.

2 Asynchronous System Design

This chapter briefly introduces asynchronous circuits and presents an overview of different state-of-the-art synthesis methods for asynchronous systems. In particular, the idea of decomposition-based logic synthesis, which is the core of this thesis, will be introduced in sections 2.2.1 and 2.2.2.

2.1 Asynchronous Circuits

The operation of asynchronous circuits is not driven by clock events, but by events of the circuit signals which are usually independently generated; thus, asynchronous circuits are highly concurrent systems. Events are considered as edges of binary signals which are observable on the wires of the circuit. Rising edges (from logical 0 to 1) can be distinguished from falling ones (from logical 1 to 0).

Figure 2.1 depicts an asynchronous implementation of a simplified VME bus controller at the gate level. Three types of events can be distinguished: Input events (*ldtack*, *dsr*) are produced by the environment *for* the circuit, output events (*lds*, *dtack*, *d*) are *produced by* the circuit for the environment and internal events (*csc*) are produced by the circuit (like outputs), but they are fed back to the circuit as inputs for the combinational logic δ. Note that outputs can also be fed back (e.g. signal *d*).

In general, the combinational logic δ cannot be free of hazards for *all* combinatorially possible input signal changes. A *hazard* is the possibility of multiple output or internal signal changes, also called *glitches*, caused by an unique input signal change, cf. [Ung69].

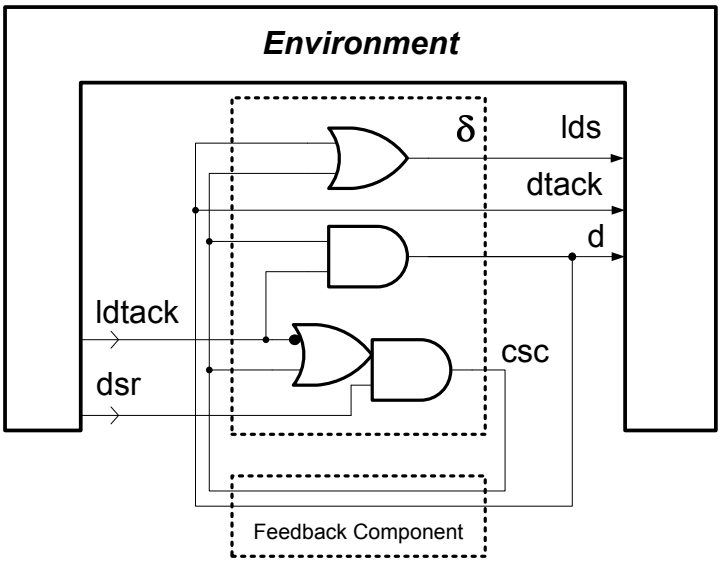


Figure 2.1: Aynchronous VME bus controller considered at the gate level.

Consequently, glitches on the feedback wires might occur and can be misinterpreted as valid input signal changes by δ, which might yield incorrect computations.

To avoid such malfunction, asynchronous circuits have to be free of hazards, in particular free of combinational hazards in δ for all *permitted* input signal changes[1] and free of hazards that inherently involve the feedback[2].

Asynchronous *systems* consist of several (smaller) interacting asynchronous circuits which communicate via some signaling protocol. Usually, so-called *handshake protocols* are used involving *requests*, which are used to initiate an action, and corresponding *acknowledgments* used to signal the completion of that action. For example, let there be two circuits: a sender S and a receiver R. A request is sent from S to R via a signal edge on a request line to indicate that S is requesting some action by R. When S is done with this action or has stored this request, it acknowledges it by sending a signal edge from R to S on an acknowledgment line. Section 2.3.2 sketches an approach how to build asynchronous systems by connecting so-called handshake components which communicate via handshake protocols. Further details about signaling protocols can be found in [DN97, Spa01, Bar00].

Further issues in asynchronous circuit or system design, in particular completion

[1] The mode of operation of an asynchronous circuit defines the *permitted* input signal changes, see Section 2.1.1.

[2] i.e. so-called essential hazards and critical races between feedback signal changes, see [Ung69]

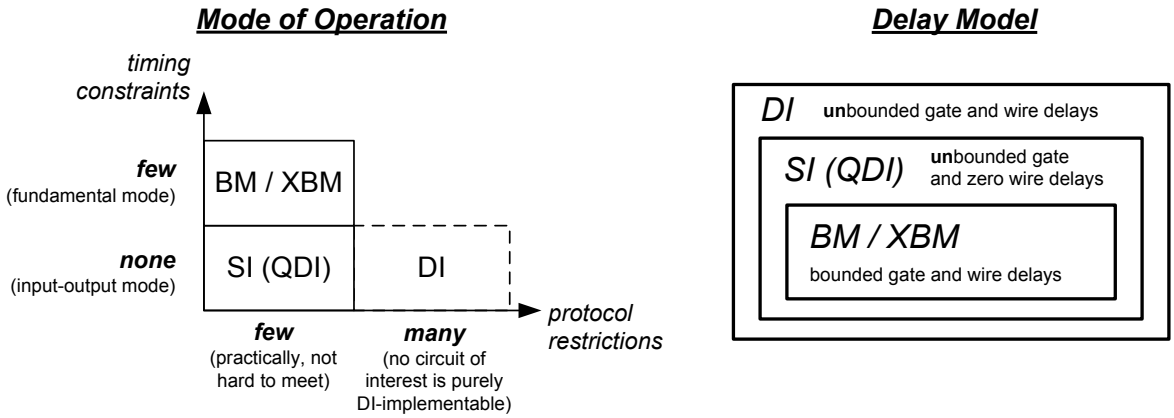


Figure 2.2: Classification of asynchronous circuits according to their modes of operation and delay models.

detection, asynchronous data path implementation styles[3], pipeline architectures and arbitration are not focused in this thesis and therefore omitted, but details can be found e.g. in [DN97, YK98, Spa01].

2.1.1 Classification of Asynchronous Circuits

As mentioned, asynchronous circuits have to be free of hazards in order to avoid wrong computations. Since there is a wide spectrum of hazard free implementable asynchronous designs a classification will now be given. One way to distinguish asynchronous designs is to understand the underlying models of delay and operation.

Every physical circuit has inherent delay. The *delay model* of a circuit characterises the delays corresponding to its gates and wires.

The *mode of operation* characterises the interaction of the circuit with its environment. If the environment is allowed to respond to some circuit's outputs without any timing constraints, then both interact in *input-output mode*. Otherwise, timing constraints are assumed, e.g. the *fundamental mode* requirement: the environment must not produce input changes until the circuit's state has stabilised [Now93]. Furthermore, arbitrary protocols – i.e. arbitrary input/output changes – cannot be implemented hazard free. Thus, the mode of operation defines restrictions to the protocol between the circuit and its environment as well.

Figure 2.2 classifies hazard free implementable asynchronous circuits w.r.t. their modes of operation (left) and delay models (right).

[3] e.g. bundled data or dual rail

Delay insensitive (DI) circuits work correctly under the unbounded gate and wire delay model, i.e. they are *insensitive* for both gate and wire delays. There are no timing constraints for the environment of DI circuits to meet w.r.t input signal changes, i.e. they are operating in input-output mode. However, the subclass of hazard free implementable DI circuits is very limited, practically all circuits of interest fall outside this subclass [Mar90]. (Hence, this class is depicted as a dashed rectangle in Figure 2.2(left).) However, simple handshake protocols (e.g. 4-phase handshaking) are DI implementable which makes the DI technology useful for interfacing several smaller circuits (which are implemented in different ways, e.g. as speed independent circuits) in order to build a larger system [SKC+99].

A practical useful restriction to delay insensitivity is the *isochronic fork* assumption[4], i.e. if there is a wire fork at some gate output y, then all gates receiving y as an input must register an edge of y ideally at the same time[5]. The isochronic fork assumption is sufficient to construct any circuit of interest [Mar90].

Speed independent (SI) circuits [MB59] are robust to any gate delay variation, but wires are assumed to have negligible (i.e. zero) delay; i.e. the so-called *unbounded* gate delay model is assumed. In fact, this is equivalent to the isochronic fork assumption: Assume there is a gate having a delay α and a non-forking wire on its output having a delay β, one can also consider the gate as having the delay $\alpha + \beta$ and the wire has no delay instead. If the delays on forking wires are isochronic, then they can be added to the preceding gate in the same way. SI circuits work in input-output mode, i.e. their environments can generate new input changes right after all causative output changes have been occurred (i.e. the environment does not need to *wait* a certain amount of time). The interface behaviour of SI circuits is usually specified by a so-called signal transition graph (STG), which is a labelled Petri net specifying the causal relations between input and output signal edges of the desired circuit, see Section 2.2 for details. However, not from every STG a hazard free circuit can be implemented. If an STG fulfills the following properties: boundedness, consistency, complete state coding (CSC) and output persistency, see Section 3.2.1 for details, then the specified behaviour is hazard free implementable, i.e. an SI circuit can be synthesised.

In contrast, *burst-mode (BM)* and *extended burst-mode (XBM)* circuits basically work in *fundamental mode* of operation, i.e. their environments must not produce an input change until the circuit (to be more precise: the circuit's state) has stabilized [Now93]. To realize hazard freedom only specific input and output signal changes are permitted.

[4] The corresponding circuits are called *quasi* delay insensitive (*QDI*).
[5] In fact, the difference in arrival times must be smaller than the smallest gate delay.

The permitted behaviour can be specified by so-called burst-mode machines (BMM) or extended burst-mode machines (XBMM). Gate and wire delays for BM or XBM circuits are assumed to be *bounded*, i.e. the so-called *bounded gate and wire delay model* is assumed.

Now, different synthesis methods will be presented, which exploit the advantages of asynchronous technology in different ways. First of all, STGs and logic synthesis as well as decomposition-based logic synthesis will be considered (Section 2.2), since it is essential in order to understand some of the following synthesis methods, presented in Section 2.3.

2.2 Logic Synthesis of Asynchronous Controllers

The term *logic synthesis* refers to design techniques where a behavioural specification of a circuit, such as an STG, is turned into an efficient design implementation in terms of logic gates, by exploiting the power of Boolean minimisation.

Signal Transition Graphs (STGs) [Chu87, Wen77, RY85] are widely used formalisms for modelling the concurrent behaviour of asynchronous control circuits, in particular of SI circuits. STGs are labelled Petri nets [Pet62, Mur89, Rei10], where the transitions are labelled with rising and falling edges of the circuit's input, output or internal signals. An STG specifying the interface behaviour of the VME bus controller (presented Figure 2.1) from the causal point of view is shown in Figure 2.4(a). Input signal transitions are drawn with a thick border, the other transitions model output signal edges. For a formal definition of STGs the reader is referred to Chapter 3.

To synthesise a circuit from an STG *specification* one can use logic synthesis or direct mapping techniques. *Direct mapping based synthesis* from an STG exploits the STG *structure* to build the final circuit, i.e. particular STG fragments are directly mapped to hardware. This synthesis approach is not considered here, but details can be found in [SBY03, BSKY04]. Note, with direct mapping based synthesis from an STG one can synthesise circuits from very large specifications, because it does not suffer from complexity problems. However, the resulting hardware is usually not very efficient, i.e. the performance, chip area and energy consumption cannot compete with synchronous pendants, cf. e.g. [SY05].

This work focuses on STG-based logic synthesis, or to be more precise on SI logic synthesis. Tools for SI logic synthesis like Petrify [CKK+97, CKK+02] and Mpsat[6]

[6] in combination with Punf

[KKY04, KMY06] can produce much better results by exploiting the power of Boolean minimisation techniques [SY05]. SI logic synthesis requires the construction of the state graph (or reachability graph) of the STG specification [CKK+02] which might lead to state explosion [Val98]. In fact, PUNF&MPSAT uses so-called Petri net unfoldings [McM93, ERV96] to represent the state space more compact, but MPSAT suffers from complexity problems when solving huge SAT problems[7] in order to extract or compute the information needed for circuit synthesis. Thus, the established tools like PETRIFY and PUNF&MPSAT need unacceptable long computation times or suffer from memory overflows when doing logic synthesis for 'real-life' specifications of medium or large sizes. As a guidance, PETRIFY can handle specifications up to 30 and PUNF&MPSAT up to 50 signals. The complexity problems are considered as the main bottleneck for an efficient controller synthesis, cf. e.g. [BL00, SY05].

A different logic synthesis approach for asynchronous controllers uses so-called *(extended) burst-mode machines ((X)BMM)* as design entry, instead of STGs. BMMs [Now93] are Mealy-type [Mea55] specifications, where state transitions or arcs are labelled with so-called *bursts*, i.e. sets of input or output signal edges. Because of their sequential nature BMMs cannot model the inherent concurrent behaviour of asynchronous controllers, as it can be done with STGs.[8] Thus, the designer is not able to appropriately specify concurrency from a causal point of view, as with STGs. Instead, concurrency is specified on the state graph level (i.e. from a temporal point of view) which makes the state space explosion explicit when trying to model high concurrency. A combined approach that uses STGs as design entry (in order to enable appropriate modelling of concurrency) and XBMMs for logic synthesis was proposed in [BEW99][9].

Burst-mode logic synthesis suffers also from complexity issues. The logic synthesisers MINIMALIST [FNT+99] (BMM) and 3D (XBMM) suffer from complexity problems in state assignment and Boolean minimisation for (X)BMMs having more than 30 signals [AN07].

2.2.1 Decomposition-based Logic Synthesis

In order to deal with the complexity issues in logic synthesis a large controller specification can be decomposed into several smaller component specifications which will then

[7] In complexity theory the satisfiability problem (SAT) for a Boolean expression in propositional logic is an $\mathcal{NP}$-complete decision problem which decides whether there is some assignment to the variables that will make the entire expression TRUE [HMU06].

[8] With XBMMs [YD99a, YD99b] this issue was slightly improved.

[9] Here, STG decomposition is a necessary step to handle the state space explosion before applying XBM logic synthesis (cf. Section 2.2.1).

be synthesised in isolation using some (established) logic synthesiser.

To cope with complexity problems in burst-mode logic synthesis, a decomposition method for BMMs was proposed in [AN07]. Although the resulting components in isolation can be implemented very efficiently in terms of area and performance, the combination of them needs some hardware overhead in order to meet the required timing constraints (cf. fundamental mode, Section 2.1.1), which reduces the efficiency though.

In this thesis, STG decomposition is focused in order to tackle state explosion; i.e. an STG specification having a huge state space will be decomposed into several smaller components having smaller state spaces each. One can distinguish coverability-based decomposition approaches from contraction-based ones. *Coverability-based STG decomposition* requires a complete disjunctive cover of the specification by particular STG components; e.g. [Poh80, Gei85] need a specification to be completely coverable by *transition-disjunctive* state machine components [Sta90] having a token count of 1. Each such state machine component must represent a *synthesisable* state machine, i.e. it must consist of all transitions labelled with relevant signal edges to produce some output signal. This significantly restricts the class of valid specifications.

Here, *contraction-based STG decomposition* [Chu87, KKT93, Wol97] is focused. According to [Chu87], for each output signal the necessary signals to produce it have to be determined – this partitions the signals in *relevant* and *irrelevant* signals. Then from a copy of the specification a component STG can be extracted by contracting all *irrelevant* signals for the considered output. From each component STG a component circuit can be synthesised using established logic synthesisers. The combination of all component circuits implements the behaviour modelled by the specification.

Observe that STG decomposition attacks state explosion, since:

- the reachability graphs (or state graphs resp.) of the component STGs are usually much smaller than the reachability graph of the specification, and
- STG decomposition is a structural operation, i.e. it works purely on the graph structure of the specification and does not require to generate its reachability graph or state graph.

STG decomposition according [Wol97] is more general than the approaches from [Chu87, KKT93], in particular it is not restricted to specific net classes and a component circuit can produce more than one output signal. [VW02, VK07] defines this decomposition technique formally and presents a correctness proof.

Most recent work on STG decomposition as well as tool implementations were done by

- Vogler et al.: the decomposition tool DesiJ implements the ideas of [VW02, KWVB03, VK07, Sch08],
- Yoneda et al.: the ideas of [YOM04, YM07] were implemented in Nutas, and
- Carmona et al.: the ideas of [CC03, CCCGV06] were implemented in Moebius.

Here, the version of Vogler and Wollowski (as an improvement of [Wol97]) is focused.

These recent approaches primarily aim at synthesis of speed-independent circuits[10] which requires at some point the construction of a state graph that fulfills the so-called CSC (complete state coding) property (see Section 3.2.1 for a formal definition). STG decomposition according to [CCCGV06] and [YM07] need specifications which fulfill CSC initially. This requires a CSC solving technique that efficiently works for large specifications. In [CC06, CC08] such a CSC solving technique based on the solution of large ILP problems was proposed, but the $\mathcal{NP}$-completeness of ILP problem solving restricts the specification size again. However, the advantage of the approaches from Carmona et al. and Yoneda et al. is: *If* the specification satisfies CSC, *then* each generated component STG satisfies CSC, too.

The decomposition according to Vogler et al. does not require the CSC satisfaction for the specification, but the component STGs might not satisfy CSC which prevents an SI circuit synthesis; this requires additional effort to avoid at least so-called *irreducible* CSC conflicts for the final components. A solution to this problem is proposed in Chapter 4. Note that the STG decomposition technique used here is the only approach which allows final components producing more than one output signal. By exploiting this opportunity, the resulting circuits can be improved e.g. in terms of area consumption, cf. Chapter 5.

Recall, the coarse grained architecture of the decomposition-based SI logic synthesis system used in this thesis is depicted Figure 1.1(b). Usually, Petrify is applied for SI logic synthesis of each component, but Punf&Mpsat can also be used. Details of DesiJ's architecture will be given in Chapter 6.

2.2.2 Introductory Example

In this thesis SI logic synthesis (as with Petrify) from large STG specifications by using STG decomposition (according to Vogler and Wollowski) is focused. To clarify

[10] Basically, this is not a restriction for the decomposition according to [Wol97], cf. [BEW99].

the idea of decomposition-based SI logic synthesis, cf. Figure 1.1(b), it will be sketched for the VME bus controller, see Figure 2.1, where state explosion is actually not an issue.

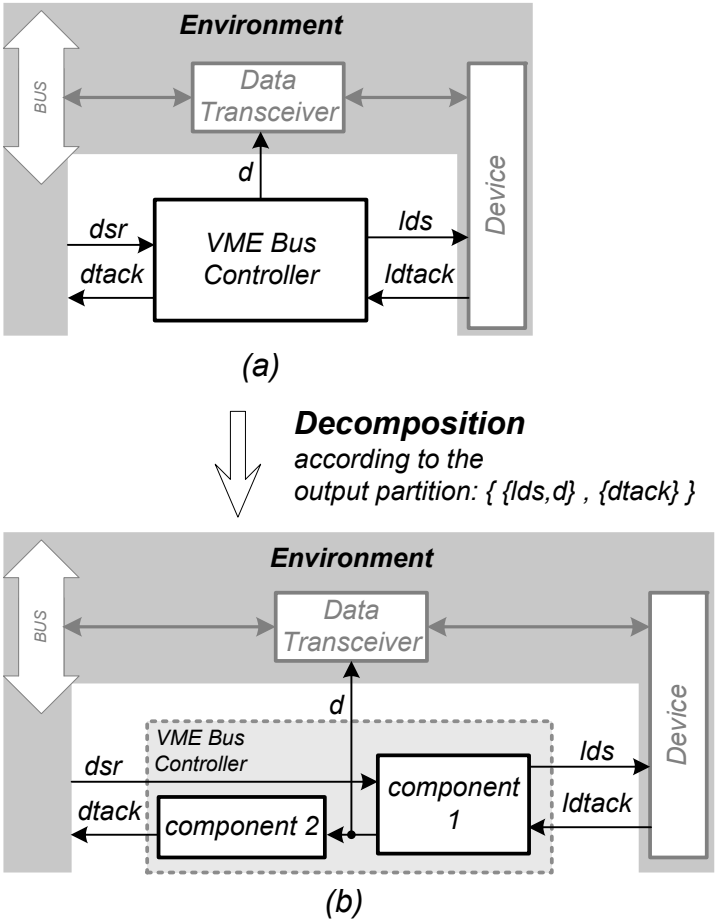


Figure 2.3: VME bus controller system (a) and the decomposed controller architecture (b)

The STG in Figure 2.4(a) specifies the interface behaviour of the desired VME bus controller in Figure 2.3(a), as causal relations between input and output signal edges.

Instead of synthesising the VME bus controller en bloc from this specification it will be decomposed into two smaller component circuits, see Figure 2.3(b): One generating output signal *lds* and *d* (component 1) and the second generates *dtack* (component 2). This output partition is arbitrary, i.e. different output partitions $\pi(Out)$ can be chosen leading to different circuit architectures.

According to the chosen output partition, the component STGs can be extracted from the specification by *reducing* it via transition contraction. For this, one takes a copy of the specification for each partition member. Then for each such copy the signal set must be partitioned into relevant and irrelevant signals w.r.t. the related partition

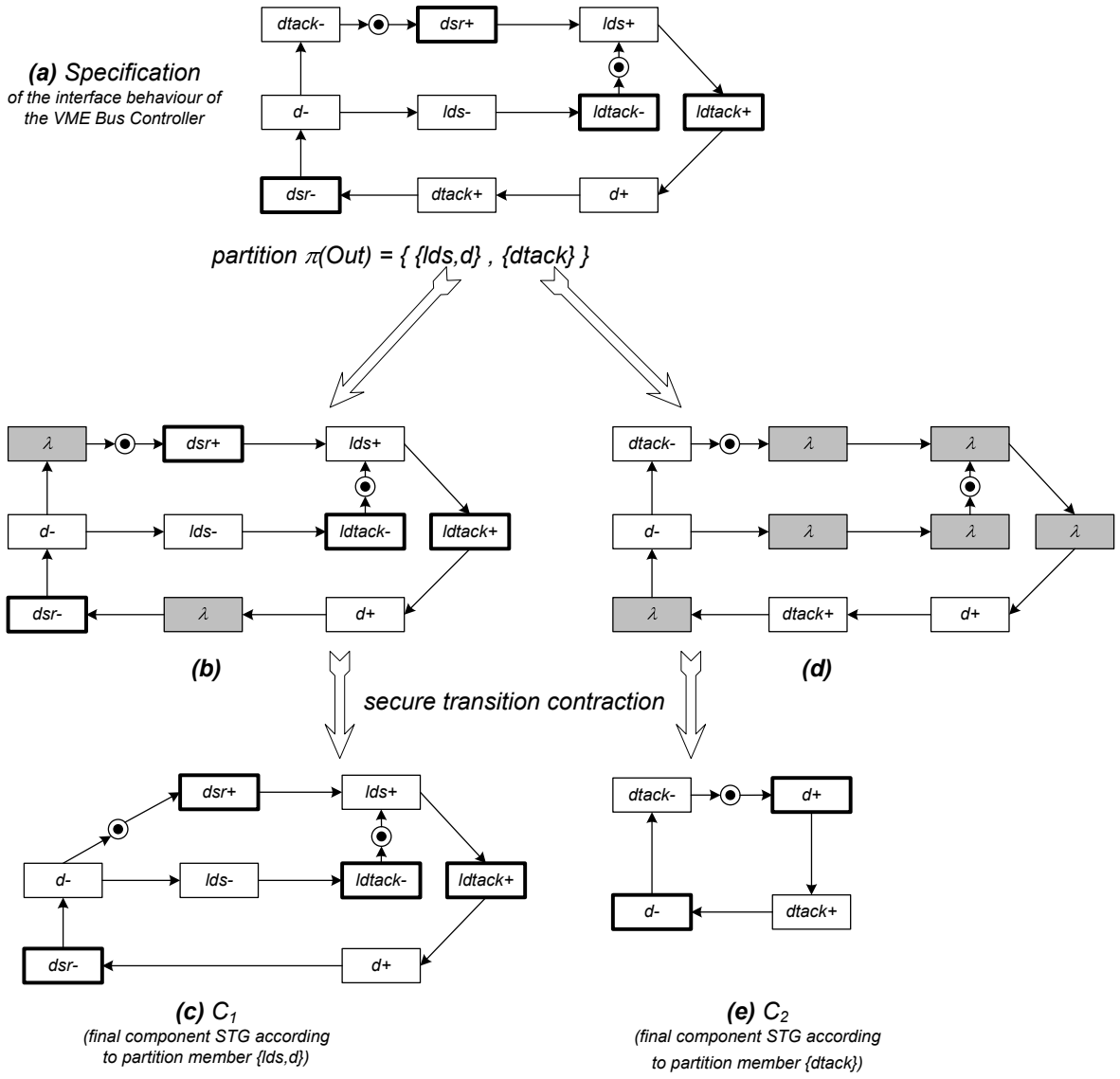


Figure 2.4: STG decomposition for the VME bus controller

member. At first glance, all signals which *directly* trigger a considered output signal are *relevant* to produce this output; all other signals are considered to be *irrelevant*. Referring to the generation of the component STG C_1 which is related to the partition member $\{lds, d\}$ (see Figure 2.4(b) and (c)) one can see, by considering the specified causal relations in Figure 2.4(a), that all transitions labelled with output edges of *lds* and *d* are directly triggered by transitions labelled with edges of *ldtack*, *dsr* and *d* itself. Transitions labelled with different signals are irrelevant (for this simple example), i.e. the *dtack*-labelled transitions; they are re-labelled with λ which is called *lambdarisation*, see Figure 2.4(b). Then every λ-transition is contracted from the STG in Figure 2.4(b) by applying so-called *secure transition contraction*[11] (see Definition 3.2.1 below) yielding the final component STG C_1, cf. Figure 2.4(c).

This reduction process works similar for component 2 in Figure 2.3(b). Only signal *d* is relevant to produce the output *dtack*, cf. Figure 2.4(d). However, *d*'s signature must be changed from output to input, since an output signal can only be produced by one component which is done by component 1 here. This introduces internal communication via signal *d* from component 1 to component 2, as depicted in Figure 2.3(b). Again, all λ-transitions have to be contracted yielding C_2, as shown in Figure 2.4(e).

In fact, for this 'small' VME bus controller specification in Figure 2.4(a) state explosion is not an issue, since its state graph just consists of 14 states. However, after decomposition C_1's state graph has 10 and C_2's 4 states. Indeed, the state sum of both component STGs is 14 again, but for the handling of complexity it can already be beneficial if each component state graph is smaller than the one of the specification; this might reduce peak memory usage. For larger specifications (which have much larger state graphs) this state reduction per component is usually more significant than for the VME bus controller example. Even the cumulated states of all component state graphs very often give a number much smaller than the state count of the specification; examples are shown in Table 2.1.

Benchmark	Specification *state count*	Components *cumulated state count*
FIFO	832	$26 + 12 + 12 + 8 + 4 + 4 = 66$
locked2	166	$34 + 20 + 20 = 74$
mread-8932	8932	$36 + 36 + 18 + 18 + 12 + 10 = 130$

Table 2.1: STG decomposition avoids state explosion. The results are taken from [VW02].

[11] This basically means to remove each λ-transition in such a way that the causal relations are preserved.

In these cases the application of decomposition-based SI logic synthesis (using DESIJ and PETRIFY) can either reduce the time for logic synthesis by orders of magnitude (in comparison to the application of pure logic synthesis to the specification), as for the benchmarks 3pp_wk.09 and SMPS4 in Table 2.2; or STG decomposition can even *enable* logic synthesis for very complex specifications from which no circuit can be synthesised when applying PETRIFY to the specification, see the benchmarks 3pp_wk.12 and SeqTree5 in Table 2.2.

Benchmark	PETRIFY	DESIJ + PETRIFY
3pp_wk.09	1467 sec	3 sec
3pp_wk.12	*impossible*	3 sec
SMPS4	216 sec	2 sec
SeqTree5	*impossible*	1 sec

Table 2.2: STG decomposition enables (fast) logic synthesis.

SI Logic Synthesis

As shown above in Figure 1.1(b), after STG decomposition (using DESIJ) logic synthesis (usually using PETRIFY) will be applied for each component STG. The idea of PETRIFY's logic synthesis [CKK+02] will now be sketched w.r.t. the component STG C_1, in Figure 2.4(c). First, C_1's state graph has to be constructed, see Figure 2.5(a). It is C_1's reachability graph, where each marking is annotated with a *state vector* representing the current level for each signal.[12]

During logic synthesis for each internal and for each output signal a Boolean *next-state function* is derived, as a mapping from $\{0,1\}^{|sv|} \longrightarrow \{0,1\}$ (where $|sv|$ is the length of the state vector). It is defined as follows: For each reachable state s the next-state value for some output z is either its signal level in s if z is not excited – i.e. no edge of z is willing to occur – or it is the complementary value of its signal level in s. In more detail, if $z = 0$ for some state s and z is excited in s (i.e. z^+ will occur), then z's next-state value is 1; if $z = 1$ in s and it is excited (i.e. z^- will occur), then z's next-state value is 0. For the state graph in Figure 2.5(a) the next-state function is defined as in Table 2.3 (left).

Note that there is an ambiguity for the next-state value of $(dsr, ldtack, lds, d) = 1110$. There are two related states (shaded grey in Figure 2.5(a)) for which two different

[12] *Current* signal level means the level after the occurrence of all signal edges which are represented by the transitions fired *before* reaching the considered marking or state (cf. Section 3.2.1 for a formal definition).

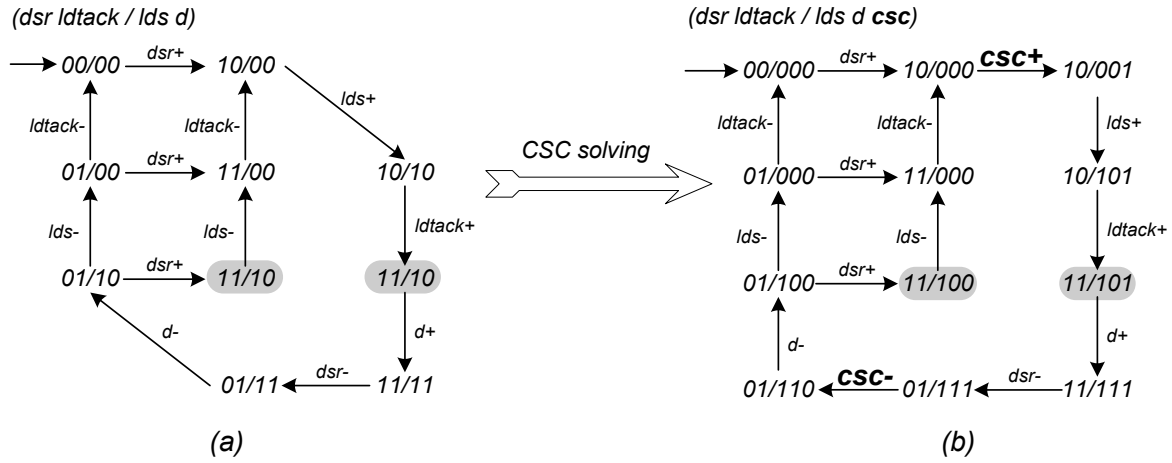


Figure 2.5: State graphs corresponding to component STG C_1 from Figure 2.4(c).

$(dsr, ldtack, lds, d)$	lds'	d'
0000	0	0
1000	1	0
0100	0	0
1100	0	0
1010	1	0
0110	0	0
1110	**0/1**	**0/1**
0111	1	0
1111	1	1
else	*	*

$(dsr, ldtack, lds, d, csc)$	lds'	d'	csc'
00000	0	0	0
10000	0	0	1
10001	1	0	1
01000	0	0	0
11000	0	0	0
10101	1	0	1
01100	0	0	0
11100	**0**	**0**	**0**
11101	**1**	**1**	**1**
01110	1	0	0
01111	1	1	0
11111	1	1	1
else	*	*	*

Table 2.3: Truth tables for the next-state functions of the internal and output signals of the VME bus controller's component 1, cf. Figures 2.3(b). Both tables are related to the state graphs in Figure 2.5, the left table corresponds to (a) and the right one to (b).

outputs are excited, either d^+ or lds^- are willing to occur. A circuit implementation 'arriving' this state cannot decide which output to produce. This phenomenon is-called a *CSC conflict* (Complete State Coding conflict, see Section 3.2.1 for a formal definition), i.e. there is a lack of information to decide which output must be produced.

To solve this problem a new internal signal *csc* can be introduced which separates the shaded states by its value, see Figure 2.5(b). However, this signal must not change the interface behaviour from the perspective of the circuit's environment, i.e. it must not delay an input edge, but merely other internal or output signal edges. This is called an *input proper* event insertion [SV07]. In Figure 2.5(b) the insertion of *csc* is input proper, because it merely delays the production of the output edges lds^+ and d^-.

From a state graph satisfying CSC a truth table for each internal and output signal can be derived as explained above, yielding a truth table for component 1 as in Table 2.3 (right). The separated states, by the value of the new signal *csc*, are typed in bold face. There is no ambiguity anymore. From this table Boolean equations for lds, d and csc can be derived by using established Boolean minimisation techniques [CKK+02]. Thus, component 1 can be implemented by:

$$\begin{aligned} lds &= csc \vee d \\ d &= ldtack \cdot csc \\ csc &= dsr \cdot (csc \vee \overline{ldtack}) \end{aligned}$$

To ensure the speed independence of the resulting circuit each logic equation has to be implemented as an atomic gate, i.e. it must be free of combinational hazards [CKK+02, Ung69].

By applying SI logic synthesis for component 2, cf. Figure 2.4(e), one would get a simple wire implementation, i.e. $dtack = d$.

Observe that the shown decomposition of the VME bus controller w.r.t. the chosen output partition $\pi(Out) = \{\{lds, d\}, \{dtack\}\}$ could be considered as optimal, because applying *pure* logic synthesis from its initial specification would deliver the same logic equations, cf. the gate-level circuit in Figure 2.1. In contrast, the application of total decomposition, i.e. producing one output signal per component, would yield a less efficient, i.e. a larger, implementation w.r.t. chip area. Chapter 5 presents ideas how to find such improved output partitions.

2.3 Other Synthesis Approaches for Asynchronous Systems

Since the reader is now familiar with the basics of logic synthesis, other (state-of-the-art) synthesis methods of asynchronous systems will be presented, which partially incorporate (decomposition-based) logic synthesis into their design flows.[13]

In general, asynchronous systems can be considered as consisting of data and control paths. Asynchronous system design usually starts from a high level language specification for the entire system. During synthesis direct mapping techniques and logic synthesis are used to derive circuit implementations for several parts of the system. In particular for the synthesis of the (entire) control path logic synthesis is preferred.

The synthesis methods presented next are distinguished by their design entries, i.e. which language is used for the initial system specification or how it is mapped to further system models.

2.3.1 Synthesis using VHDL and Verilog

Hardware designers are used to apply established HDLs such as VHDL or Verilog which are dedicated for synchronous system design. Using them for asynchronous system design as well would considerably lower the learning effort for synchronous system designers. However, the specification of asynchronous characteristics might need elaborate descriptions, since VHDL and Verilog are tailored to the specific needs for synchronous system design. This usually leads to more complex code when specifying asynchronous systems [SY05].

Some approaches [BCK+04, CKLS06, Man06] try to exploit as much tool support from established synchronous design flows as possible. They use a standard tool for synchronous system design in order to get a synchronous netlist implementation (usually in a pipeline architecture style). Then by modifying this netlist the circuit is transformed into an asynchronous version. Due to this extensive synchronous pre-design the advantages of asynchronous technology cannot be exploited to a large extent. However, these approaches are successfully applied by some start-up companies, like Elastix [Ela] and Achronix [Ach]; Achronix promotes its high performance FPGAs and Elastix promises to reduce energy consumption of the chips significantly.

Further approaches go into another direction: The design-flow aims at asynchronous system implementation from the start. This opens more opportunities for circuit implementations and it enables the exploitation of the positive characteristics of asynchronous

[13] The content of this section is based on [WV09].

technology to a large extent. However, the corresponding tools PIPEFITTER [BL00] and PN2DC [BSKY04] only provide a *subset* of synthesisable VHDL and Verilog statements, so far [SY05]. Both tools start by splitting the system into its data and control path. Then data path synthesis is carried out by direct mapping, based on libraries containing asynchronous data path components. For synthesis of the control path either logic synthesis with PETRIFY is used (which suffers from complexity problems, as described in Section 2.2) or direct mapping of Petri net components to hardware by using the tool OPTIMIST [SBY03, BY02]. Indeed, the direct mapping approach does not suffer from complexity problems, but the resulting hardware is usually not very efficient, i.e. the performance, chip area and energy consumption cannot compete with the results from logic synthesis.

2.3.2 Synthesis via Syntax-Directed Translation (SDT)

Alternatively, synthesis using an HDL specific to the needs for asynchronous system design could be applied, such as the HDLs HASTE and BALSA. They are related to VHDL or Verilog. All of these languages are based on CSP (communicating sequential processes) [Hoa85], i.e. they are dedicated for concurrent system design.

On the one hand, designers using these specific HDLs need deeper knowledge about asynchronous technology as for the methods presented in Section 2.3.1. On the other hand, asynchronous characteristics can be specified much more adequate without overhead. BALSA and HASTE are well suited to synthesise efficient asynchronous circuits or systems.

HASTE is a further development of TANGRAM [vBKR+91, vB93] which was developed in the early 1990's at Philips. BALSA is rather similar to TANGRAM or HASTE and was developed by the 'Amulet Group' at the University of Manchester [BE97, Bar98, Bar00, EB02]. BALSA was developed for research purposes (i.e. non-commercial use) in order to investigate and explore synthesis methods for 'TANGRAM-like' specifications.

For synthesis, the EDA tool from Philips [vB93] as well as the BALSA-tool [Bal] use *syntax-directed translation (SDT)*: Each construct of the HDL can be implemented by a manually designed and highly optimised circuit, a so-called *handshake component (HC)*. For example, Figure 2.6(a) depicts the BALSA Sequence-HC and Figure 2.7(a) the BALSA Fetch-HC [Bar00]. The composition of all HCs via so-called *handshake channels* implements the entire *handshake circuit*. Figure 2.8 (bottom-left) shows a handshake circuit implementing an 8-bit single place buffer resulting from SDT of the BALSA code at the top of this figure, cf. also [EBJ+06]. The hardware shown at the bottom-right of this figures shows the resulting circuit after applying direct mapping

for each HC to its corresponding manually designed circuit.

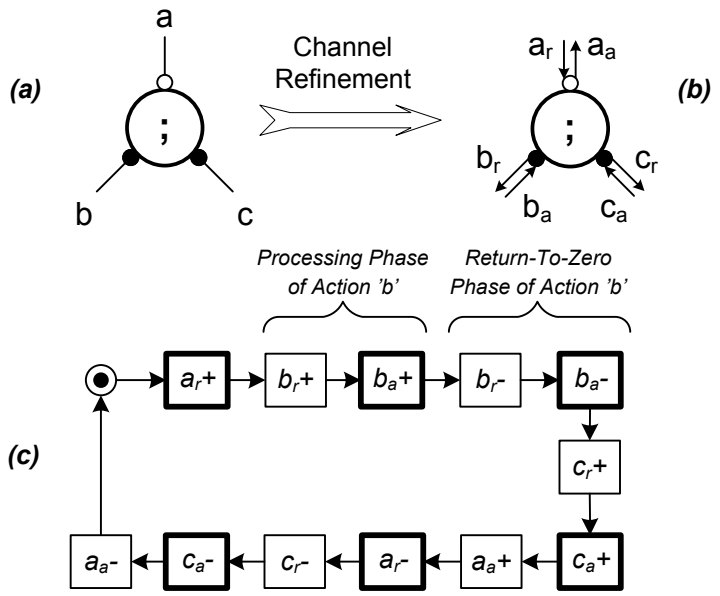


Figure 2.6: Balsa Sequence-HC

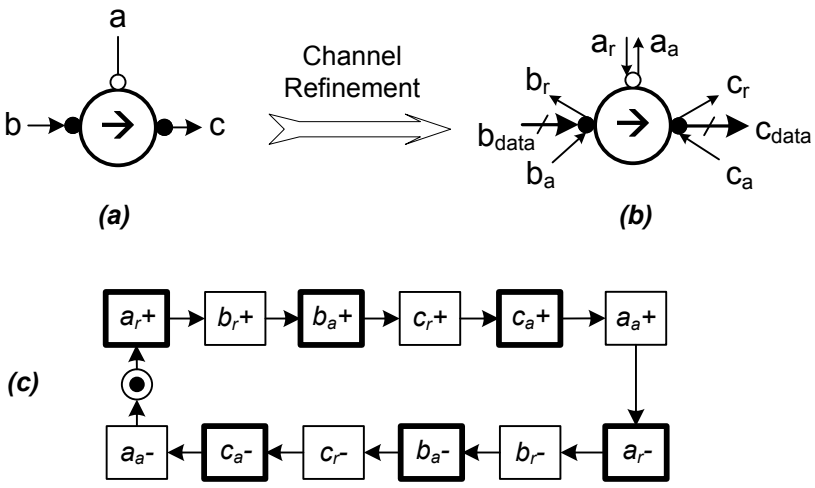


Figure 2.7: Balsa Fetch-HC

Handshake channels are implemented by handshake signals realising the necessary asynchronous event-based communication initiated from an *active port* of one component (depicted by a small black-filled circle) and responded by a *passive port* of another component (a small white-filled circle). As one can see in Figure 2.6(b) each handshake channel from (a) is implemented by two handshake signals: one for **r**equest and one for **a**cknowledgement. The interface behaviour of the Sequence-HC is specified by the

```
import [balsa.types.basic]
procedure buffer1 (input i : byte; output o : byte) is
   variable x : byte
begin
   loop
      i -> x     -- Input communication
      ;          -- Sequence operator
      o <- x     -- Output communication
   end
end
```

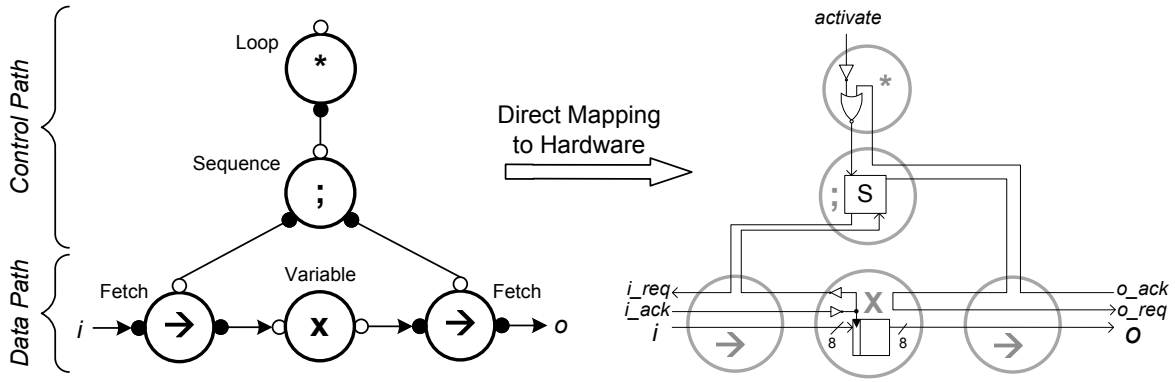


Figure 2.8: Balsa specification of an 8-bit single place buffer (top) and the resulting handshake circuit after SDT (bottom-left) as well as its hardware implementation after applying direct mapping for each HC (bottom-right).

STG in (c): It waits for a handshake on its passive port and performs one handshake each on its active ports to execute the actions 'b' and 'c'.

Often, so-called *4-phase handshakes* are executed, by performing a processing phase for an action followed by a return-to-zero (RTZ) phase. The *processing phase* will be initiated by a rising signal edge on the request line and completed by a rising edge on the acknowledge line. The *return-to-zero phase* will be initiated by a falling signal edge on the request line and completed by a falling edge on the acknowledge line. The correct operation of the Sequence-HC requires that the processing phases of both actions 'b' and 'c' are executed in sequence.

Some HCs, e.g. the Fetch-HC in Figure 2.7, may transport data (via data channels) besides processing the handshake. Thus, each channel which transports data is implemented by additional *data wires*, as shown in Figure 2.7(b). For data channels the direction of the data flow is indicated by an arrow, as for the channels labelled with 'b' and 'c' in Figure 2.7(a). The handshake protocol for the Fetch-HC is modelled by the STG in Figure 2.7(c).

In Balsa implementations often 4-phase handshake protocols and so-called *early data processing* is used, i.e. data processing merely takes place during the processing phases of the handshake, but not during its RTZ phases. (Observe that RTZ phases are pure control overhead.[14]) When a handshake component executes this protocol via some channel the values carried on the related data wires must be stable during the entire processing phase.

The partition of data and control path for some handshake circuit is not explicit. Figure 2.8 (bottom) shows one simple option to do such a partition: There are handshake components considered as *control HCs*, like the Loop- and Sequence-HC, forming the control path of the system and others considered as *data HCs*, like the Variable- and Fetch-HC, forming its data path. However, one could consider the handshake protocol part of each data HC as control as well. This might be more important for HCs having a mixed data-/control-processing nature such as the While-HC [Bar00], which basically realises a *While* loop, as in high level programming languages. In general, the more control can be identified (i.e. the larger the control path), the larger the optimisation space when applying logic synthesis to synthesise the system's control path, as presented later in Section 2.3.3.

The interconnection of HCs for some handshake circuit directly follows the syntactical flow of the HDL specification, cf. Figure 2.8.

[14] The application of so-called 2-phase handshake protocols can avoid this overhead, but this yields larger hardware implementations [Bar00].

Since circuit structure and source code are directly related, SDT allows the synthesis of very large asynchronous systems following a modular design principle, cf. Figure 2.8. On the one hand, there are no complexity problems; the translation of the specification to hardware is transparent and the resulting circuit structure is 'easily' comprehensible. On the other hand, SDT might lead to large and slow circuits; they might be heavily over-encoded due to the extensive use of handshake communication.

However, Philips Semiconductors successfully applied SDT for the development of its first commercial asynchronous product (1998: an asynchronous version of an 80C51 micro-controller [vGvBP+98] has been developed in order to design a low-power pager [vBJN99]).

2.3.3 Control Resynthesis and other Improvements for SDT

As mentioned, synthesis using SDT, as presented in the latter section, might lead to inefficient and heavily over-encoded circuits. This section presents opportunities for improving the SDT results.

Control Resynthesis

This kind of SDT-improvement, w.r.t. a Balsa design entry, is the main motivation for this work. It is a promising approach to reduce the control overhead by eliminating the over-encoding for the control path. The idea of *control resynthesis* is to replace the 'SDT synthesised' control path of the system by a 'logic synthesised' version, i.e. the control path will be *re*-synthesised by applying logic synthesis.

As mentioned, one way to split the system into control and data path is to partition the handshake components into control components (control HCs) and data components (data HCs). Figure 2.9 (left) sketches a handshake circuit, where the grey circles represent control components and the white ones data components. The data-path (i.e. all data components) of the resulting circuit is implemented by direct mapping, cf. Figure 2.8, i.e. the sub-circuits implementing the data HCs can be taken from the Balsa-library without modifications. However, the control path can be improved by applying resynthesis. Connected control HCs (highlighted by the grey borders in Figure 2.9 (left)) are combined to *control clusters*, as shown in Figure 2.9 (right), which should consist of as many handshake components as possible. The idea of control resynthesis is to remove all 'unnecessary' internal handshake signals (represented by the dashed arcs) from such a control cluster such that its external behaviour remains unchanged. This eliminates the over-encoding. Then for each control cluster an improved circuit can be synthesised via logic synthesis. This promises a significant area

reduction of the system's control path; the larger the clusters, the more area reduction can be achieved.

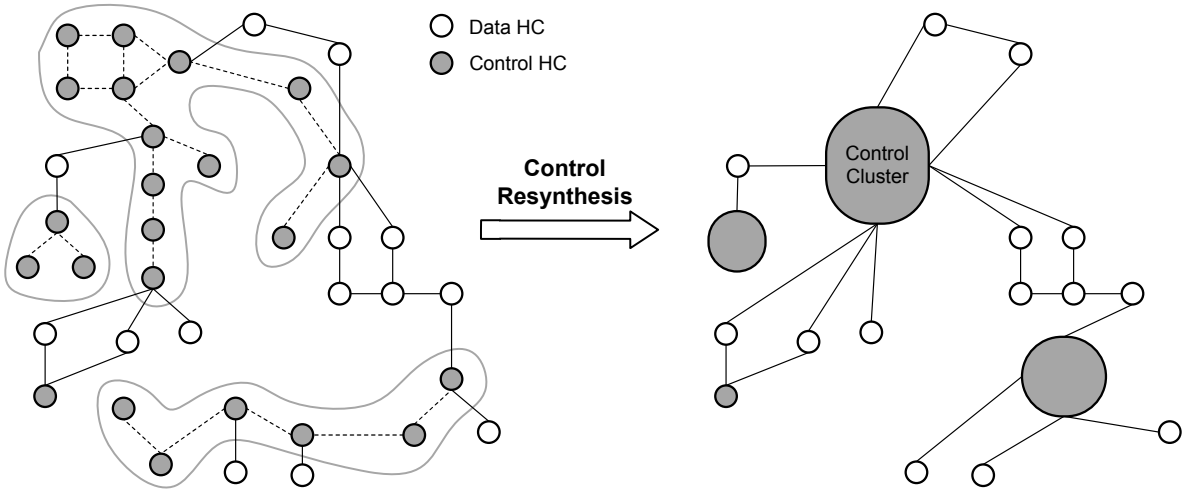


Figure 2.9: Control resynthesis for a handshake circuit

First results for control resynthesis applied to a memory controller are presented in [KVL96]. All control HCs are combined into *one* cluster; this saves 60% of gates and memory elements for the final control implementation (the relation between control and data HCs was 38:52). Since pure logic synthesis suffers from complexity problems, often several smaller clusters must be used, instead of one huge cluster.

Pioneering work can also be found in [CNBE02, CNBE03]. It uses burst-mode machines for specifying the interface behaviour of a cluster and MINIMALIST [FNT+99] as logic synthesiser. For several examples (in particular for a 32-bit microprocessor core) performance improvements from 8% to 54% were achieved, but at the expense of area, i.e. up to 63% enlargement. There are several reasons for this area enlargement. The logic synthesis using MINIMALIST was tuned to speed improvement which might increase area consumption. The technology mapping step was 'naively' implemented resulting in significant area overheads [CNBE02]. In contrast, the original HCs are manually designed and they have highly optimised implementations. Furthermore, the size of the control clusters might be too small to exploit the advantages of logic synthesis, because the smaller the clusters the less over-encoding (i.e. the less unnecessary internal handshakes) can be removed. However, complexity problems in logic synthesis limit the size of the clusters.

Decomposition-based logic synthesis, as introduced in Section 2.2.1, can be applied to tackle complexity. Carmona et al. present in [FC08] first results for an STG decomposition based resynthesis approach for handshake circuits. Each control component is

replaced by an STG modelling its interface behaviour (HC-STG). So far, such an HC-STG is only given for a small subset of possible components. All HC-STGs of a cluster of connected control components are composed by parallel composition according to the netlist structure.

Then all transitions representing inter-component handshake signals are contracted such that only the external behaviour remains, cf. also the dashed arcs in Figure 2.9(a). Considering the example in Figure 2.8 (bottom), the Loop (*) and the Sequence (;) component form a control cluster as shown in isolation in Figure 2.10 (left). Both corresponding HC-STGs (right) are composed yielding the STG in Figure 2.11(a). (In fact, for this simple example the parallel composition of the Loop and Sequence HC basically corresponds to the Sequence HC again, but it has three additional *redundant* places which are depicted by the grey arcs.) To remove the over-encoding, all internal handshakes are hidden by labelling the corresponding transitions with λ (b) which then will be removed by applying transition contraction (cf. Definition 3.2.1 below) yielding the corresponding *cluster STG* (c).

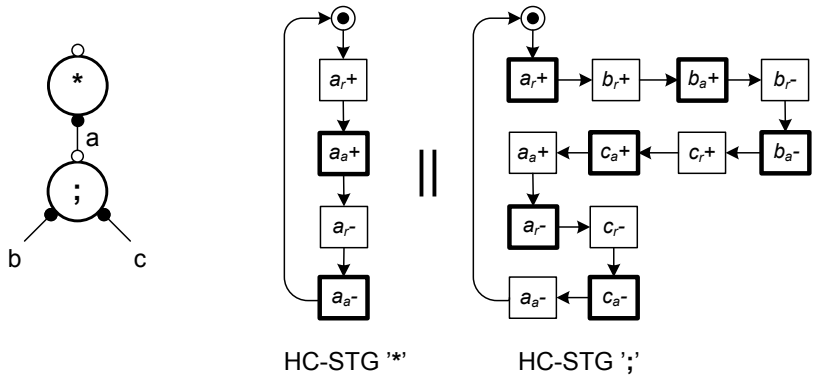


Figure 2.10: Control cluster example

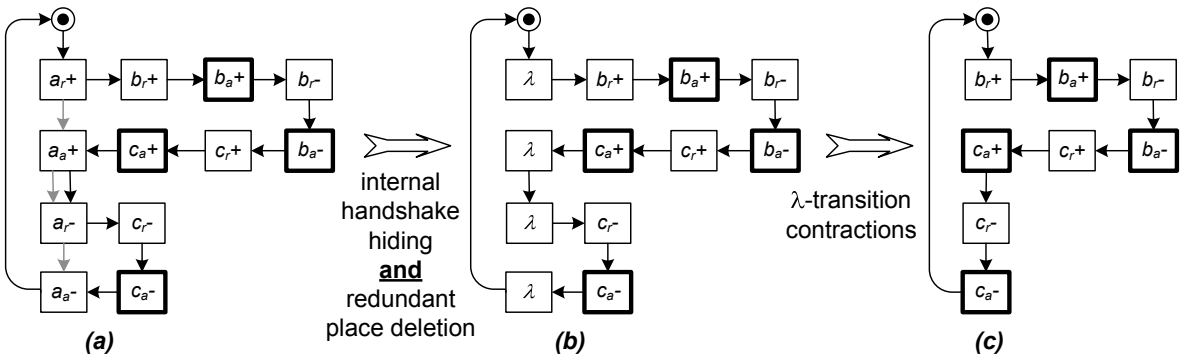


Figure 2.11: Construction of the cluster STG w.r.t. Figure 2.10

In general, the resulting cluster STGs might be too complex such that pure logic synthesisers cannot handle them. In [FC08] decomposition-based logic synthesis according to [CCCGV06] is used to handle complexity. As mentioned above, it requires a specification having CSC. In [CC08], a technique based on ILP solving was suggested which on the one hand suffers from complexity in solving $\mathcal{NP}$-complete ILP problems. On the other hand, this method is incomplete, i.e. even if a solution exists, there is no guarantee to find it. Thus, the cluster building in [FC08] is tuned to form 'well-structured' cluster STGs for which the ILP-based CSC solving method hardly fails. Logic synthesisers such as Petrify can synthesise the components yielding the final SI controller. This resynthesis method was integrated into the Balsa design flow. For the control paths of five benchmarks the area consumption was reduced from 10% to 40%, but there is nothing stated about the resulting performance.

In this thesis, a different decomposition-based logic synthesis approach is proposed which does not require CSC satisfaction for the initial specification and which is therefore more flexible. Thus, the corresponding decomposition tool DesiJ (see Chapter 6) might support control resynthesis much better, since larger cluster STGs can be handled. For this, DesiJ can be incorporated into an EDA system based on Balsa synthesis, as presented in Figure 2.12.[15]

The proposed EDA framework shows how the tools Balsa, Petrify, Punf, Mpsat and DesiJ can interact to efficiently produce asynchronous systems from complex Balsa specifications.

As a first step, the Balsa program will be translated (syntax-directed) into a handshake circuit consisting of interconnected HCs. Next, the control path of this handshake circuit will be extracted as sketched above and will be clustered into n control clusters (cf. Figure 2.9) such that each cluster is suitable for the subsequent logic synthesis. The remaining part of the handshake circuit forms the data path. For its synthesis the conventional Balsa backend *balsa-netlist* can be used, i.e. each HC is directly mapped to its manually designed implementation, cf. Figure 2.8 (bottom). Each control cluster has to be translated into a cluster STG – by parallel composition of the HC-STGs and internal handshake hiding, i.e. contraction of all transitions representing internal signals of this cluster (cf. figures 2.10 and 2.11).

From each such an usually complex cluster STG an efficient circuit implementation can be synthesised by using a decomposition-based SI logic synthesis system, as presented in Figure 1.1(b). Thus, DesiJ can be used for decomposing each cluster STG.

[15] It is an FMC model consisting of active system components (called *agents*) as well as passive ones (*storages*) [KGT06]. Agents and storage elements are depicted as rectangles and ovals resp.

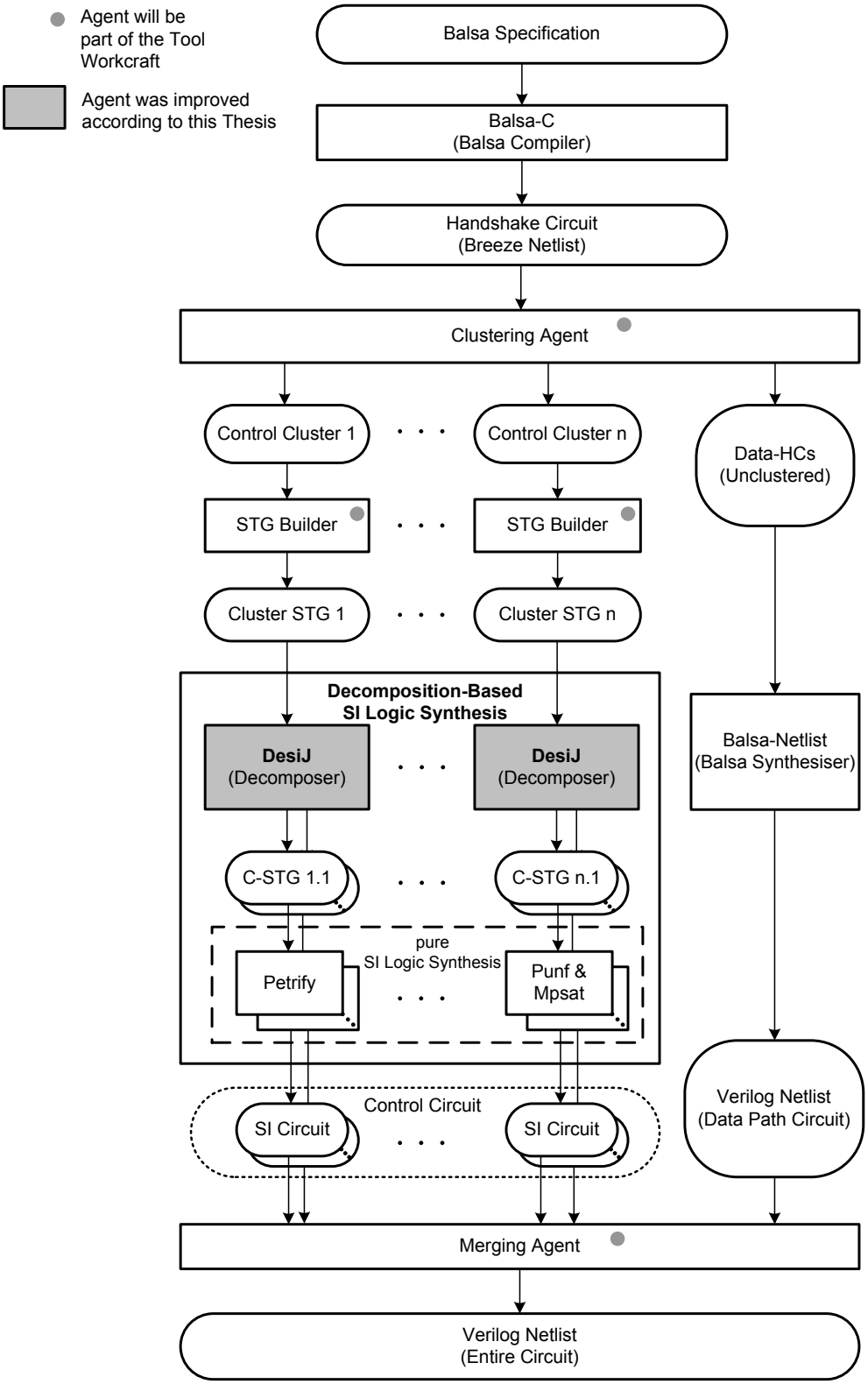


Figure 2.12: Proposed EDA framework for a Balsa-based design-flow using control resynthesis. It incorporates decomposition-based SI logic synthesis, as shown in Figure 1.1(b).

By this way, a complex cluster STG i ($i \in 1...n$) will be decomposed into several smaller component STGs (C-STG $i.j$ with $j \in \mathbb{N}$) which then usually can be handled by pure logic synthesis using PETRIFY or PUNF&MPSAT. Finally, the merging agent combines all the SI implementations of the control components with the data path netlist, yielding the desired asynchronous system. First promising results for control resynthesis using DESIJ will be discussed in sections 4.5 and 5.5.

If decomposition-based logic synthesis for some cluster STG still fails, direct mapping based synthesis from this cluster STG can be applied, i.e. particular STG fragments are directly mapped to hardware [SBY03, BSKY04]. However, the resulting circuits are usually not as efficient as the corresponding logic synthesised versions.

Alternatively, the failure can be reported to the clustering agent which decides to synthesise either the entire cluster or merely some HCs of the cluster by *balsa-netlist* and initiates synthesis of the remaining part of the cluster via decomposition-based logic synthesis again. Of course, the application of such *direct* synthesis techniques should be the last resource, because they might destroy the desired efficiency improvement for the resulting control path.

For the agents not marked with a grey dot there are rather mature tool implementations available. Recall, DESIJ will be discussed in Chapter 6. The other agents are under construction. The tool WORKCRAFT [PKY09, Wor] from the Newcastle University tries to complete this EDA system by implementing all agents labelled with a grey dot.

In this thesis, STG decomposition in the context of control resynthesis is focused. The following two subsections present other improvements for SDT. The first one can be combined with resynthesis as well. The second one is orthogonal to control resynthesis, but there are very few results so far.

Improvement for SDT: Handshake Component Optimisations

Often used HCs are improved in terms of their interface specification and implementation such that the communication with adjacent HCs is much faster in comparison when using the original HC versions. For instance, an improved version of the Sequence HC in Figure 2.6 may behave as specified by the STG in Figure 2.13. Indeed, the correct operation of the Sequence component requires that the processing phases of the actions 'b' and 'c' are executed in sequence, but this does not hold for the return-to-zero (RTZ) phases, cf. Section 2.3.2 and [PTE05]. The improved Sequence HC starts the processing phase of the second command 'c' as soon as the processing phase of the first one 'b'

is finished. This reduces latency and allows the concurrent execution of the processing phase of 'c' and the RTZ phase of 'b'.

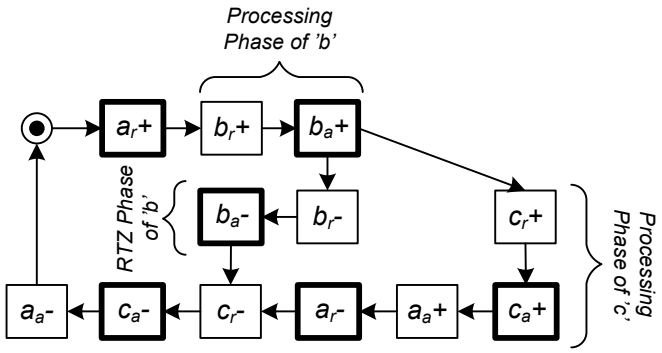


Figure 2.13: Improved Sequence component (cf. Figure 2.6)

For example, such a replacement of the original HCs with their improved versions doubles the performance of an asynchronous 32-bit processor [PTE05]. However, no correctness proof is given there in order to show that this HC replacement does not change the desired interface behaviour of the entire handshake circuit.

Note that control resynthesis can also be applied for handshake circuits containing these improved HC versions.

Improvement for SDT: Towards Pipelining

The SDT approach leads to handshake circuits usually containing complex trees of control HCs. Each data item, necessary for a computation, must explicitly be requested (*pull style operation*). Thus, the control flow is often slower than the data flow and slows down the entire circuit performance.

[Tay07] and [TEP08] try to reduce this control overhead by introducing new HCs which process data at the moment they arrive, i.e. without even knowing whether this data processing is needed or not (*push style operation*). The resulting circuits behave like a pipeline. Taylor et al. can improve the performance by 60%, but it causes an increase in area (up to 200%) and energy consumption. The latter shortcoming is a real drawback since handshake circuits are actually well-known for low energy consumption [vGvBP+98].

However, these improvements require adaptations of the Balsa language. Furthermore, its application requires more special knowledge about asynchronous circuit design than original Balsa, which is easier to learn for synchronous circuit developers, because of its similarities to VHDL and Verilog.

[BTE09] and [PtBdWM10] present synthesis methods for asynchronous pipeline circuits from *original* Balsa and Haste specifications. The final circuit architecture follows the underlying data-flow instead of the HDL specification syntax. [PtBdWM10] can triple the throughput for their benchmarks as well as they can even achieve significant improvements in power consumption in comparison to the application of SDT (as in Section 2.3.2). These improvements can be reached when designing purely data-driven circuits (e.g. a FIFO).

Unfortunately, there are merely very first results available [BTE09, PtBdWM10] and no further research is done so far. Thus, it is unclear whether an improvement in comparison to control resynthesis can be reached, especially when considering more control-driven or even control *dominated* circuits.

2.3.4 Syntax-Directed STG Generation

Finally, in this subsection and the next one two further synthesis methods for asynchronous systems will be presented. In fact, they will only be sketched, since they are not focused in this thesis.

Yoneda et al. [YMKM05] derive a control STG from a Balsa specification according to its syntax. For each Balsa control statement an STG fragment is given. *Elementary control statements* such as active read or register write can be distinguished from *combined control statements* such as sequence, parallel or while statements. An STG modelling the control of a whole procedure or program is generated inductively, by a hierarchical composition of several elementary or combined STG fragments according to the structure of a higher level combined STG fragment. This STG construction is merely intuitively (but not formally) presented. The generated STG satisfies CSC by construction and is subject to logic synthesis. To handle complexity, a simple form of STG decomposition is used [YOM04, YM07], cf. Section 2.2.1.

In future research, the syntax-directed STG generation should be formalised and the resulting STGs have to be compared with the ones resulting from control resynthesis.

Using a prototype tool implementation Yoneda et al. show that for specific benchmarks a three times performance improvement for the controllers can be reached as well as a reduction of gates by 25% compared to the application of SDT (as in Section 2.3.2). However, this method is restricted to a Balsa *subset* and so far there is no library for the specific data path components used there. In future work, it should be investigated whether these specific data path components can be replaced by Balsa data HCs.

2.3.5 CHP-based Methods

Besides Balsa and Haste the language CHP (communicating hardware processes) [Mar89] can be used for the initial system specification. (CHP is a variant of CSP [Hoa85] designed for communicating hardware.) Asynchronous system synthesis inspired by [MM98, MLM99, WM01, WM03] does not follow the specification structure (i.e. it is not syntax-directed), but it follows the data flow. The system specification is decomposed into a more concurrent version which is semantically equivalent under certain assumptions; in particular, all unnecessary synchronisation is tried to be removed, while elementary data dependencies are preserved.

The Caltech synthesis tools (CAST) have been used to synthesise some high performance circuits [MLM+97], but with significant manual intervention. For automatic synthesis particular requirements are necessary which might be difficult to understand and to meet even by experienced designers [Tay07].

3 Basic Definitions

This chapter provides basic notions for Petri nets [Pet62, Sta90, Rei10], STGs [Wen77, Chu87], speed independent logic synthesis [CKK+02] as well as STG decomposition [VW02, VK07, Sch08] which are necessary for chapters 4 and 5. This chapter basically summarises the basic definitions sections from [WWSV09, WSVW10, WVW11].

Definition 3.0.1 (Multiset)
A *multiset ms over a set A* is a function $ms : A \to \mathbb{N}_0$. Let ms and ms' be multisets over A; then for $a \in A$:

- $a \in ms \Leftrightarrow ms(a) > 0$
- $(ms + ms')(a) = ms(a) + ms'(a)$
- $(ms - ms')(a) = \max(0, ms(a) - ms'(a))$

A subset A' of A is considered implicitly as multiset $ms_{A'}$ with $ms_{A'}(a) = 1$ if $a \in A'$ and $ms_{A'}(a) = 0$ otherwise. △

3.1 Petri Nets

A (labelled) *Petri net* is a 6-tuple $N = (P, T, W, M_N, \Sigma, l)$ where P and T are disjoint and finite sets of *places* and *transitions*. $W : P \times T \cup T \times P \to \mathbb{N}_0$ is the *weight function* and M_N the *initial marking*, where a *marking* is a multiset of places, i.e. a function $P \to \mathbb{N}_0$ which assigns a number of *tokens* to each place. The marking of a

set of places is defined as the sum of all individual markings. Σ is a set of *actions*, and $l : T \to \Sigma \cup \{\lambda\}$ is the *labelling function* where λ denotes the empty word. If necessary, P_N, T_N etc. for the components of N is written, and P', P_i etc. for the nets N', N_i etc. Analogous conventions apply later on.

Graphical Representation:

A Petri net can be considered as a bipartite graph with weighted and directed edges between places and transitions. In a graphical representation, places are drawn as circles containing a number of tokens corresponding to their marking, transitions are drawn as rectangles together with their labelling, and the weight function is drawn as directed arcs xy whenever $W(x, y) \neq 0$ (and labelled with $W(x, y)$ if $W(x, y) > 1$). A place p is called *marked graph place* or *MG-place* if $\sum_{t \in T} W(t, p) = 1 = \sum_{t \in T} W(p, t)$. Unmarked MG-places are not drawn; they are implicitly given by an arc between the respective transitions; cf. Figure 3.1 below.

Structural Notions:

The *preset* of a place or transition x is denoted as ${}^\bullet x$ and defined by ${}^\bullet x = \{y \in P \cup T \mid W(y, x) > 0\}$, the *postset* of x is denoted as $x^\bullet$ and defined by $x^\bullet = \{y \in P \cup T \mid W(x, y) > 0\}$. One can say that there is an *arc* from each $y \in {}^\bullet x$ to x.

For a place or transition x, $N \setminus x$ denotes the net in which x and all incident arcs are deleted; for a marking M, $M|_{P'}$ denotes its restriction to $P' \subseteq P$, and $M|_{-p}$ is shorthand for $M|_{P \setminus \{p\}}$.

A nonempty sequence $w = x_1 x_2 \dots x_n$ of places and transitions without duplicates is a *path (of N)* if $W(x_i, x_{i+1}) > 0$ for $1 \leq i < n$. Obviously, places and transitions have to alternate on a path. With an abuse of notation a path will often be considered as the set containing its elements, writing for example $p \in w$. A path w is a *marked graph path* or *MG-path* if every place of w is an MG-place. For a marking M, the *marking $M(w)$ of a path w* is defined as $M(w \cap P)$. A path w is called *non-joining (non-forking resp.)* if for every transition t on w except the first (last resp.) one, $|{}^\bullet t| \leq 1$ ($|t^\bullet| \leq 1$ resp.). It is called *non-merging (non-branching)* if for every place p on w, $|{}^\bullet p| \leq 1$ ($|p^\bullet| \leq 1$). When connecting two paths $w_1 = x_1 \dots x_n$ and $w_2 = x_n \dots x_m$, $w_1 w_2$ for $x_1 \dots x_n \dots x_m$ will be written.

Dynamic Notions:

A transition t is *enabled under a marking M* if $\forall p \in {}^\bullet t : M(p) \geq W(p, t)$, which is denoted by $M[t\rangle$. An enabled transition can *fire* or *occur* yielding a new marking M', written as $M[t\rangle M'$, if $M[t\rangle$ and $M'(p) = M(p) - W(p, t) + W(t, p)$, for all $p \in P$.

A *transition sequence* $v = t_1 \dots t_n$ is *enabled under a marking M* (yielding M') if

$M[t_1\rangle\ M_1[t_2\rangle \dots\ M_{n-1}[t_n\rangle M_n = M'$, for this $M[v\rangle$, $M[v\rangle M'$ resp. is written; v is called *firing sequence* if $M_N[v\rangle$. The empty transition sequence λ is enabled under every marking. M is called *reachable* if a transition sequence v with $M_N[v\rangle M$ exists, and $[M_N\rangle$ is the set of all *reachable markings*.

A transition t is *2-live* if there is a firing sequence for every $n \geq 0$ which contains t n times; a transition t is *live* if every reachable marking activates a firing sequence containing t. A net is *2-live*, *live* resp. if each transition is 2-live, live resp.

N is called *bounded* if, for some constant $k \in \mathbb{N}_0$, $M(p) \leq k$ for every reachable marking M and every place p ; if $k = 1$, N is called *safe*. N is bounded if and only if the set $[M_N\rangle$ of reachable markings is finite.

Transition Labels:

The notion of enabledness can be lifted to transition labels as well: $M[l(t)\rangle\rangle M'$ can be written if $M[t\rangle M'$. This is extended to sequences as usual – deleting λ-labels automatically since λ is the empty word; i.e. $M[a\rangle\rangle M'$ means that a sequence of transitions fires, where one of them is labelled with a while the others (if any) are λ-labelled.

A sequence $v \in \Sigma^*$ is called a *trace of a marking* M if $M[v\rangle\rangle$, and a *trace* of N if $M = M_N$. The *language of* N is the set of all traces of N and denoted by $L(N)$.

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

Definition 3.1.1 ((Transition-)Simulation)
For $A \subseteq T$, the *A-labelling* l_A is defined by $l_A(t) = t$ if $t \in A$ and $l_A(t) = \lambda$ otherwise; the idea is that the transitions outside of A can be regarded as unobservable.

A *simulation from* N_1 *to* N_2 is a relation $\mathcal{S}$ between markings of N_1 and N_2 such that $(M_{N_1}, M_{N_2}) \in \mathcal{S}$ and for all $(M_1, M_2) \in \mathcal{S}$ and $M_1[t\rangle M_1'$ there is some M_2' with $M_2[l_1(t)\rangle\rangle M_2'$ and $(M_1', M_2') \in \mathcal{S}$. A simulation is a *transition-simulation* if it is a simulation when using the labelling l_{T_1} for both N_1 and N_2 in case $T_1 \subseteq T_2$, the labelling l_{T_2} in case $T_2 \subseteq T_1$ resp.

A relation $\mathcal{B}$ is a *bisimulation* between N_1 and N_2 if it is a simulation from N_1 to N_2 and $\mathcal{B}^{-1}$ is a simulation from N_2 to N_1. If such a bisimulation exists, the nets are called *bisimilar*. Transition-bisimulations are defined analogously. △

If a simulation exists between N_1 and N_2, then N_2 can go on simulating all actions of N_1 forever; if a bisimulation exists, the nets can work side by side such that in each

stage each net can simulate the actions of the other.

Reachable Markings:

The *reachability graph* RG_N of a Petri net N is an arc-labelled directed graph on the reachable markings with M_N as root; there is an arc from M to M' labelled with $l(t)$ whenever $M[t\rangle M'$. RG_N can be seen as a finite automaton (where all states are accepting). In this thesis only *bounded* Petri nets which have finitely many reachable markings are considered. N is *deterministic* if its reachability graph is a deterministic automaton, i.e. if it contains no λ-labelled transitions and if for each reachable marking M and label $a \in \Sigma$ there is at most one M' with $M[a\rangle M'$. For deterministic nets, language equivalence and bisimulation coincide.

3.1.1 Marking Equation

Let N be a labelled Petri net, the matrix $I \in \mathbb{Z}^{|P|\times|T|}$ defined by $I(p,t) = W(t,p) - W(p,t)$ for all $p \in P$, $t \in T$ is called *incidence matrix* of N. For a firing sequence v of N, the vector $x = (x_1, \ldots, x_{|T|})$ such that x_i is the number of occurrences of t_i within v, for each $i \in \mathbb{N}, i \leq |T|$, is called *Parikh vector* of v.

Given a firing sequence v of N, where $M_N[v\rangle M_1$, $M_1(p)$ is equal to the tokens of $M_N(p)$, plus the tokens added by the transitions of ${}^\bullet p$ in v, minus the tokens removed by transitions of $p^\bullet$ in v, for all places $p \in P$. This can be written in the matrix form $M_1 = M_N + I \cdot x$, where x is the Parikh vector of v.

If a marking M_1 is reachable from M_N, then there is a firing sequence v such that $M_N[v\rangle M_1$, and the so-called *marking equation* $M_1 = M_N + I \cdot y$ has at least the Parikh vector of v as non-negative integer solution.

Note that the existence of such a solution is a necessary, but not a sufficient condition for M_1 to be reachable from M_N; i.e. if there is no solution, M_1 is not reachable from M_N, but the inverse does not hold in general. However, for certain subclasses of Petri nets (e.g. marked graphs, where every cycle is marked [Mur89]) the marking equation provides an exact characterisation of the set of reachable markings.

3.2 Signal Transition Graphs

A Signal Transition Graph (STG) is a tuple $N = (P, T, W, M_N, In, Out, Int, l)$, where $(P, T, W, M_N, Sig^{\pm}, l)$ is a Petri net, In, Out and Int are disjoint sets of *input*, *output* and *internal signals*, and $Sig = In \cup Out \cup Int$ is the set of all signals; *signature* refers to this partition of the signal set. $Sig^{\pm} = Sig \times \{+, -\}$ is the set of *signal edges* or *signal transitions*; its elements are denoted as s^+, s^- resp. instead of $(s, +)$, $(s, -)$ resp.

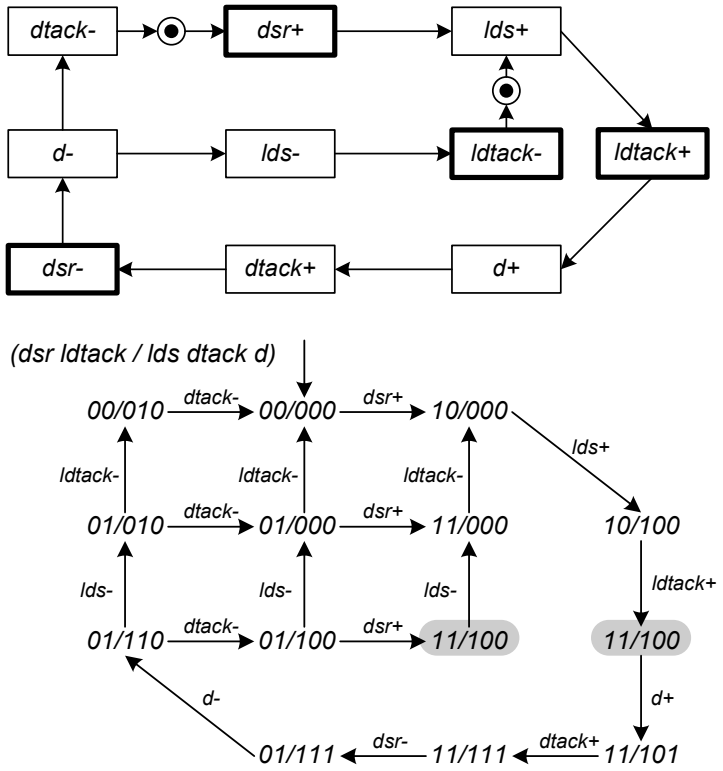


Figure 3.1: An STG modelling a simplified VME bus controller (top) and its state graph with a CSC conflict between the shaded states (bottom).

A plus sign denotes that a signal value changes from *logical low* (written as 0) to *logical high* (written as 1), and a minus sign denotes the opposite direction. We write $s^{\pm}$ if it is not important or unknown which direction takes place; if such a term appears more than once in the same context, it always denotes the same direction. The set of *locally controlled* or just *local signals* is defined as $Loc = Out \cup Int$, these are produced by the STG. The set of all *external signals* is defined as $Ext = In \cup Out$, these are observable in the environment.

To keep the notation short, input/output/internal signal edges are just called input/output/internal edges and each output/internal edge is also called *non-input edge*; transitions labelled with them are called input/output/internal transitions or *non-input transitions* resp. If transitions are labelled with λ they do not correspond to any signal change, i.e. they are not internal transitions; they are also called *dummy-transitions*. An example of an STG specifying merely causal relations between input and output edges is shown in Figure 3.1 (it models the interface behaviour of the simplified VME bus controller introduced in Section 2.2.2); recall, input transitions are drawn with a thick border.

3.2.1 SI Implementability

STGs are widely used for specifying the behaviour of asynchronous circuits, in particular the important subclass of speed-independent (SI) circuits. The idea is as follows: a reachable marking of the STG roughly corresponds to a state of the intended circuit (viz. the values of its signals). If some marking activates an output (or internal) edge, the circuit must produce the same edge if it is in a corresponding state and the environment of the circuit must be ready to receive it; if some marking activates an input edge, the environment is allowed to produce it and the circuit must be ready to receive it.

Consistency:

For the first step from markings to circuit states, one defines the notion of *state assignment*: for an STG N, a *state vector* is a function $sv : Sig \to \{0, 1\}$, which assigns a Boolean value to each signal. A *state assignment* assigns a state vector sv_M to each marking M of $[M_N\rangle$; it must satisfy for every signal $x \in Sig$ and every pair of markings $M, M' \in [M_N\rangle$:

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

If such an assignment exists, it is uniquely defined by these properties[1], and the reachability graph and the underlying STG are *consistent*. From an *inconsistent* STG, one cannot synthesise a circuit. The *state graph* of an STG is its reachability graph where each marking is annotated with its state vector; cf. Figure 3.1 (bottom).

Complete State Coding:

Now, the important concept of *Complete State Coding (CSC)* will be explained. If there is a state assignment, N has *CSC* if any two reachable markings M_1 and M_2 with the same state vector (i.e. $sv_{M_1} = sv_{M_2}$) enable the same output and internal signal edges. Otherwise, N has a *CSC conflict*, cf. e.g. Figure 3.1 (bottom), and no circuit can be synthesised directly. If CSC is violated, one tries to achieve it by inserting internal signals such that the state vectors of M_1 and M_2 differ and the external behaviour of the STG is unchanged; thus the internal signal insertion must be *input proper* [SV07], i.e. no input edge must be delayed by any internal edge (cf. also Definition 3.2.3 below).

If CSC cannot be achieved by an input proper signal insertion, the conflict is called *irreducible*. *(Dynamic) self-triggers* are a special type of irreducible CSC conflicts char-

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

acterised by a transition sequence $M_1[t_1 t_2\rangle M_2$, where t_1, t_2 are labelled with the same input signal, but complementary edges, and M_2 does not activate the same local signal edges as M_1. A *structural self-trigger* is defined as two transitions t and t' which are labelled with complementary edges of the same input signal satisfying $t \in {}^\bullet({}^\bullet t')$, and t is called the *entry* and t' the *exit transition*; a structural self-trigger is necessary for a dynamic one.

Output Persistency:

Output persistency guarantees the robustness (hazard freedom) of the desired SI circuit, i.e. when a signal disables another one, then both signals must be inputs; thus, each activated non-input edge will eventually happen. A circuit is called *speed-independent (SI)* if it is output persistent in all behaviours under a given environment. Thus the intended circuit will work correctly under arbitrary delays of gates (while the signal propagation is considered instantaneous). One can lift the notion of output persistency to the level of state graphs and STGs as well, see [CKK+02].

From the state graph of a bounded, consistent and output persistent STG satisfying CSC one can derive a speed-independent circuit [CKK+02], i.e. a Boolean function for each output or internal signal. This function has to be mapped to Boolean gates. Since this synthesis process needs a representation of the state graph, it suffers from the state space explosion problem [Val98]; there are tool implementations e.g. PETRIFY.

3.2.2 STG Operations

Now, a number of operations will be presented which are important for decomposition. A decomposition of an STG N is correct if the parallel composition of its components matches the behaviour of N. To define this formally, *parallel composition* will be introduced, and for this one has to consider the distinction between input and output signals.

Parallel Composition:

The notion of parallel composition is that the composed systems run in parallel and synchronise on common signals – corresponding to component circuits that are connected on wires corresponding to these signals. Since a system controls its outputs, it is not allowed that a signal is an output of more than one component; input signals, on the other hand, can be shared. An output signal of a component may be an input of other components, and in any case it is an output of the composition. Internal signals of one component must not be used by the others; this is no serious restriction and can always be achieved by a suitable renaming of the respective signals. (A composition

can also be ill-defined due to what e.g. Ebergen [Ebe92] calls *computation interference*; this is a semantic problem, cf. the discussion after Definition 3.2.3.)

The parallel composition of STGs N_1 and N_2 is defined if $Out_1 \cap Out_2 = Int_1 \cap Sig_2 = Int_2 \cap Sig_1 = \emptyset$. The place set of the composition is the disjoint union of the place sets of the components; thus, markings of the composition (regarded as multisets) can be considered as the disjoint union of markings of the components. To define the transitions, let $A = Sig_1 \cap Sig_2$ be the set of common signals. If e.g. s is an output of N_1 and an input of N_2, then an occurrence of an edge $s^\pm$ in N_1 is 'seen' by N_2, i.e. it must be accompanied by an occurrence of $s^\pm$ in N_2. Since one does not know a priori which $s^\pm$-labelled transition of N_2 will occur together with some $s^\pm$-labelled transition of N_1, it has to be allowed for each possible pairing. Thus, the *parallel composition* $N = N_1 \parallel N_2$ is obtained from the disjoint union of N_1 and N_2 by combining each $s^\pm$-labelled transition t_1 of N_1 with each $s^\pm$-labelled transition t_2 from N_2 if $s \in A$. In the formal definition of parallel composition, $\star$ is used as a dummy element, which is formally combined e.g. with those transitions that do not have their label in the synchronisation set A. (It is assumed that $\star$ is not a transition or a place of any net.) Thus, N is defined by

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

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

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

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

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

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

Figure 3.2 shows an example of a parallel composition. For simplicity, a place of the composition is denoted as p instead of $(p, \star)$ or $(\star, p)$, and the same applies to unsynchronised transitions.

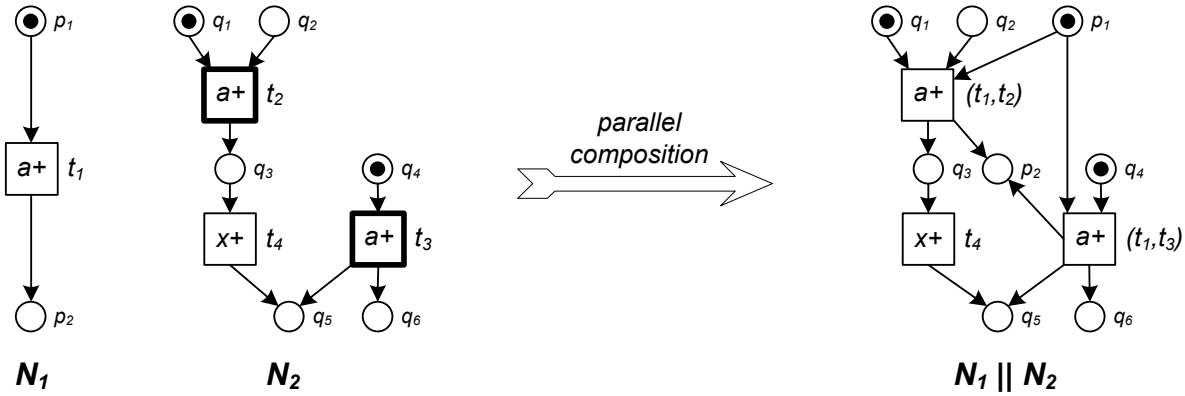


Figure 3.2: Parallel composition example. The two net fragments on the left share signal a, as an output in the left one and as input (twice) in the right one. Hence, in their parallel composition (right) a is an output.

According to [VK07], N is deterministic and consistent if N_1 and N_2 are. However, as illustrated in Figure 3.2, N might have structural auto-conflicts even if none of the N_i has them. The parallel composition of a finite family (or collection) $(C_i)_{i \in I}$ of STGs can be defined as $\|_{i \in I} C_i$, provided that no signal is an output signal of more than one of the C_i.

(De-)Lambdarisation and Hiding:

Next, the concept of *lambdarising* a signal will be introduced. It simply means to change the labelling function such that all transitions corresponding to this signal are labelled with λ and to remove this signal from the signature; *delambdarising* a signal means to restore the former labelling and signature. By contrast, *hiding* a signal set $H \subseteq Out$ from an STG N results in the STG $N/H = (P, T, W, M_N, In, Out \setminus H, Int \cup H, l)$, i.e. some output signals are now considered to be internal signals.

Transition Contraction:

The most important operation for decomposition is transition contraction (see e.g. [And83] for an early reference); it will essentially be repeated from [VK07], where further discussions can be found.

Definition 3.2.1 (Transition Contraction)
Let N be a Petri net and $t \in T$ with $l(t) = \lambda$, ${}^\bullet t \cap t^\bullet = \emptyset$ and $W(p,t), W(t,p) \leq 1$ for all $p \in P$. The *t-contraction* N' of N can be defined by

$$\begin{aligned} P' &= \{(p,\star) \mid p \in P \setminus ({}^\bullet t \cup t^\bullet)\} \\ &\quad \cup \{(p,p') \mid p \in {}^\bullet t, p' \in t^\bullet\} \\ T' &= T \setminus \{t\} \\ W'((p,p'),t_1) &= W(p,t_1) + W(p',t_1) \\ W'(t_1,(p,p')) &= W(t_1,p) + W(t_1,p') \\ l' &= l|_{T'} \\ M_{N'}((p,p')) &= M_N(p) + M_N(p') \end{aligned}$$

$$In' = In \quad Out' = Out \quad Int' = Int$$

In this definition, $\star \notin P \cup T$ is a pseudo element; we assume $W(\star,t_1) = W(t_1,\star) = M_N(\star) = 0$. Observe that $l|_{T'}$ is different from $l_{T'}$.

One can say that the markings M of N and M' of N' satisfy the *marking equality* if for all $(p,p') \in P'$

$$M'((p,p')) = M(p) + M(p').$$

For two different transitions t_1, t_2 with $t_1 \neq t \neq t_2$, the unordered pair $\{t_1,t_2\}$ is called 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 $l(t_1) = l(t_2) \neq \lambda$, it is called a *new structural auto-conflict*.

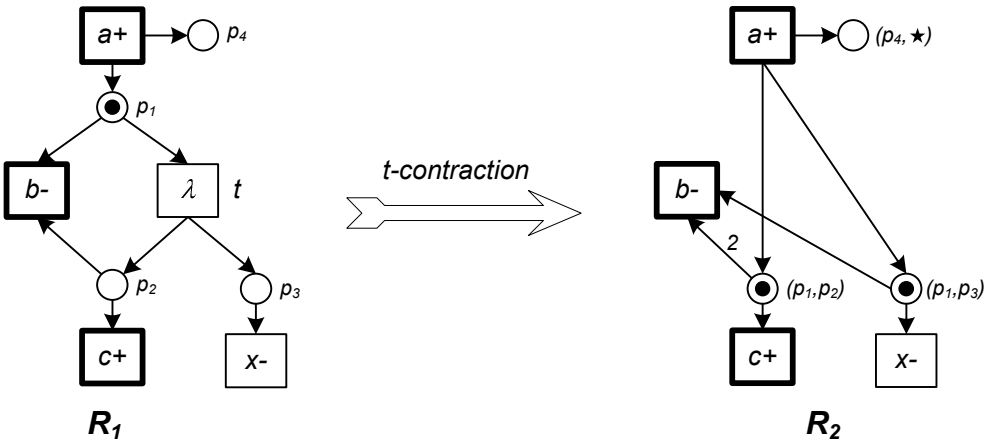


Figure 3.3: Example of a transition contraction in an STG.

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

Note that, in general, N' might fail to be consistent, even if N is; but secure contractions preserve consistency (see [VK07]).

Figure 3.3 shows a part of a net and the result of contracting the λ-transition, where

the b^-- and the x^--labelled transition form a new conflict pair; note that this is also true for b^- and c^+, although they are already in structural conflict in N.

Deletion of Redundant Transitions and Implicit Places:

Finally, redundant transitions and implicit places will be defined; the deletion of such a transition, place resp., (including the incident arcs) is another operation that can be used in the decomposition algorithm.

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

A place p is *implicit* if it can be removed from the net without changing the set of firing sequences. However, detecting implicit places is $\mathcal{PSPACE}$-complete[2] and during decomposition only redundant places (which are implicit) are deleted.

Definition 3.2.2 (Redundant Places)
The place q is *(structurally) redundant* [Ber87] if there is a set of places Q – called *reference set* – with $q \notin Q$, a valuation $V : Q \cup \{q\} \to \mathbb{N}$ and some $d \in \mathbb{N}_0$ which satisfy the following properties for all transitions t:

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

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

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

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

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

△

Observe that items (1) and (2) ensure that q is something like a linear combination of the places in Q with factors $V(s)/V(q)$. Indeed, for the case $d = 0$, (1) says that q is such a combination initially; (2), in the case of equality, says that this relationship is preserved when firing any transition. The proof that q is indeed redundant argues that initially the valuated token number of q is at least d larger than the valuated token sum on Q for all reachable markings, while the third item says that each transition needs at most d valuated tokens more from q than from the places in Q. This shows that for the enabling of a transition the presence or absence of q does not matter. Therefore, the

[2] The $\mathcal{PSPACE}$-completeness can be shown by reduction from the problem whether a place of a safe net can ever become unmarked; cf. [Esp98] for the complexity of the latter problem.

deletion of a redundant place in N turns each reachable marking of N into one of the transformed STG that enables the same transitions, hence the deletion gives a bisimilar STG – which is consistent if N is.

Redundant places are implicit (but in general not vice versa). Since the techniques for the detection of implicit and redundant places resp. are still not efficient enough, only the subsets of loop-only, duplicate and shortcut places are deleted.

An MG-place p is a *loop-only place* if p and t form a loop with arcs of weight 1 for all $t \in {}^\bullet p \cup p^\bullet$. Another simple case is that of a duplicate: place p is an (extended) *duplicate* of place q, if for all transitions t $W(t,p) = W(t,q)$, $W(p,t) = W(q,t)$ and $M_N(p) \geq M_N(q)$. An MG-place p is a *shortcut place* if there is an MG-path w between $t \in {}^\bullet p$ and $t' \in p^\bullet$ with $M_N(p) \geq M_N(w)$. In [SVJ05] it was shown that shortcut places are indeed redundant; they will be used later in the correctness proofs.

3.2.3 STG Decomposition

For the STG decomposition algorithm, from [VW02, VK07] a partition of the output signals of the given *specification* STG N is chosen, and the algorithm decomposes N into component STGs, one for each set in this partition. For synthesis, from each component equations for the corresponding outputs are derived from the respective state graph, instead of deriving the equations from the state graph of N.

Very often, the cumulated states of all component state graphs give a number much smaller than the state count of N, in which case the decomposition can be seen as successful. Actually, it might already be beneficial if each state graph is smaller than the one of N, in particular for reducing peak memory usage.

Of course, the behaviour of the specification should be preserved in some sense; this is captured by a variant of bisimulation, tailored to the specific needs of asynchronous circuits:

Definition 3.2.3 (Correct Decomposition [SV07])
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:
$(M_N, M_C) \in \mathcal{B}$ and 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 $M'|_{P_i}[x^{\pm}\rangle\rangle$, 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 $v \in (Int_N^{\pm})^*$.

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. If $H = \emptyset$, we simply say that $(C_i)_{i\in I}$ is correct w.r.t. to N. △

In a simple case, $(C_i)_{i\in I}$ consists of just one component C_1 (immediately implying (C2)). Thus the presented correctness definition also covers the question whether one STG is a correct implementation of another. For instance, this C_1 could be the result of solving a CSC conflict in N; if C_1 is correct w.r.t. N, then it has indeed the same behaviour as N.

Essentially, the correctness definition requires C to be bisimilar to N. The distinguishing features are the following. C is allowed to have fewer input and output signals than N; this is possible for input signals which are irrelevant for producing the right outputs and for outputs that actually never have to be produced. Observe that, correspondingly, C can match an input of N according to (N1) by doing nothing if C does not have this input; (N2) and (C1) guarantee that C will still produce exactly the required outputs from an external point of view. From an internal point of view outputs do not have to be matched directly; (N2) allows the components to prepare the production of an output by some internal changes, e.g. to achieve CSC or to inform other components; (C1) allows the specification to perform also internal signals. In contrast, input signals must be matched directly according to (N1), see the discussion below about speed independence and input properness.

There is no clause requiring a match for inputs of C; this implies that N and C are not necessarily language equivalent, as e.g. required in [Chu87]. The justification is that if N does not have a match for an input of C, then the environment is assumed not to produce this input according to the specification N; any behaviour of C after such an input is thus irrelevant for any well-behaved environment; see [VW02] for a detailed discussion why this feature is adequate and useful.

(C2) ensures that no computation interference occurs, i.e. if a component produces

an output (which is under the control of this component), then the other components expect this signal if it belongs to their inputs, and no malfunction of these other components must be feared. (C2) is actually also satisfied for $x \in Int_i$, since internal signals of one component are by definition unknown to the other components.

Finally, (N3) and (C3) prescribe the matching of an internal signal by a sequence of internal signals. Considering external behaviour the identity of an internal signal does not matter. However, performing an internal signal change could make a choice, e.g. by disabling an output; according to (N3) and (C3) this choice has to be matched.

A speed-independent circuit operates in *input/output mode* with its related environment, i.e. output changes are generated by the circuit as a consequence of certain input changes which in turn are reactions on certain output changes again. Hence, an STG specifying an SI circuit has to be *input proper*, i.e. an input might not be activated by an internal edge, otherwise the environment might produce the input before the internal signal is produced by the circuit. Alternatively, one could make timing assumptions for the environment, such that an internal signal change is always faster than an input change. In this case, the enabling of an input by an internal signal should not be interpreted as a causal but a temporal relation.[3]

In [SV07] it was shown that the above correctness notion implies that the STG C in Definition 3.2.3 is in a sense input proper; in particular, if the solution of a CSC-conflict inserts an internal signal in front of an input, it is not correct in the sense of Definition 3.2.3.

The following theorem specialises another result from [SV07] stating that so-called *hierarchical decomposition* results in a correct decomposition.

Theorem 3.2.4
Let $(C_i)_{i\in I}$ be a correct decomposition of an STG N' when hiding H', and let N' be correct w.r.t. N when hiding H. Then $(C_i)_{i\in I}$ is a correct decomposition of N when hiding $H \cup H'$.

The decomposition algorithm will now be discussed in more detail. In the following, it is assumed that a deterministic, consistent specification N is given without internal signals[4].

First, one chooses a *feasible partition*, i.e. a family $(In_i, Out_i)_{i\in I}$ for some set I such that the sets Out_i are a partition of Out, $In_i \subseteq Sig \setminus Out_i$ for each i and furthermore:

[3] This can be modelled by a so-called tcb-concurrency [WB00].
[4] For the decomposition algorithm, internal signals can be considered as outputs; see [SV07] for more details.

- If signal s and output x of N are in structural conflict, then $x \in Out_i$ implies $s \in In_i$ if $s \in In$ and $s \in Out_i$ if $s \in Out$ for each $i \in I$ (s is called *conflicting signal*).

- If there are $t, t' \in T$ with $t' \in (t^\bullet)^\bullet$ (t is called *syntactical trigger of* t'), then $l(t') \in Out_i$ implies $l(t) \in In_i \cup Out_i$.

Observe: if one has a feasible partition, one can build another feasible one by adding additional input signals to one of the members.

A decomposition is called *total* if it is guided by the *finest partition*, i.e. usually yielding components having one output each.

For each member (In_i, Out_i) of the partition, an *initial component* is generated from N: in a copy of the original STG N, every signal not in $In_i \cup Out_i$ is lambdarised and the signals in In_i are considered as inputs of this (initial) component – even if they are outputs of N. Then the following three *reduction operations* are applied to such an initial component until no more λ-labelled transitions remain:

- secure contraction of a λ-labelled transition

- deletion of an implicit place

- deletion of a redundant transition

Let us refer to this version of the algorithm as *CIR-algorithm*, where CIR stands for contraction, implicit and redundant.

Unfortunately, it is not always possible to contract all λ-transitions. Besides the technical cases where the contraction is not defined or not secure (possibly leading to an incorrect decomposition), the contraction might also generate a new auto-conflict. The latter reveals non-determinism which is present in the respective initial component, but not in the specification. This indicates that the component has not enough information to properly produce its outputs. Such a contraction is *disallowed* and consequently a new signal is added as follows.

If λ-transitions remain, *backtracking* is applied, i.e. a new input is added to the component. Technically, this input is added to the initial partition and the new corresponding initial component is derived and reduced from the beginning. The new input signal is taken from the former label of a non-contractible λ-transition. As discussed above, the new partition is feasible again. This cycle of reduction and backtracking is repeated until all λ-transitions of the initial component can be contracted.

In principle, every so-called *totally admissible operation* [VK07] can be used for reduction. It is proven in [VK07] that the decomposition algorithm using arbitrary totally admissible operations always returns a *correct* decomposition. The precise definition of totally admissible operation is not important here; it is enough to know the next proposition taken from [VK07], which gives a sufficient condition for an operation to be totally admissible.

Proposition 3.2.5 (Totally Admissible Operation)
An operation is admissible *if, whenever applied to an STG satisfying (a) and (b) below, it results in a bisimilar STG satisfying these properties again:*

(a) There is no structural λ/output conflict, i.e. between a λ-transition and one labelled with an output.

(b) No λ-transition is a syntactical trigger of an output-transition.

The operation is totally admissible *if additionally it decreases the termination function* (sc, tc, pc) *(with lexicographical ordering), where sc is the number of lambdarised signals (w.r.t. some STG N), and tc and pc are the numbers of transitions and places.*

4

Avoiding Irreducible CSC Conflicts by Internal Communication

This chapter contributes to adapt the STG decomposition method of [VW02, VK07] to SI logic synthesis. It is shown how irreducible CSC conflicts, caused by STG decomposition, can be avoided by introducing internal communication between the components.[1] In particular, it is shown how self-triggers, a special type of irreducible CSC conflicts, in component STGs can arise and how they can be avoided by introducing new internal communication signals, such that the external behaviour is unmodified. Then improvements are presented regarding

- the handling of general irreducible CSC conflicts in Section 4.2.1,
- new structural techniques for introducing internal communication by *correctly* inserting new internal signals into the an unweighted specification and decompose it anew, see Section 4.2 and Section 4.3, and
- an optimisation for simple structured specifications which avoids the second run of decomposition as well as its correctness proof, see Section 4.4.

Experimental results are presented in Section 4.5. Finally, the limitations and an outlook to future research are discussed in Section 4.6.

[1] The content of this chapter is based on [WWSV09, WSVW11].

First of all, the following section introduces some necessary results; in particular, some new operations on STGs are introduced, a generalisation of the applied decomposition algorithm as well as the corresponding correctness proofs.

4.1 Preliminaries

Here, some results are stated that are needed to apply the new method (presented in this chapter) and to prove it correct.

Proposition 4.1.1
Transition contraction, implicit place deletion and redundant transition deletion preserve liveness and 2-liveness.

In [KSVW09], liveness preservation is shown. The proof of the second part is analogous (even simpler) and therefore omitted. The idea is as follows: For the first two operations, there is a transition simulation between the respective nets. Hence, if there is a transition sequence containing a transition t n times in the original net, there is a corresponding sequence in the derived net (provided t is still present). For redundant transition deletion, observe that the firing of a loop-only transition does not change the marking and its deletion creates a net with an almost identical reachability graph, except that some loops are missing. For a duplicate transition, the same holds, except that some duplicate arcs (if they are distinguished) are removed.

The next result is is essentially taken from [VK07, KS07].

Proposition 4.1.2
A place p of N is implicit if and only if for all reachable markings M of N, $M|_{-p}[t\rangle \Rightarrow M[t\rangle$. If N' is obtained from N by deleting p, then $\mathcal{B} = \{(M, M|_{-p}) \mid M \in [M_N\rangle\}$ is a (transition-)bisimulation between N and N'.

Lemma 4.1.3
Let $\mathcal{S}$ be a transition-simulation between N_1 and N_2 for the common labelling l'. If $l' = l_{T_1}$, $\mathcal{S}$ is a simulation for any labellings l_1 and l_2 of N_1 and N_2 with $l_1(t) = l_2(t)$ if $t \in T_1$ and $l_2(t) = \lambda$ otherwise. Analogously for $l' = l_{T_2}$.

Proof. Clearly, $(M_{N_1}, M_{N_2}) \in \mathcal{S}$. So let $(M, M') \in \mathcal{S}$.

Since $\mathcal{S}$ is a transition simulation, $M[t\rangle M_1$ implies $M'[t\rangle\rangle M_1'$ due to $M'[vtv'\rangle M_1'$ with $l'(v) = l'(v') = \lambda$ and $(M_1, M_1') \in \mathcal{S}$. By definition, $l_2(vtv) = l_2(v)l_2(t)l_2(v') = \lambda l_1(t)\lambda = l_1(t)$. Hence, $M'[l_1(t)\rangle\rangle M_1'$, when labellings l_1, l_2 are used.

In case, $l' = l_{T_2}$, N_2 has no λ-labelled transitions. Hence, $M[t\rangle M_1$ implies $M'[l'(t)\rangle\rangle M_1'$ with $(M_1, M_1') \in \mathcal{S}$. If $t \in T_2$, this means $M'[t\rangle\rangle M_1'$ due to $M'[t\rangle M_1'$; hence $M'[l_1(t)\rangle\rangle M_1'$, when labellings l_1, l_2 are used. Otherwise, $M'[\lambda\rangle M_1' = M'$. □

Since a transition bisimulation is a simulation in both directions, the above lemma can also be applied to it.

Now a proposition will be stated that simply deals with implicit places and redundant transitions; it is needed to deal with the interplay of decomposition and the new method of signal insertion. Its formulation is quite subtle; observe that it does not say that after deleting one of two implicit places the other is still implicit. The latter is actually wrong: if there are two places which are duplicates of each other, one can be deleted but then the other may not be a duplicate anymore.

Proposition 4.1.4
Let N' be obtained from N by insertion of an implicit place p' (i.e. p' is implicit in N'). If p is implicit in N, then

(1) p is implicit in N' and

(2) p' is implicit in $N' \setminus p$.

Assume t is a redundant transition in N and not adjacent to p' in N' nor is its duplicate, in case that t is a duplicate transition; then

(3) p is implicit in $N' \setminus t$ and

(4) t is redundant in N'.

Proof. We check the condition of Proposition 4.1.2.

(1) If $M' \in [M_{N'}\rangle$, then $M'|_{-p}[t\rangle \Rightarrow M'|_{-p,p'}[t\rangle$ for $t \in T$. Since p is implicit in $N = N' \setminus p'$ and p' in N', this implies $M'|_{-p'}[t\rangle$ and $M'[t\rangle$.

(2) Each reachable marking of $N' \setminus p$ has the form $M'|_{-p}$ for $M' \in [M_{N'}\rangle$, which follows with (1) from the transition bisimulation of Proposition 4.1.2. If $M'|_{-p,p'}[t\rangle$ in $N' \setminus p$, then $M'[t'\rangle$ as above and thus $M'|_{-p}[t\rangle$.

(3) By Proposition 4.1.2, deleting transitions does not affect the implicitness of a place.

(4) Redundancy is structurally defined and the new place p does not change the neighbourhood of t and neither that of its duplicate if it exists; hence the claim follows.
□

Lemma 4.1.5
In a net N, let p be an implicit MG-place with ${}^\bullet p = \{t_1\}$ and $p^\bullet = \{t_2\}$. If t_2 is 2-live, there is a path from t_1 to t_2 in $N \setminus p$.

Proof. (Sketch) Assume otherwise. Consider a firing sequence w in $N \setminus p$ containing t_2 $M_N(p)+1$ times (since t_2 is 2-live such a sequence exists); consider the corresponding Petri net process π (cf. e.g. [Rei85]). From π all events (with their postsets) that do not have a path (in π) to a t_2-labelled event can be removed, resulting in a new process π' and a corresponding firing sequence w' which also contains t_2 $M_N(p)+1$ times.

Since paths in π correspond to paths in $N \setminus p$ (via the labelling of π), neither π' nor w' do contain t_1. Hence, t_2 can fire $M_N(p)+1$ times in $N \setminus p$ without firing t_1. This is not possible in N, a contradiction. □

4.1.1 Place Refinement and Subnet Contraction

In this section some new operations are introduced, which are shown to be compatible with decomposition, reduction resp. In particular, *gyroscope insertion* adds essentially what is known as toggle transition; it is the essential operation to introduce internal communication.

Definition 4.1.6 (Place refinement, subnet contraction and gyroscope insertion)
Let N be an STG.
(1) For a place $p \in P$, consider a net N' (cf. Figure 4.1) with:

- $P' = P \setminus \{p\} \cup \{p_{in}, p_{out}, p_1, p_2\}$
- $T' = T \cup \{g_1, g_2\}$
- $W'(x,y) = W(x,y)$ if $x, y \in P \setminus \{p\} \cup T$
  $W'(t, p_{in}) = W(t,p)$ for $t \in T$
  $W'(p_{out}, t) = W(p,t)$ for $t \in T$
  $W'(p_i, g_i) = W'(g_i, p_{3-i}) = W'(p_{in}, g_i) = W'(g_i, p_{out}) = 1$, $i = 1, 2$
- $M_{N'}(p') = M_N(p')$ for $p' \in P \setminus \{p\}$
  $M_{N'}(p_1) = 1$, $M_{N'}(p_2) = 0$
  $(M_{N'}(p_{in}), M_{N'}(p_{out})) = (in, out)$ with $in + out = M_N(p)$

In N', the labels of the new transitions and their signature can be chosen arbitrarily. Starting from N, N' is called a *place refinement of p with initial marking (in, out)*; starting from N', N is called a *subnet contraction* if g_1 and g_2 are λ-transitions.

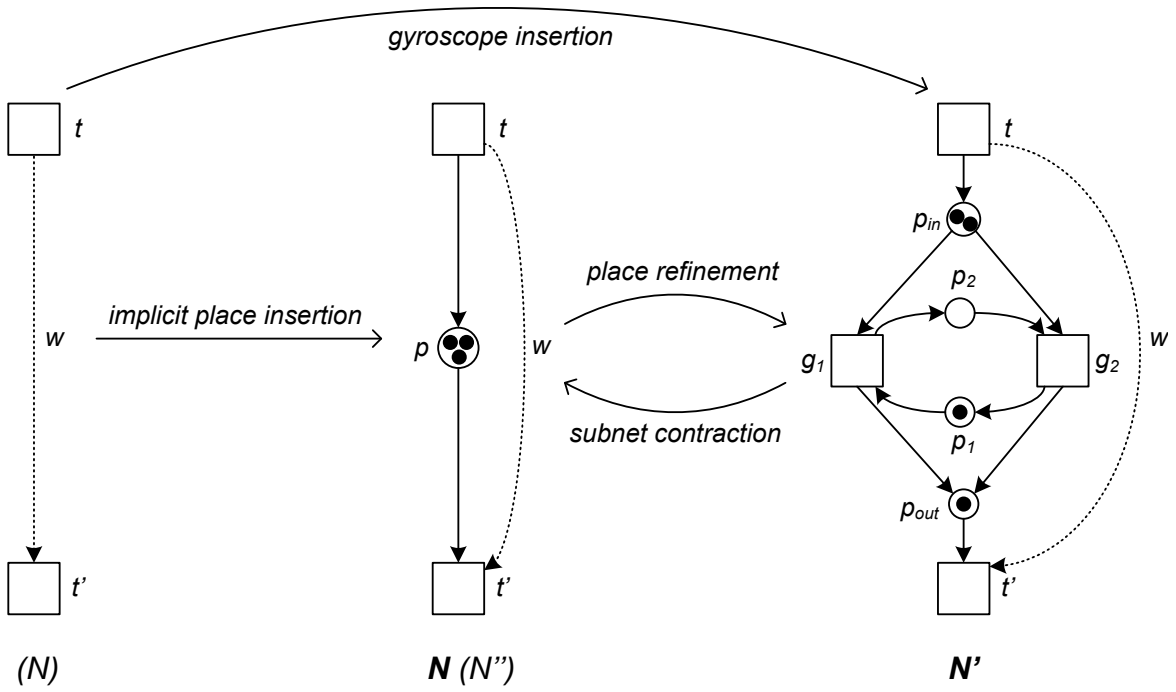


Figure 4.1: Gyroscope insertion between t and t' with initial marking $(2, 1)$ via place insertion (here: a shortcut place due to w) and place refinement.

(2) A *gyroscope insertion* with initial marking (in, out) inserts a new implicit place $p \notin P$ with $in + out$ tokens into N (giving the intermediate N'') and applies place refinement with initial marking (in, out) to it (giving N'); again cf. Figure 4.1. A *gyroscope insertion between* t and t' $(t, t' \in T)$ with initial marking (in, out) inserts this place p between t and t' (i.e. ${}^\bullet p = \{t\}$ and $p^\bullet = \{t'\}$ with arc weights 1).

A gyroscope insertion is called an *input/output/internal gyroscope insertion* if g_1 and g_2 are labelled in N' with s^+, s^- resp. and s is a fresh input, output or internal signal resp.; it is called a *dummy gyroscope insertion* if g_1 and g_2 are labelled with λ. △

Subnet contraction is used for the correctness proofs below and not really intended as reduction operation. But in principle, one could try to apply it if only backtracking is the alternative, although the odds for it to be applicable seem to be low:

Proposition 4.1.7
Let N' be a place refinement of N as in Definition 4.1.6.

(1) N and N' are transition-bisimilar.

(2) Subnet contraction is a totally admissible operation.

(3) If $p' \neq p$ is implicit in N, then p' is implicit in N'. If additionally $l'(g_1) = l'(g_2) = \lambda$, $N \backslash p'$ is the subnet contraction of $N' \backslash p'$ as in Definition 4.1.6.

Proof. (1) Obviously,

$$\begin{aligned}\mathcal{B} \;=\; & \{(M, M') \mid M'(\{p_1, p_2\}) = 1,\ M|_{-p} = M'|_{\hat{P}} \\ & \text{(with } \hat{P} = P \setminus \{p_1, p_2, p_{in}, p_{out}\}),\ M(p) = M'(\{p_{in}, p_{out}\})\}\end{aligned}$$

is a transition bisimulation between N and N'. Observe that to match a firing $M[t\rangle M_1$ of N, it might be necessary to fire g_1 or g_2 in N' first if $p \in {}^\bullet t$.

(2) One can apply Proposition 3.2.5: clearly, subnet contraction decreases the termination function; no new structural λ/output conflicts are created since the same transitions are in structural conflict (except for g_1 and g_2) and only the transitions in $p_{out}^\bullet$ get new triggers (${}^\bullet p_{in}$), but since they have the dummy transitions g_1 and g_2 as triggers in N', also 3.2.5(b) is preserved. (1) and Lemma 4.1.3 imply bisimilarity.

(3) The latter claim is obviously true. Regarding the first one, let $M' \in [M_{N'}\rangle$ and let M be the unique marking with $(M, M') \in \mathcal{B}$ from (1), and $M'|_{-p'}[t\rangle$. Since $p' \notin {}^\bullet g_1 \cup {}^\bullet g_2$, one has $M'[t\rangle$ for $t \in \{g_1, g_2\}$. Otherwise, ${}^\bullet t$ coincides in N and N' except that possibly p has to be exchanged with p_{out} if $t \in p^\bullet$ in N. Hence, $M|_{-p'}[t\rangle$ in N (since $(*)$ $M(p) \geq M'(p_{out}) \geq W'(p_{out}, t)$), $M[t\rangle$ by assumption and thus $M'[t\rangle$ due to bisimilarity. Observe that in this case, neither g_1 nor g_2 has to fire due to $(*)$ and $W'(p_{out}, t) = W(p, t)$. □

Proposition 4.1.8
Let N' result from N by a gyroscope insertion with initial marking (in, out) as in Definition 4.1.6.

(1) N and N' are transition-bisimilar.

(2) If $p' \neq p$ is implicit in N, then p' is implicit in N'. $N' \backslash p'$ is the gyroscope insertion of $N \backslash p'$, in particular p is a new implicit place for $N \backslash p'$.

(3) If the insertion is internal and for all $t' \in p_{out}^\bullet$ there is $l'(t') \in (Out \cup Int)^{\pm}$, then N is correct w.r.t. N' and vice versa.

Proof. (1) Let N'' be the result of the insertion of the new implicit place p, and let $\mathcal{B}_1$ be the transition-bisimulation between N and N'' as implied by Proposition 4.1.2. Proposition 4.1.7(1) implies that there is a transition-bisimulation $\mathcal{B}_2$ between N'' and N'. Transitivity of transition-bisimulations (cf. [KS07] for a slightly different version)

implies that $\mathcal{B} = \mathcal{B}_1 \circ \mathcal{B}_2$ is a transition bisimulation between N and N' (for the common labelling l_T).

(2) The first claim follows from Propositions 4.1.4(1) and 4.1.7(3), the second from Propositions 4.1.4(2) and 4.1.7(3).

(3) Consider $\mathcal{B}$ from (1). One easily checks the condition of Definition 3.2.3: in general, a transition firing in N is matched by the same firing in N' and vice versa, except that firings of g_1 and g_2 in N' are ignored in N and a firing of a $t' \in p_{out}^\bullet$ might be preceded by firings of g_1 and/or g_2. □

4.1.2 Quasi-Feasible Partition

Finally a new generalization of the decomposition algorithm will be presented; the point is that the definition of a feasible partition will be relaxed, i.e. that the same algorithm can essentially be applied with additional starting points.

Definition 4.1.9 (Quasi-Feasible Partition)
A *quasi-feasible partition* is a family $(In_i, Out_i)_{i \in I}$ for some set I such that the sets Out_i are a partition of Out, $In_i \subseteq Sig \setminus Out_i$ for each i and furthermore:

- If signal s and output x of N are in *dynamic* conflict, then $x \in Out_i$ implies $s \in In_i$ if $s \in In$ and $s \in Out_i$ if $s \in Out$ for each $i \in I$.
- If there are $t, t' \in T$ with $t' \in (t^\bullet)^\bullet$ (t is called *syntactical trigger of* t'), then $l(t') \in Out_i$ implies $l(t) \in In_i \cup Out_i$. △

In the definition of a totally admissible operation, see Proposition 3.2.5, condition (a) has to be changed to: there is no *dynamic* λ/output conflict. This modified (a) is also used as an invariant for the proof of the new correctness theorem for decomposition.

Theorem 4.1.10
When starting from a quasi-feasible partition, decomposition of a deterministic N (using totally admissible operations defined with the modified (a)) results in deterministic components that form a correct implementation of N. If only the operations listed in this thesis are used and N is consistent (free of dynamic io-conflicts resp.), then so are the components.

Proof. The proof follows the lines of the proof of Theorem 4.1 in [VK07]. Carefully checking the all in all eight (sub-)proofs, one sees that no essential changes are needed when working with the modified (a) and also considering only the really essential[2]

[2]for SI-synthesis

dynamic conflicts between an input and an output transition instead of the structural ones. There are two exceptions though: for proving that secure transition contraction is totally admissible, one has to fill two proof gaps for Lemma 4.3 and 4.4.

Now it will be sketched how this can be done and we start with Lemma 4.4. In the proof, all markings considered in angelic bisimulations are now assumed to be reachable – this is no problem. In case (c) for $i = j$, contraction is applied to a transition t of a component, it is called C, and the result is C'; C satisfies (the now modified) (a). A reachable marking of C, call it M, is related by the marking equality to a reachable marking M' of C'; an output transition t_1 is enabled under M', and one has to show that it is also enabled under M.

For a contraction of type 1, the proof in [VK07] is still valid, and one still can assume that ${}^\bullet t_1 \cap t^\bullet$ is empty. Assume that t_1 is not enabled under M; this must be due to places in the common preset of t and t_1 (so it clearly is a type-2 contraction). Let k be the maximal number of tokens on some of these places missing for enabling t_1.

Since $M'[t_1\rangle$, there must be at least k tokens on each place in $t^\bullet$ under M (so that each new place containing the 'critical' place from ${}^\bullet t_1$ allows t_1 to fire in C'). These tokens have been put there by t (a type-2 contraction is considered, and in particular some place in $t^\bullet$ was empty initially); hence t has fired at least k times in the firing sequence w reaching M. The tokens were not needed by any transition for reaching M; thus, one can delete the last k occurrences of t in w, getting a firing sequence reaching some marking M''. M'' has k more tokens on each place in ${}^\bullet t$ compared to M, so it enables t_1. By choice of k, firing t under M'' disables t_1, while firing t_1 empties some place in ${}^\bullet t$, i.e. it disables t. This is a contradiction to (a).

We continue with the proof gap for Lemma 4.3, which concerns Part (1). Here, a component C satisfying the modified condition (a) has to be considered – and for the claim about input/output conflicts in the theorem, the assumption that there are no dynamic input/output conflicts has to be considered, too. Further, there is a transition t to be contracted in C, an output transition t_2 with ${}^\bullet t_2 \cap {}^\bullet t \neq \emptyset$ (i.e. the contraction is of type 2) and a dummy transition t_1 with ${}^\bullet t_1 \cap t^\bullet \neq \emptyset$; for the claim about input/output conflicts, the case that t_1 is an input transition has to be considered, too. Assume that in C' (i.e. after the contraction) there is a dynamic conflict between t_2 and t_1 under reachable marking M'; a *contradiction* using (a) (freeness from input/output conflicts, resp.) will be derived and the fact that there is *no* dynamic conflict between t_2 and t in C (no dynamic conflict between t_2 and input transition t_1 resp.).

By Theorem 3.3 (1) of [VK07], there is a reachable marking M of C related to M' by the marking equality (restrict the simulation in 3.3 (1) to reachable markings). As

shown above, one knows that M enables t_2.

Let k be the maximal number of tokens that is missing on some $p \in {}^\bullet t \cap {}^\bullet t_1$ to enable t_1 under M. (In particular, if there is no place in the intersection, k is 0.) If k is positive, there must be k surplus tokens on all places in $t^\bullet$, and one can 'unfire' t k times as above; in this case, one now has a modified M, still satisfying the marking equality with M' and enabling t_2, such that for some $p \in {}^\bullet t \cap {}^\bullet t_1$, there are just enough tokens on p to enable t_1 under M.

If $k = 0$, it might be that all places in ${}^\bullet t$ have more tokens than needed to enable t_1 (this is true for some $p \notin {}^\bullet t_1$ if it has at least one token). Then t is fired until some $p \in {}^\bullet t$ has just enough tokens to enable t_1 under M (possibly, $p \notin {}^\bullet t_1$ is empty). Again, the modified M still satisfies the marking equality with M' and enables t_2.

Let $p_1 \in {}^\bullet t_1 \cap t^\bullet$; since (p, p_1) has enough tokens to enable t_1 under M', this means that p_1 has enough additional tokens that guarantee that t_1 is enabled under M as far as p_1 is concerned. Since this works for arbitrary $p_1 \in {}^\bullet t_1 \cap t^\bullet$, t_1 is enabled under M.

By (a) (freeness from input/output conflicts, resp.), any place not adjacent to t has enough tokens under M (or M') to enable t_1 and t_2 together. For any new place (p', p_1), p' also has enough tokens under M and the same holds for p_1 since M enables t_1 and $p_1 \notin {}^\bullet t_2$. Thus, (p', p_1) inherits enough tokens to satisfy independently the needs of t_2 and the combined needs of t_1, resulting from arcs from p' and p_1 to t_1. This is a contradiction to the assumption that there is a dynamic conflict between t_2 and t_1 under M'. □

4.2 Basic Idea

Now, the problem that the decomposition of [VW02, VK07] can lead to components with irreducible CSC conflicts will be tackled: considering Figure 4.2, the desired overall circuit OC modelled by the specification STG N – indicated in Figure 4.3(a) – can be implemented using two smaller component circuits CC for output y and DC for output x that are acting in the same overall environment. CC and DC result from the logic synthesis of the component STGs C_r and C_d partially shown in Figure 4.3(b) and (c), accompanied by parts of their state graphs; the clouds should be ignored for now. C_r and C_d are the outcomes of N's decomposition, which involved a transition contraction for each. As one can see, C_r has a CSC conflict between the highlighted states, since both have the same state vector $(a, y) = (0, 0)$ and in only one of them output y^+ is enabled.

This conflict describes a *(dynamic) self-trigger*, i.e. a firing sequence of two *input*

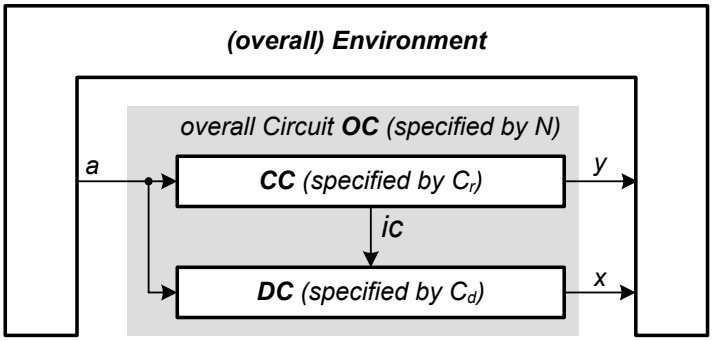


Figure 4.2: Circuit architecture of the resulting circuit.

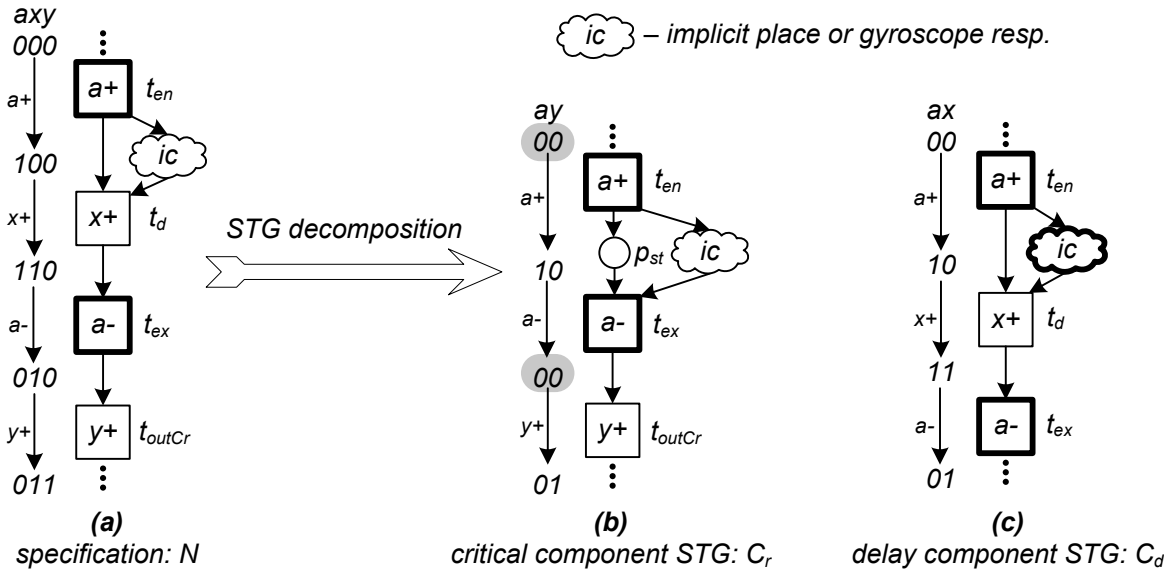


Figure 4.3: Self-trigger avoidance by internal communication. The decomposition of N was guided by the initial partition $\pi = \{(\{a, ...\}, \{x\}), (\{a, ...\}, \{y\}), ...\}$

transitions labelled with the same signal $a \in In_N$[3] and complementary edges returning to the same state vector.

The insertion of an *internal* signal transition ic^+ (at the position of the cloud) in order to solve this conflict is impossible, since it would not be input proper, cf. the discussion after Definition 3.2.3; hence, the conflict is *irreducible*. From the circuit's perspective, there is the risk that the environment can produce the second signal edge before the first one is completely registered by the implementing circuit. Thus, the second signal edge is called *critical edge* because it may arrive too fast causing malfunction of the circuit.

In this context, the component having the self-trigger is called *critical*. To find such self-triggers by considering the structure only, one can look for a *structural self-trigger* which consists of two transitions t and t' which are labelled with complementary edges of the same input signal and satisfy $t \in {}^\bullet({}^\bullet t')$; a structural self-trigger is called *MG-self-trigger* if the place between t and t' is an MG-place. A structural self-trigger is necessary for a dynamic one.

Figure 4.4 shows a 'real-life' specification that may arise in the context of control resynthesis, cf. Section 2.3.3. The control cluster at the top of Figure 4.4 is made up of a tree having a Sequence HC as root and two Parallel HCs (or Concur HCs resp.) as leafs.[4] The composition of the corresponding HC-STGs and the removal of internal handshakes, as shown above in Figure 2.11, yields the cluster STG in the middle of Figure 4.4. After total decomposition of this cluster STG each component STG has at least one self-trigger (see the grey ovals at the bottom of this figure), i.e. none of the component STGs is SI implementable.

A possible solution to avoid the self-trigger is to exploit a signal edge $x^\pm$ with $x \in Out_N$ between a^+ and a^- in N (cf. Figure 4.3): one can tune the decomposition not to contract the respective transition; instead, backtracking is performed and x is added as an *input* for C_r. Although this approach does help in some cases, in our experience an irreducible CSC conflict very often remains, in particular for the example in Figure 4.4.

Another possibility is to add x as an *output* to C_r; in this case, C_d will not be generated anymore and instead C_r is responsible for x and y and consequently larger. However, this very often yields a new irreducible conflict, which has to be resolved, too. Hence, repeated 'merging' of components may result in an uncontrollable growth of the resulting components, destroying the advantage of decomposition though. In Section 5.3

[3] In fact, one could also have $a \in Out_N$; this case can be treated in much the same way as described in the following – choosing $t_d = t_{ex}$ below.

[4] Therefore, one can call it Sequence-Parallel-Tree (SPT) with 2 levels – *SPT-2*.

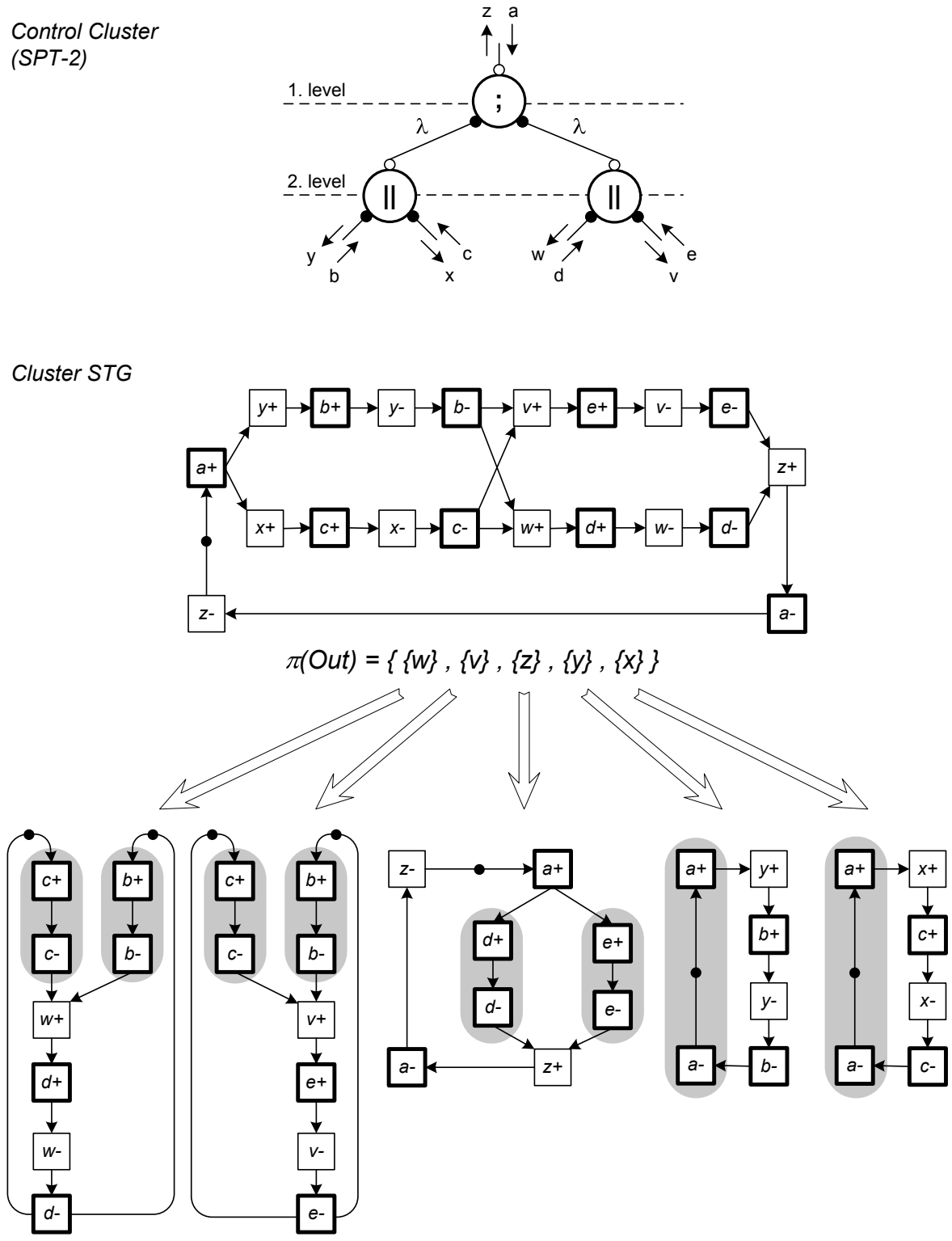


Figure 4.4: Control resynthesis example: after decomposition each component STG has at least one self-trigger.

this approach will be considered in more detail. For the example in Figure 4.4 this kind of self-trigger avoidance would eventually lead to no decomposition of the cluster STG, i.e. a decomposition according to $\pi(Out) = \{\{v, w, x, y, z\}\}$.

Now, we return to the idea to insert the internal signal ic between a^+ and a^-, see Figure 4.3 again.[5] The problem is that this requires the circuit environment to wait with lowering a for the occurrence of ic, although it does not even know about ic. The new idea is to achieve the desired causal relationship by making ic a communication signal between the components: an output between a^+ and a^- has to be identified in N, the so-called delay transition (here x^+), and the corresponding so-called *delay component* C_d that produces signal x; then, ic has to be inserted as an output in C_r and as an input in front of x^+ in C_d. This way, a^- waits for x^+, which in turn waits for ic. To view the situation from the point of C_r: its environment consists of all other components (including C_d) and the circuit environment, and this combined environment now indeed waits with lowering a for the occurrence of ic (i.e. C_d *delays* the overall environment such that a^- cannot arrive too fast causing malfunction of the circuit).

Finally, ic will be hidden (cf. set H in Definition 3.2.3) such that, from the point of view of N, one has inserted internal signal ic at the position of the cloud, and this is correct. This view is the key to prove the new method correct:

1. Insert ic into N preserving correctness; since only nets without internal signals can be decomposed, one can regard the result as $N'/\{ic\}$, where ic is an output of N'.

2. N' will be decomposed with the same initial partition as before, except that ic is added to the outputs of C_r and to the inputs of C_d. And indeed, in the reduction of C_r, x^+ is contracted putting ic into the right position. (Observe that the path between the a^+- and the a^--labelled transitions in N could be longer. In this case, a might not be a signal of C_d, but the former arguments are also valid in such a case.)

Furthermore, it should be possible to avoid recomputation of the other components: for such a component, ic is lambdarised initially, so it should be possible to remove it immediately, since it can even be removed when representing an internal signal. Then, the reduction can proceed as in the case of N, leading to the same result as before.

An important issue to apply this approach is to identify a delay transition in N (i.e. an output of the delay component), so it will be defined more formally now: Consider

[5] For the time being, one can ignore the fact that one really has to insert a signal *edge* $ic^{\pm}$.

Input	specification N — critical component C_r with structural self-trigger t_{en}, t_{ex} delay component C_d with t_d 'between' t_{en} and t_{ex} in N, $l(t_d) \in Out_d^{\pm}$
Output	modified specification N' — modified critical component C_r' — modified delay component C_d'

1 in N, perform an output gyroscope insertion for a fresh output signal ic via an implicit place p and intermediate N'' such that in N'', $t_{en} \in {}^\bullet p$, $t_d \in p^\bullet$, $l(p^\bullet) \subseteq Out_d^{\pm}$ and $l({}^\bullet p) \subseteq Sig_r^{\pm}$; this gives N'
2 C_r' is obtained from reduction of N' with partition member $(In_r, Out_r \cup \{ic\})$
3 C_d' is obtained from reduction of N' with partition member $(In_d \cup \{ic\}, Out_d)$

Figure 4.5: Algorithm AVOID-0 for inserting internal communication between two components.

a self-trigger of C_r between the markings M_1 and M_2, i.e. there is a transition sequence $M_1[t_{en}t_{ex}\rangle M_2$ and (t_{en}, t_{ex}) is the *entry/exit transition pair* of the self-trigger. A place p_{st} with $t_{en} \in {}^\bullet p_{st}$ and $t_{ex} \in p_{st}{}^\bullet$ is called *self-trigger place*, see also Figure 4.3(b).

Definition 4.2.1 (Delay Transition)
A transition t_d of N is called *delay transition* w.r.t. a transition pair (t_{en}, t_{ex}) if $t_d \neq t_{en}$ is labelled with a local (i.e. non-input) signal edge and t_d occurs in *all* transition sequences t_{en} ... t_{ex} enabled under a reachable marking of N. △

Intuitively, if t_{ex} fires after t_{en}, then t_d must fire in between. Thus, by delaying t_d the firing of t_{ex} will also be delayed, i.e. the critical edge – and this avoids the self-trigger.

For an implementation of the presented idea, one problem remains to be solved. To ensure the practically important consistency, one cannot only insert ic^+ between a^+ and a^-, but one has to introduce a suitable ic^--transition somewhere else, too. This is problematic since it is unclear how to find a proper insertion point such that the behaviour is preserved; the usual methods for event insertion would build the reachability graph of N, which must be avoided.[6] Instead, the cloud is replaced by a gyroscope (simulating a toggle transition), cf. Section 4.1.1. Another possibility is to replace it with a 4-phase handshake between the critical and the delay component; this approach will be investigated in future work, see Section 4.6.

The above considerations are formalised in the algorithm AVOID-0 in Figure 4.5. It is correct according to the following theorem.

[6] Unfolding based methods are more competitive but also not suited for very large specifications.

Theorem 4.2.2
The algorithm AVOID-0 *is correct: if* $(C_i)_{i\in I}$ *is correct w.r.t.* N*, then* $((C_i)_{i\in I'}, C'_r, C'_d)$ *with* $I' = I \setminus \{r, d\}$ *is correct w.r.t.* N'*, which is correct w.r.t.* N *when hiding* $\{ic\}$*.*

Proof. Proposition 4.1.8(3) implies that $N'/\{ic\}$ is correct w.r.t. N, or equivalently, that N' is correct w.r.t. N when hiding $\{ic\}$. In the following, it will be shown that the components $((C_i)_{i\in I'}, C'_r, C'_d)$ can be constructed from N' by the decomposition algorithm for some feasible initial partition. Then it is guaranteed that $((C_i)_{i\in I'}, C'_r, C'_d)$ is correct w.r.t. N'; cf. the discussion before Proposition 3.2.5.

Let the gyroscope be defined as in Definition 4.1.6. First, the new feasible partition is the original one exchanging the members for r and d as in Figure 4.5. This is feasible since

- the new output transitions g_1 and g_2 are in structural conflict only with each other and they carry the same signal
- g_1 and g_2 are only triggered by transitions with label in $In_r = In'_r$ or by each other; they only trigger transitions with label in $Out_d = Out'_d$ (and each other) and $ic \in In'_d$.

Now, consider a component C_j with $j \in I'$ and the corresponding initial component N'_j derived from N'. In N'_j, ic is lambdarised, which is feasible since ic only triggers outputs of C_d; hence $l'_j(g_1) = l'_j(g_2) = \lambda$. Then, Proposition 4.1.7 allows to apply subnet contraction to the inserted gyroscope yielding the STG N''_j. Clearly, in N''_j, the resulting place is implicit by definition of gyroscope insertion and can be deleted resulting in the original initial component N_j, which can be reduced to the final component C_j with the same sequence of operations applied to N_j originally. □

Observe that the resulting decomposition is correct in the sense of Definition 3.2.3, but it is not even guaranteed that the structural self-trigger actually disappears, but often the method is successful, cf. Section 4.5.

Of course, there might be several structural self-triggers and one has to apply AVOID-0 a number of times; this procedure results in a series of decompositions $(C^j_i)_{i\in I}$ and modified specifications N_j, $j = 1, \ldots, k$ as follows.

From the given decomposition $(C_i)_{i\in I}$, which is correct w.r.t. N, one gets $(C^1_i)_{i\in I}$ and N_1 from AVOID-0 such that $(C^1_i)_{i\in I}$ is correct w.r.t. N_1 and N_1 is correct w.r.t. N when hiding $\{ic_1\}$ by Theorem 4.2.2. Then AVOID-0 can be applied to $(C^1_i)_{i\in I}$ and N_1 and so on.

Generally, when this results in $(C_i^j)_{i\in I}$ and N_j, then in the next stage AVOID-0 produces $(C_i^{j+1})_{i\in I}$ and N_{j+1} such that the former is correct w.r.t. to the latter, which in turn is correct w.r.t. N_j when hiding $\{ic_{j+1}\}$. If one gets $(C_i^k)_{i\in I}$ and N_k after the last stage, then Theorem 3.2.4 shows that $(C_i^k)_{i\in I}$ is correct w.r.t. N_{k-1} when hiding $\{ic_k\}$.

One can repeat this argument: generally, if one has already concluded that $(C_i^k)_{i\in I}$ is correct w.r.t. N_{j+1} when hiding $\{ic_k, ic_{k-1}, \ldots, ic_{j+2}\}$, then it can be seen from Theorem 3.2.4 and the fact that N_{j+1} is correct w.r.t. N_j when hiding $\{ic_{j+1}\}$ that $(C_i^k)_{i\in I}$ is correct w.r.t. N_j when hiding $\{ic_k, \ldots, ic_{j+1}\}$. Finally, it can be seen that $(C_i^k)_{i\in I}$ is correct w.r.t. N when hiding $\{ic_1, \ldots, ic_k\}$.

One can improve the efficiency as follows: if there are several irreducible CSC conflicts concerning different components, step 1 of AVOID-0 is executed iteratively for each of the conflicts using the respective modified specification, and afterwards all affected components are newly generated in a single second reduction pass.

The algorithm AVOID-0 has to recalculate the affected components; instead, in order to improve efficiency, one could apply gyroscope insertions directly to these components. Section 4.4 presents an algorithmic solution for a special case of specifications where this is possible.

Observe that AVOID-0 is restricted to specifications where the self-triggers of the components can be avoided by using just one delay component which has to produce all output signals specified by the transition labels in $p^\bullet$. Such a delay component might get too complex for logic synthesis if it has to produce so many outputs. To overcome such an uncontrollable growth of the delay component a generalisation of AVOID-0 will be presented in Section 4.3.3. However, AVOID-0 produces efficient results for the practical important case if p is a shortcut place, i.e. there is an MG-path from t_{en} to t_d.

4.2.1 Beyond Self-Triggering

In the remainder of this chapter, self-trigger avoidance will only be considered. This is not a real restriction since on the one hand self-triggers appear very often in the benchmark examples, see Section 4.5. On the other, self-triggers are the most severe types of irreducible CSC conflicts – w.r.t. to the presented approach – since for each self-trigger only one transition pair (t_{en}, t_{ex}) can be identified for which a delay transition must be found. Now, it can be proposed how to deal with general irreducible CSC conflicts by identifying several transition pairs that yield several opportunities to determine a delay transition.

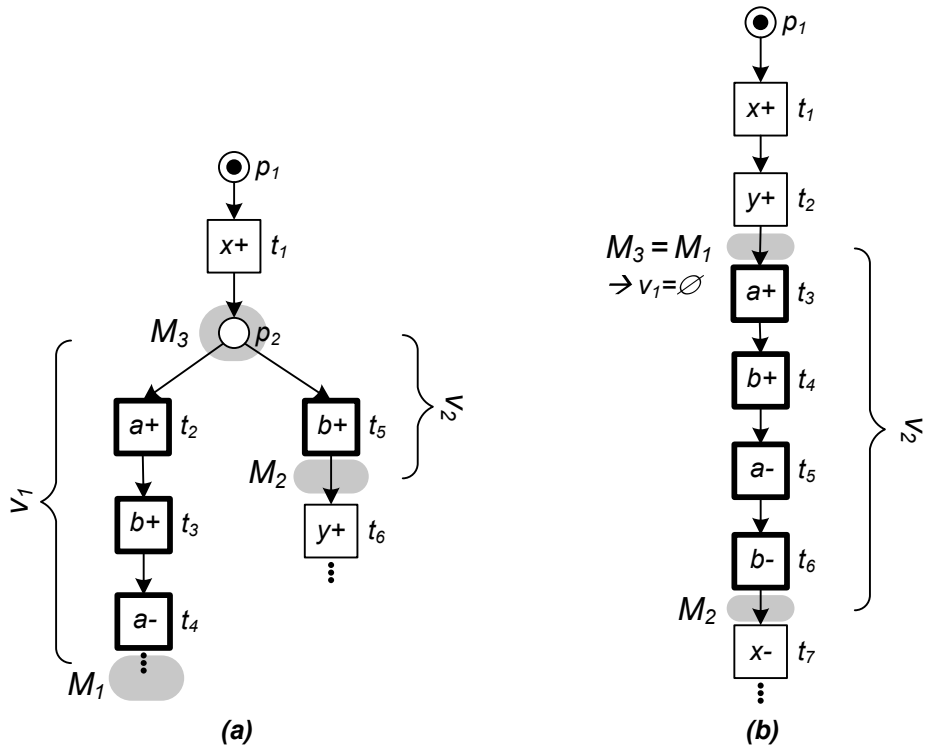


Figure 4.6: Critical components having irreducible CSC conflicts between M_1 and M_2: (a) a type II conflict in terms of [KMY06] and (b) a conflict of type I.

Consider the two critical components in Figure 4.6 having an irreducible CSC conflict between M_1 and M_2 each. Such conflicts can be characterised by a third marking M_3 and two traces $M_3[w_1\rangle\rangle M_1$ and $M_3[w_2\rangle\rangle M_2$, where w_1 and w_2 consist of input events only. In principle MPSAT and PETRIFY are able to report these traces, and they can be mapped to corresponding firing sequences v_1 and v_2; observe that v_1 could even be empty, as in Figure 4.6(b). Following [KMY06] it is enough to consider conflict types as in Figure 4.6, since other CSC conflicts can be reduced to these types of conflicts; Figure 4.6(a) refers to a typical type II conflict (according to [KMY06]) and (b) to a type I conflict.

Figure 4.6 also demonstrates that, for general irreducible CSC conflicts, there are usually several pairs (t_a, t_b) – like an entry/exit transition pair – where one can look for a delay transition that fires between both transitions. For example in Figure 4.6(a) one can identify the pairs $(t_2, t_3), (t_3, t_4)$ and in Figure 4.6(b) $(t_3, t_4), (t_5, t_6), (t_3, t_5)$ etc. Irreducible CSC conflicts can be avoided if our approach succeeds for just one of those transition pairs. Hence, the focus on self-triggers (which exhibit only one possibility for identifying an entry/exit transition pair) is not a real restriction.

4.3 Algorithmic Solution and Generalisations

In this section, several open questions are tackled in order to develop an algorithmic solution to avoid irreducible CSC conflicts for unweighted STGs (i.e. all arcs must have the weight 1, apart from that there are no further structural restrictions).

An open question for applying the algorithm AVOID-0 (see Figure 4.5) for more general structured specifications is to find a delay transition for a given self-trigger by exploring the (usually large) specification in an efficient way. A solution to this problem will be presented in Section 4.3.1.

Another problem is how to insert an implicit place q between t_{en} and t_d (as a necessary step for the internal gyroscope insertion) if there is no MG-path between these transitions, see e.g. Figure 4.7 and also Figure 4.16(b) below (when interpreting the cloud symbol as q). Note that q is not a shortcut place there and hence one has to determine the entire sets ${}^\bullet q$ and $q^\bullet$ in an efficient way. In Section 4.3.2 a *structural* technique will be presented to insert an implicit place into a Petri net.

Finally, in Section 4.3.3 an improvement of AVOID-0 from Figure 4.5 will be proposed in order to avoid uncontrollable growth of the delay component (i.e. preventing the delay component from generating unnecessary output signals) by introducing internal communication between the critical component and several auxiliary components. This

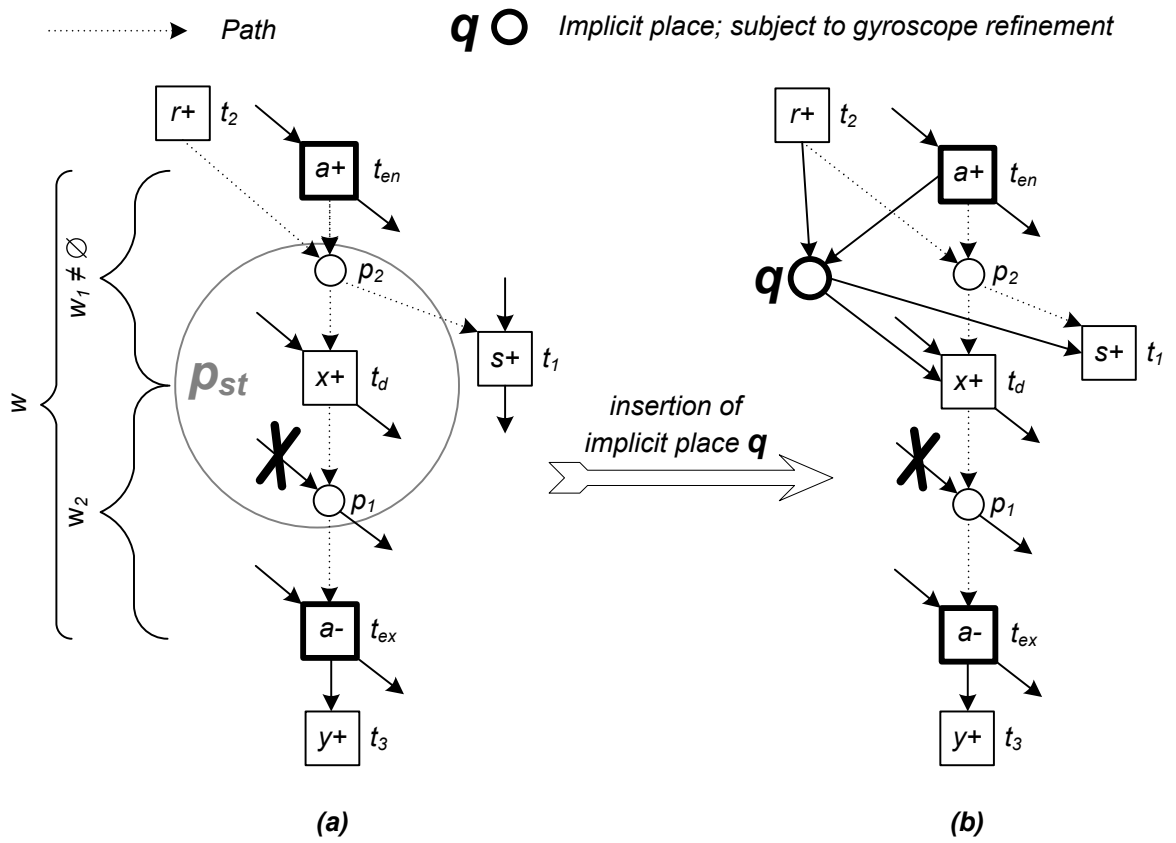


Figure 4.7: Specifications N yielding at least one delay candidate t_d for a potential self-trigger, given by t_{en} and t_{ex}; p_{st} sketches a self-trigger place of C_r as a result of reducing all elements in w, cf. also Figure 4.3(b).

is also a solution to the open problem of having several delay components which will be discussed in Section 4.6.

4.3.1 How to Identify a Delay Transition

A delay transition is defined based on traces of the specification N (see Definition 4.2.1), but due to the complexity of N a structural condition is needed to identify such a transition. Since it is difficult to find a necessary condition that is not too restrictive or a real sufficient condition (that is *only* satisfied for delay transitions), an indication for a delay transition is proposed – which is called a delay candidate:

Definition 4.3.1 (Delay Candidate)
For a specification N and an entry/exit transition pair (t_{en}, t_{ex}), a *delay candidate* t_d is a local transition on a path $w = w_1 w_2$ starting from t_{en} leading to t_{ex}, where w_1 is a path from t_{en} to t_d ($t_{en} \neq t_d$), while w_2 is a path from t_d to t_{ex} without merge places, i.e. $\forall p \in w_2 \, |{}^\bullet p| = 1$. No transition on $w \setminus \{t_{en}, t_{ex}\}$ has its signal in Sig_r. △

Observe that the path w_2 could even be empty; in this case there are no restrictions on the specification's structure. The non-merging path w_2 indicates that t_d has to fire before t_{ex}, except that the initial marking of w_2 might initially allow to fire t_{ex} without firing t_d. The last condition reflects the idea that w is contracted to p_{st} in C_r, cf. Figure 4.7(a), i.e. none of its transitions except t_{en} and t_{ex} is *r-relevant*; the signals of C_r are called r-relevant, in particular when considered in the context of the full STG N.

Although a delay candidate is not necessarily a delay transition according to Definition 4.2.1, our criterion is sufficient for many benchmark examples – see our examples in Section 4.5.

Algorithmic Idea

Instead of simply searching for all paths w (from t_{en} to t_{ex}) containing local signal transitions (as possible delay candidates), the application of a more efficient technique is suggested:

First, one can ignore all transitions labelled with an *r-relevant* signal edge except t_{en} and t_{ex}; they can be deleted as well as all arcs to t_{en} in N resulting in N', since for delay candidates one is only interested in local transitions which must be located within the circle p_{st}, cf. Figure 4.3(b) as well as 4.7(a).[7]

Second, one can apply an ordinary depth-first search (DFS) [CLRS01] starting at t_{en} in order to determine all transitions in N' that are reachable from t_{en}; the others are

[7] The arcs from t_{ex} do not matter for this subsection.

removed as well. One could also perform a breadth-first search and store the distances from t_{en}; this would help to find a delay candidate close to t_{en}, see below.

Third, in N' a *backward* search starting at t_{ex} will be applied to find all paths w_2 from a delay candidate t_d to t_{ex}; the search can be restricted, because only non-merging places p should be visited (i.e. $|{}^\bullet p| = 1$). To find all such paths, one can modify the backward-directed DFS [CLRS01]: when going backward, vertices are marked as visited as usual (this avoids repeated vertices on a path), but when returning from the recursion, the mark is erased (and the vertex can be used in other paths). One can also consider just one or a few paths, hoping that the respective candidates will help to avoid the self-trigger.

After a potential path w_2 is found, one can apply a backward-directed breadth-first search (BFS) to find a shortest path w_1 from t_{en}. For this, all the vertices of w_2 must be marked as visited, initially. Only if t_{en} is reachable, t_d is a delay candidate. The idea is that the algorithm in Section 4.3.2 starts at t_{en} and tries to reach some t_d quickly; hence, t_d should be chosen such that w_1 is short.

Usually, there are several possible paths w_2 and so several possible delay candidates. It is subject to future work to investigate how the delay candidate selection influences the synthesis time and the resulting circuit area and performance.

This technique is purely structural (and hence really fast), but might return delay candidates which are not delay transition in terms of Definition 4.2.1. To confirm the latter, one could apply for each delay candidate t_d (integer) linear programming techniques which will be presented later in Section 5.3 in order to approximate a reachability condition for t_d being a 'real' delay transition.

4.3.2 Inserting Implicit Places into a Petri Net

For a given entry transition t_{en} and a delay candidate t_d (with a path w_1 from t_{en}) the algorithm in Figure 4.8 inserts a redundant place q (w.r.t. some set Q) into an unweighted STG N such that $t_{en} \in {}^\bullet q$ and $t_d \in q^\bullet$, cf. Figure 4.7.

The insertion of q will be initiated by calling the function insertImplicitPlace. In line 8, a *forward traversal* in N from t_{en} towards t_d will be started by calling the place function with the unique place p as argument such that $p \in w_1$ and $p \in t_{en}{}^\bullet$. Every time the function place is processed as well as in lines 27 or 31, a redundant place q could be inserted via the operations specified in lines 9 and 10. The redundancy of q is assured, since for every call of place and every new place p of Q, ${}^\bullet p$ as well as $p^\bullet$ will be added to the potential preset (preq) and postset of q (postq) – except for loop

```
   // Global variables
1  multiset of transitions  preq, postq   // potential pre- and postset of q
2  set of places            Q
3  set of places            w_1           // shortest path from t_en to t_d
4  boolean                  delayFound
   // Main function
5  bool insertImplicitPlace()
6     preq ← ∅ ; postq ← ∅ ; Q ← ∅ ; delayFound ← false
7     p ← w_1 ∩ t_en•
8     if (place(p, t_en))
9        P' ← P ∪ {q} ; M_N'(q) ← M_N(Q)
10       for all t ∈ T do W'(t, q) ← preq(t) ; W'(q, t) ← postq(t)
11       return true
12    else return false
13 bool transitionF(transition t)          bool transitionB(transition t)
14    (stop, value) ← checkTransitionF(t)     (stop, value) ← checkTransitionB(t)
15    if (stop) return value                  ...
16    if (delayFound = false)
17       p ← w_1 ∩ t•
18       return place(p, t)
19    for all p ∈ t• \ Q do                   for all p ∈ •t \ Q do
20       if (place(p, t)) return true            ...
21    return false                            ...
22 bool place(place p, transition t)   // p, t adjacent
23    if M_N(Q) + M_N(p) > 1 return false
24    Q ← Q ∪ {p}
25    preq ← preq + •p ; postq ← postq + p•
26    (preq, postq) ← (preq − postq, postq − preq)    // removes loops
27    res ← true
28    for all t' ∈ p• − {t} ∧ postq(t') > preq(t') do res ← res ∧ transitionF(t')
29    for all t' ∈ •p − {t} ∧ postq(t') < preq(t') do res ← res ∧ transitionB(t')
30    if (¬res) restore old values of in, out and Q  // before present call of place()
31    return res
```

Figure 4.8: Algorithm InsertImplicit for inserting an implicit place q. It is assumed that t_{en}, t_{ex} and t_d are known to the algorithm as global constants. '...' means line repetition from transitionF.

```
1   (bool,bool) checkTransitionF(transition t)
2     if (t = t_en) return (true,false)
3     if (delayFound = false)
4       if (t = t_d) delayFound ← true
5         return (true, true)              // successful termination
6       if (t ∈ w_1) return (false,false)
7     if (l(t) ∈ Loc_N^±) return (true,true)  // successful termination
8     if (t = t_ex) return (true,false)
9     return (false,false)
10  (bool,bool) checkTransitionB(transition t)
11    if (t = t_ex) return (true,false)
12    if (l(t) ∈ Sig_r^±) return (true,true)  // successful termination
13    return (false,false)
```

Figure 4.9: Strategy CheckTransition. These functions know all global variables of InsertImplicit and check validity of transitions for the preset (checkTransitionB) and the postset (checkTransitionF) of q – with result $(stop, value)$. If $stop$ = false, the traversal should be continued and t is not valid ($value$ = false). Otherwise, $value$ says whether t is valid (successful termination) or not; in the latter case, place has to backtrack and to remove its current p from Q.

transitions – see lines 25 and 26 (for a formal proof, see below). Consequently, for every incoming token to p a token is added to q, for every token removed from p a token is removed from q, too. If the redundant place q were unsafe, then one would get a CSC conflict due to firing at least two edges of the new internal signal ic; thus p is not added if it would lead to at least two tokens on q initially, see line 23.

To make q useful, the preset and postset of q must fulfil some application specific requirements which are tested by calling transitionF for each transition in $p^\bullet$ (line 28) and transitionB for each transition in ${}^\bullet p$ (line 29) – F and B resp. indicate a *forward* or *backward* traversal. The functions checkTransition[F/B] in line 14 are used to terminate the traversal at certain transitions, see Figure 4.9. They are tuned to make $q^\bullet$ consist of local transitions only – including t_d – such that the later gyroscope refinement of q with a new internal signal ic is input proper according to [SV07], which means that no input will be delayed by ic. Furthermore, ${}^\bullet q$ should consist of t_{en} and other transitions labelled with r-relevant edges only, such that the recalculated critical component C'_r gets no additional relevant signals, except for ic as an additional output; this avoids uncontrollable growth of C'_r.

If checkTransition[F/B] does not return to stop the search, then transition[F/B] calls the

place function again to find a suitable place in the postset or preset resp. of t. With such a call, the post- and preset of q are extended until these fulfil the requirements specified in checkTransitionF and checkTransitionB resp. Note that the algorithm is correct for arbitrary functions checkTransition[F/B], see Proposition 4.3.2 below.

To ensure that $t_d \in q^\bullet$, the algorithm applies a forward traversal straight to t_d as long as the delayFound flag is not set, see lines 16 – 18. Similarly, in the implementation of place, the next transition t' on w_1 is chosen first in line 28 (not shown). Eventually, the flag will be set by checkTransitionF, and then all the other possible paths starting from the places in w_1 will be traversed in forward and backward direction.

The algorithm works on a net N and constructs a copy N' extended with q in lines 9 and 10.

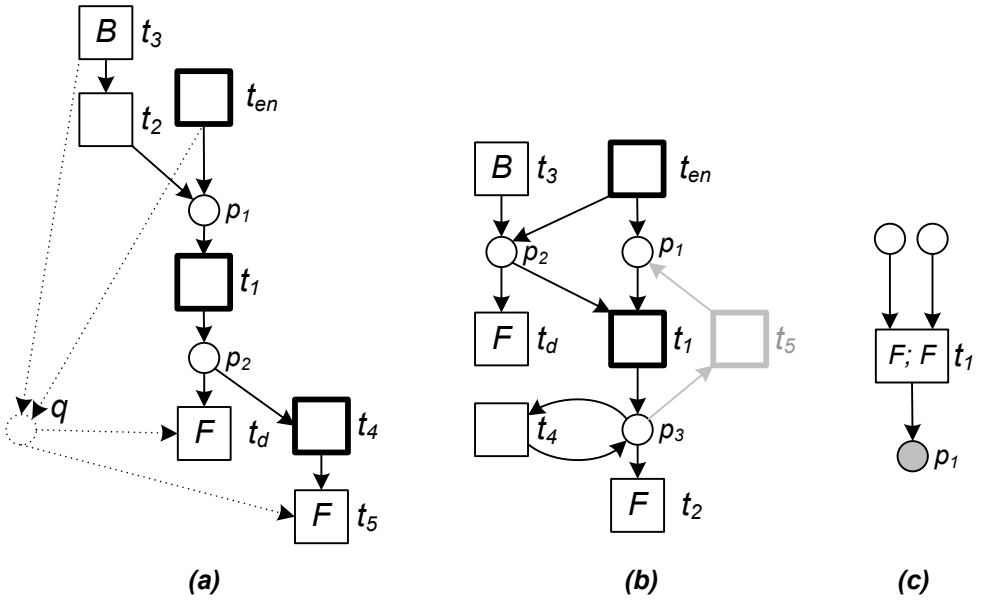


Figure 4.10: Examples for InsertImplicit: the loops in lines 28 and 29 process the resp. nodes from left to right; at transitions labelled with F the forward search terminates successfully, and analogously for B.

It might be clearer how InsertImplicit works when investigating the examples shown in Figure 4.10. For the example in (a) by ignoring the dashed arcs, w_1 is $t_{en}p_1t_1p_2t_d$; thus, the algorithm calls place(p_1, t_{en}) (e.g. t_1 is added to postq), transitionF(t_1), place(p_2, t_1) (e.g. t_1 is removed from postq) and transitionF(t_d). Now flag delayFound is set and the last call returns successfully. Next, transitionF(t_4) does not directly terminate since t_4 is an input transition, but the next call of transitionF returns successfully as well. Going back to the call place(p_1, t_{en}), transitionB(t_2) (no termination

since $l(t_2) \notin Sig_r^{\pm}$) and then transitionB($t_3$) are performed. Finally a redundant place q with ${}^\bullet q = \{t_{en}, t_3\}$ and $q^\bullet = \{t_d, t_5\}$ is inserted, cf. the dashed components in (a).

Next consider Figure 4.10(b), first without transition t_5. The forward traversal reaches t_d, and then transitionF(t_1) and place(p_3, t_1) are called. Here, t_4 is added to preq and postq in line 25, and removed again in line 26. Thus, for t_4 neither transitionF nor transitionB are called in lines 28 or 29. Eventually, the algorithm inserts a redundant place q, where ${}^\bullet q = \{t_{en}, t_3\}$ and $q^\bullet = \{t_d, t_2\}$.

Now consider transition t_5; when reaching p_3, the following calls are transitionF(t_5), place(p_1, t_5) and transitionF(t_1). Here, the check requires to continue (t_1 is an input transition), but the algorithm fails in line 19, since there is no place left in ${t_1}^\bullet \setminus Q$ – i.e. no redundant place q can be inserted. Of course this failure is not very welcome, but the repeated traversal of places in Q – like p_3 – would prevent the termination of the algorithm.

Figure 4.10(c) indicates a situation, where t_1 has already been visited by forward traversals – and declared as suitable for $q^\bullet$ via line 7 of checkTransitionF – for two times. Now place p_1 is reached via a call place(p_1, t), and t_1 is added to *preq* in line 25; this would make t_1 a (generalised) loop transition for q. But after line 26, $preq(t_1) = 0$ and $postq(t_1) = 1$, and due to the condition in line 29, transitionB will not be called for t_1; the search terminates by returning true.

Proposition 4.3.2
For any terminating checkTransition{F/B} operation without side-effects on N, Q, preq or postq, the algorithm InsertImplicit *terminates such that the place q is implicit in N'. If* false *is returned, no place q will be inserted.*

Proof. First observe the following properties: ($*$) for every call of place(p, t), there is $p \notin Q$; this follows immediately from lines 7 and 8, 19 and 20 resp., where these are the only places from which place(p, t) is called.

($**$) The number of calls of place on the call-stack is bounded by $|P|$; which directly follows from ($*$).

Termination: Obviously, insertImplicitPlace() is only called at the beginning, and transition{F/B}() and place() are calling each other alternately in the sense that they alternate on the call-stack. Together with ($**$), this implies that the call-stack has bounded depth. Since only **for** loops are used, only finitely many functions are called in each function call f(), and eventually a function is called or f() returns. This implies the claim.

Correctness: Now, it will be proven by induction that the properties of Definition 3.2.2

are fulfilled every time a function is called or returned from. More precisely, it will be shown that q would be redundant for the valuation $V \equiv 1$ if N would be modified as in lines 9 and 10. Hence, one only has to consider the values of *preq*, *postq* and Q.

Clearly, q is redundant for $V \equiv 1$ after the initialisation of the global variables in line 6: q is an unconnected place then, and $d = 0$. Furthermore, one only has to consider the lines 24–26, since only here *preq*, *postq* and Q will change significantly. If they are restored (to the values before line 24) in line 30, q is redundant by induction assumption. The new values (i.e. after execution of line 26) will be denoted with *preq*′, *postq*′, W' etc. Observe that $M_{N'}(p) = M_N(p)$ for every place $p \neq q$ and $W'(x, y) = W(x, y)$ if $x \neq q \neq y$.

(1)

$$\begin{aligned} & d' \\ = \;& M_{N'}(q) - \sum_{s \in Q'} M_{N'}(s) \\ (\text{definition of } M_{N'}(q)) \;=\; & M_{N'}(Q') - \sum_{s \in Q'} M_{N'}(s) = 0 = d \end{aligned}$$

(2)

$$\begin{aligned} & 0 \\ \leq \;& W'(t,q) - W'(q,t) - \sum_{s \in Q'} (W'(t,s) - W'(s,t)) \\ (\text{definition of } \mathit{preq} \text{ and } \mathit{postq}) \;=\; & \mathit{preq}'(t) - \mathit{postq}'(t) - \sum_{s \in Q'} (W(t,s) - W(s,t)) \\ (\text{lines 24–26 and } (*)) \;=\; & ((\mathit{preq} + {}^\bullet p) - (\mathit{postq} + p^\bullet))(t) \\ & -((\mathit{postq} + p^\bullet) - (\mathit{preq} + {}^\bullet p))(t) \\ & -(W(t,p) - W(p,t)) - \sum_{s \in Q} (W(t,s) - W(s,t)) \\ (\text{Definition 3.0.1}) \;=\; & \max(0, \underbrace{\mathit{preq}(t) + {}^\bullet p(t) - \mathit{postq}(t) - p^\bullet(t)}_{a}) \\ & - \max(0, \underbrace{\mathit{postq}(t) + p^\bullet(t) - \mathit{preq}(t) - {}^\bullet p(t)}_{-a}) \\ & -W(t,p) + W(p,t) - \sum_{s \in Q} (W(t,s) - W(s,t)) \end{aligned}$$

$$
\begin{aligned}
(\max(0,a) - \max(0,-a) = a) &= preq(t) + {}^\bullet p(t) - postq(t) - p^\bullet(t) \\
&\quad -W(t,p) + W(p,t) - \sum_{s\in Q}(W(t,s) - W(s,t)) \\
(\text{definition of } preq \text{ and } postq, &= W(t,q) - W(q,t) + W(t,p) - W(p,t) \\
\text{e.g. } {}^\bullet p(t) = W(t,p) \le 1) &\quad -W(t,p) + W(p,t) - \sum_{s\in Q}(W(t,s) - W(s,t)) \\
&= W(t,q) - W(q,t) - \sum_{s\in Q}(W(t,s) - W(s,t)) \\
(\text{induction}) &\ge 0
\end{aligned}
$$

(3)

$$
\begin{aligned}
& d' \\
&\ge W'(q,t) - \sum_{s\in Q'} W'(s,t) \\
(\text{definition of } postq) &= postq'(t) - \sum_{s\in Q'} W(s,t) \\
(\text{lines 24–26 and } (*)) &= ((postq + p^\bullet) - (preq + {}^\bullet p))(t) \\
&\quad -W(p,t) - \sum_{s\in Q} W(s,t) \\
(\text{Definition 3.0.1}) &= \max(0, \underbrace{postq(t) + p^\bullet(t) - preq(t) - {}^\bullet p(t)}_{a}) \\
&\quad -W(p,t) - \sum_{s\in Q} W(s,t)
\end{aligned}
$$

1. Case: $a \le 0$

$$
\begin{aligned}
\dots &= 0 - W(p,t) - \sum_{s\in Q} W(s,t) \\
&\le W(q,t) - \sum_{s\in Q} W(s,t) \\
(\text{induction and } (1)) &\le d = d'
\end{aligned}
$$

2. Case: $a > 0$

$$
\begin{aligned}
\ldots &= postq(t) + p^\bullet(t) - preq(t) - {}^\bullet p(t) \\
&\quad -W(p,t) - \sum_{s \in Q} W(s,t) \\
\text{(definition of } postq, &= W(q,t) + W(p,t) - preq(t) - {}^\bullet p(t) \\
p^\bullet(t) = W(p,t) \leq 1) &\quad -W(p,t) - \sum_{s \in Q} W(s,t) \\
&= W(q,t) - preq(t) - {}^\bullet p(t) - \sum_{s \in Q} W(s,t) \\
&\leq W(q,t) - \sum_{s \in Q} W(s,t) \\
\text{(induction and (1))} &\leq d = d'
\end{aligned}
$$

□

4.3.3 Self-Trigger Avoidance for Unweighted STGs

We can now take a closer look at the internal gyroscope refinement via the implicit place q in N, which introduces internal communication to avoid the self-trigger in C_r.

If one applies the implicit place insertion of q in N as proposed in Section 4.3.2, then also after the gyroscope refinement of q, there are structural conflicts between all (local) transitions of $q^\bullet$. These structural conflicts imply that one has to change the feasible partition for the second reduction pass. Since the delay component produces the signal of t_d, it now has to produce all the signals of the transitions in $q^\bullet$, also those that were produced by other so-called *auxiliary components* in the first pass. Thus, the modified delay component STG could be very complex such that a circuit can perhaps not be synthesised anymore. This phenomenon will be visualised with the help of Figures 4.7(b) and 4.11:

Assume the unmodified delay component only produces output signal x, another component produces signal s, and the delay transition t_d for the self-trigger between t_{en} and t_{ex} is identified. An internal gyroscope with a new signal ic is inserted via the implicit place q. Observe that $q^\bullet$ does not only contain t_d, but also t_1; thus t_d and t_1 are in structural conflict because of q – and after q's refinement the conflict is caused by the gyroscope place p_{out} (see also Figure 4.1). Using feasible partitions only, the component for x now has to produce s as well; one gets a new component combining the delay component C_d and the component generating signal s. This can lead to

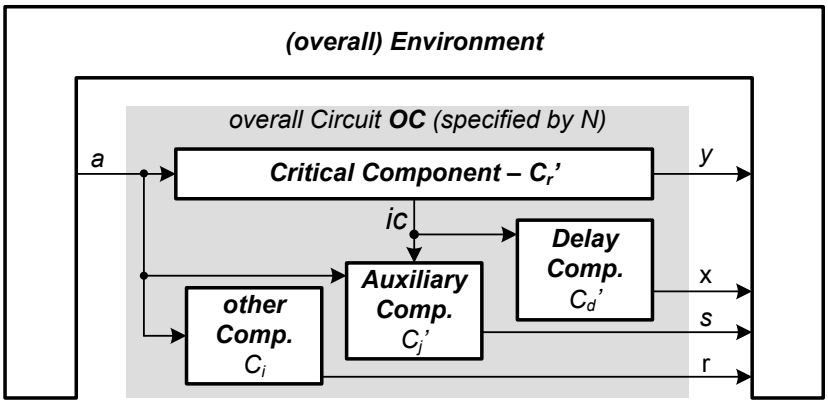


Figure 4.11: Architecture of the resulting decomposed circuit specified by Figure 4.7(a).

large delay component STGs from which no circuit can be synthesised (because of their complexity).

Thus, for the second reduction pass, the concept of quasi-feasible partition as studied in Subsection 4.1.2 can be applied. After an output gyroscope insertion introducing ic via implicit place q, the original feasible partition can be kept, except for adding ic as an output for C_r and as an input for all components producing a signal of a transition in the postset of q; these components include C_d, the others are called auxiliary components; cf. algorithm AVOID-1 in Figure 4.12 that generalises AVOID-0 from Figure 4.5 since it deals with several auxiliary components, in addition to the delay component.

This algorithm does not necessarily produce a correct decomposition, but a failure can be recognised:

Proposition 4.3.3

The algorithm AVOID-1 *is correct in the following sense: if $(C_i)_{i\in I}$ was obtained from N by decomposition (and hence correct w.r.t. N), then either the partition used by* AVOID-1 *is quasi-feasible and $((C_i)_{i\in I'}, C'_r, C'_d, (C'_j)_{j\in J})$ with $I' = I \setminus (\{r, d\} \cup J)$ is correct w.r.t. N', which in turn is correct w.r.t. N when hiding $\{ic\}$, or there is some dynamic self-trigger with the two transitions of ic in C'_d.*

Proof. When decomposing N, a feasible partition was used (or at least a quasi-feasible one, if already some internal signal has been introduced). If the partition used in AVOID-1 is not quasi-feasible, then there are two output transitions t_1 and t_2 in N' which are in conflict under some reachable marking and have signals produced by different components. Since this conflict is obviously new, t_1 and t_2 must have been in the postset of q in the intermediate N''. If they are in conflict under some reachable marking M

Input	specification N — critical component C_r with an irr. CSC conflict t_{en}, t_{ex} delay component C_d with t_d 'between' t_{en} and t_{ex} in N, $l(t_d) \in Out_d^{\pm}$ — other components $(C_i)_{i \in I \setminus \{r,d\}}$
Output	modified specification N' — modified critical component C'_r — modified delay component C'_d — modified auxiliary components $(C'_j)_{j \in J},\ J \subset I$

1 in N, if insertImplicitPlace() returns true, then perform an output gyroscope insertion for a fresh output signal ic via the implicit place q and intermediate N''; in N'', $l({}^\bullet q) \subseteq Sig_r^{\pm}$ and $l(q^\bullet) \subseteq Out_d^{\pm} \cup \bigcup_{j \in J} Out_j^{\pm}$ where $J \subset I$ and C_j is called auxiliary component; this gives N'

2 C'_r is obtained from reduction of N' with partition member $(In_r, Out_r \cup \{ic\})$

3 C'_d is obtained from reduction of N' with partition member $(In_d \cup \{ic\}, Out_d)$

4 **for all** $j \in J$ **do** C'_j is obtained from reduction of N' with partition member $(In_j \cup \{ic\}, Out_j)$

Figure 4.12: Algorithm Avoid-1 for inserting internal communication to avoid an irreducible CSC conflict – insertImplicitPlace() is specified in InsertImplicit in Figure 4.8.

there, this conflict must concern q, i.e. without q they would be concurrently enabled. With the implicit q, they can still fire one after the other in any order. Since q has no loops, this means that they are concurrently enabled under M, so in fact there is no conflict in N''.

If q is safe, there can be no conflict in N' as well: t_1 and t_2 are only enabled when p_{out} is marked, and then it is marked with one token as q was. Furthermore, the transitions for the new signal ic are not in conflict with any other transition.

If q is not safe, some reachable marking of N' puts at least two tokens on p_{in} resulting in a dynamic self-trigger for the latter transitions. This self-trigger is still present in C'_d.

The correctness now follows from Theorem 4.1.10 as in the proof of Theorem 4.2.2. In addition to one delay component C_d, there are several auxiliary components $C_{a_1}, ..., C_{a_n}$. The partition in Avoid-1 is quasi-feasible, since

- the output transitions g_1 and g_2 are only in structural conflict with each other and they are labelled with the same signal anyway.

- g_1 and g_2 are only triggered by transitions with labels in $Sig_r = Sig'_r \setminus \{ic\}$ or by each other; they only trigger transitions with labels in $Out_d + \bigcup_{j \in J} Out_{a_j} = Out'_d + \bigcup_{j \in J} Out'_{a_j}$ and each other; $ic \in In'_d$, $ic \in In'_{a_j}$.

Consider a component C_k with $k \in I'$ and the corresponding initial component N'_k derived from N'. In N'_k, ic is lambdarised, since ic only triggers outputs of C_d and $C_{a_1}, ..., C_{a_n}$. As a first step, one can apply subnet contraction to the gyroscope yielding STG N''_k. The resulting place is implicit by Definition 4.1.6 and can be deleted yielding the original initial component N_k. Hence, N_k can be reduced to the final component C_k with the same sequence of reduction operations originally applied. □

In general, the components have many self-triggers or irreducible CSC conflicts resp., so AVOID-1 has to be applied a number of times. If AVOID-1 gets no solution for a conflict, then e.g. backtracking – i.e. reintroduction of an actual irrelevant signal – might avoid the considered conflict.

Optimising Gyroscope Insertion

Often, k components ($k > 1$) yield the same self-trigger, i.e. characterised by the same entry/exit transition pair in N. For each component a separate internal signal is needed, since it has to be produced by the respective component. However, it is not necessary to apply both algorithms for finding an appropriate delay transition (cf. Section 4.3.1) and INSERTIMPLICIT k times. After the first run of both algorithms, an implicit place q with its sets ${}^\bullet q$ and $q^\bullet$ is known, and one can insert k concurrent gyroscopes via k copies $q_1, ..., q_k$ of q, i.e. with ${}^\bullet q_i = {}^\bullet q$ and ${q_i}^\bullet = q^\bullet, \forall i \in \{1...k\}$, see Figure 4.13(a).

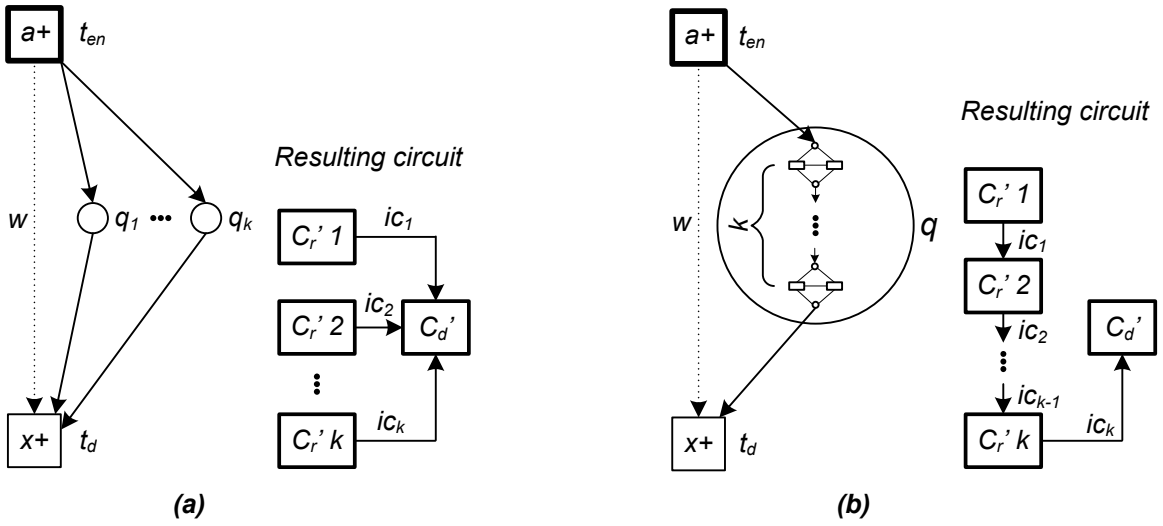


Figure 4.13: The same conflict occurs in k different components: (a) concurrent gyroscope insertion, (b) sequential gyroscope insertion.

Observe that this approach massively increases the concurrency degree of the new

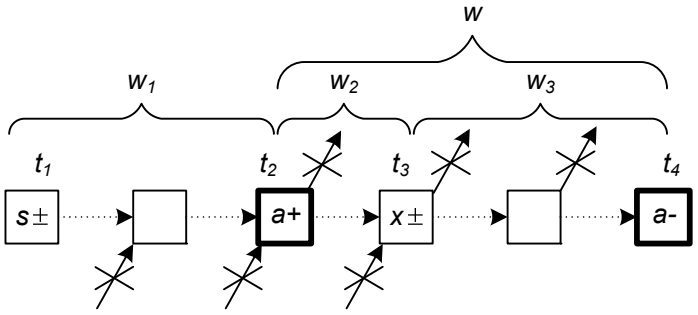
delay component (and also of the auxiliary components, not shown here), since after q_i's refinement – with a new gyroscope labelled with ic_i – each transition labelled with ic_i is a trigger of t_d; so each ic_i will be a relevant signal for the new delay component, i.e. $ic_i \in In_d, \forall i \in \{1...k\}$. This can be avoided by introducing only one place q and refine it via a sequence of k gyroscopes, as shown in Figure 4.13(b). This is correct w.r.t. the concurrent insertion in Figure 4.13(a), since each of the k internal gyroscopes must fire to activate t_d, but the order of firing does not matter, because each signal ic_i is internal (i.e its edge occurrence has no direct influence on an input occurrence from the environment). Hence, one can arbitrarily model a sequence of gyroscopes, as in Figure 4.13(b); consequently, the new delay component gets only one new relevant signal instead of k signals. Of course, the chosen sequence of the k gyroscopes might have effects on the final circuit's performance, area and/or its synthesis time; this will be investigated in future work.

4.4 Optimisation for Simple Structured STGs

In this section, a very simple case of self-triggering will be considered, where the specification is essentially an MG-path in the relevant part. Assume a deterministic and *live* net N is given; also the latter is a common and sensible assumption on STGs, and in fact the proofs below only presuppose 2-liveness. Further, we restrict ourselves to the CIR-algorithm; recall that this algorithm preserves liveness and 2-liveness according to Proposition 4.1.1.

Let $(C)_{i\in I}$ be correct w.r.t. N. In the critical component C_r, let there be an MG-self-trigger due to two input transitions t_2 and t_4 labelled with a^+ and a^- for $a \in In_N$. These transitions, the specification N and all components are inputs to the algorithm AvoidNoRecalc in Figure 4.14, which returns a modified specification and modified critical and delay components; the critical component is always, and the delay component if possible, generated without a second decomposition pass.

In general, the algorithm returns FAIL whenever the structural conditions in line 1 for the specification are not fulfilled. A path between t_2 and t_4 (line 1) is guaranteed to exist since such a path exists in C_r and the reduction does not generate new paths, and sometimes there will be an MG-path; condition (a) guarantees that contracting all transitions on w results in the place p_{st} that 'carries' the self-trigger. However, the path might fail to be non-forking (d) or there might be no proper output x which can help to destroy the self-triggering (b). In this case, the self-triggering might be avoided by adding additional inputs to C_r as discussed in Section 4.2.



Input	specification N — all components $(C_i)_{i\in I}$ — index r of critical component — MG-self-trigger place p_{st} in C_r with ${}^\bullet p_{st} = \{t_2\}$ and $p_{st}^\bullet = \{t_4\}$
Output	modified specification N' with new output ic modified critical component C'_r — modified delay component C'_d

1 in N, find an MG-path w from t_2 to t_4 with the following properties
return FAIL if no such w exists
(a) $M_N(w) = M_{C_r}(p_{st})$ and $l(w \cap T \setminus \{t_2, t_4\}) \cap Sig_r^{\pm} = \emptyset$
(b) a delay transition t_3 of w is labelled with an output edge $x^{\pm}$
(c) w_2, w_3 are the sub-paths of w from t_2 to t_3, from t_3 to t_4 resp.
(d) no transition of $w \setminus \{t_4\}$ has more than one outgoing arc

2 choose d such that $x \in Out_d$, choose a fresh signal $ic \notin Sig_N$

3 in N, perform an output gyroscope insertion between t_2 and t_3 for ic with the initial marking $(M_N(w_2), 0)$, giving N' (via an intermediate N'' with an additional p)

4 in C_r, perform an output gyroscope insertion between t_2 and t_4 for ic with the initial marking $(M_N(w_2), M_N(w_3))$, delete p_{st}; this gives C'_r

5 in N, find an MG-path w_1 from some transition t_1 to t_2 with the following properties ($t_1 = t_2$ is possible):
• $l(t_1) \in Sig_d^{\pm}$ and $l(w_1 \cap T \setminus \{t_1\}) \cap Sig_d^{\pm} = \emptyset$
• no transition of $w_1 \setminus \{t_1\} \cup w_2$ has more than one incoming arc
if no such w_1 exists, *return* with C'_d obtained from N' and the respective partition member $(In_d \cup \{ic\}, Out_d)$ by reduction

6 in C_d, perform an input gyroscope insertion between t_1 and t_3 for ic with the initial marking $(M_N(w_1 \cup w_2), 0)$, giving C'_d

Figure 4.14: Algorithm Avoid-NoRecalc for inserting internal communication without recalculation.

If all conditions are satisfied, a possible delay transition labelled with output x and the delay component C_d producing it are chosen (line 2). In the remaining part of the algorithm, the corresponding gyroscope insertions into N as well as into the components C_r and C_d take place, where for C_d a proper starting transition t_1 has to be found (line 5); cf. Figure 4.3 for the special case $t_1 = t_{en}$ or $t_1 = t_2$ resp. If the latter is not possible, C'_d can be recalculated from the modified specification N'.

A concrete implementation has some freedom since there might be more than one suitable output on w, i.e. there might be several possible delay transitions. In such a case, one might choose an output and the corresponding delay component such that a path w_1 exists at all or is better suited than others. If $a \in Sig_d$, then $t_1 = t_2$ and the conditions for w_1 are trivially fulfilled. In this case, t_2 might additionally be a syntactical trigger of t_3 in C_d; then, it is possible that the signal a is no longer necessary and might be lambdarised in C'_d and contracted, resulting in a correct decomposition.[8]

Theorem 4.4.1
For 2-live nets, the algorithm Avoid-NoRecalc *is correct in the sense of Theorem 4.2.2. The self-trigger in C_r is not present in C'_r anymore.*

Proof. Regarding termination, observe that N is finite and therefore the path search terminates. In the rest of the proof, it will be shown that C'_r and C'_d can also be derived via a correct reduction from N', and the claim then follows from Theorem 4.2.2.

Consider the component C'_r. Let $(R_m)_{0 \leq m \leq n}$ be the intermediate STGs encountered during the generation of C_r. Now it will be shown that the same sequence of reduction operations can be applied to the initial component N'_r obtained from N', resulting in a sequence $(R'_m)_{0 \leq m \leq n}$ of intermediate STGs with $R'_n = C'_r$ and the following properties: in every R'_m:

(1) there is exactly one transition $t^m \in p^\bullet_{out}$ with $t^m \neq t_2$, and

(2) there are non-forking paths v^m_1 from t_2 to t^m and v^m_2 from t^m to t_4 (here, one can ignore $p_{in} \in t_2{}^\bullet$ and t_4 can have more than one place in its postset).

(3) R'_m can be derived from R_m by an output gyroscope insertion between t_2 and t^m with initial marking $(M_N(w_2), M_N(w_3) - M_{R'_m}(v^m_2))$ (via addition of p giving an intermediate R'').

[8] The next proof shows that C'_d can be obtained from N' with the decomposition algorithm. If a is lambdarised in the respective initial component, the same sequence of reduction operations can be applied to get C'_d with a lambdarised. If continued reduction removes these λ-transitions, the result is one which someone could also get from the decomposition algorithm, hence it is correct.

Clearly, this is initially true: for R_0 and R'_0 choose $t^0 = t_3$, $v_1^0 = w_2$ and $v_2^0 = w_3$. So let the claim be fulfilled for some R_m and R'_m and let R_{m+1} be obtained from R_m by some reduction operation; the three cases for this reduction operation will be considered next. Since the CIR algorithm preserves 2-liveness, this is therefore given for all nets considered.

Deletion of an implicit place p' Observe that no place q of v_1^m or v_2^m is implicit since Lemma 4.1.5 implies in this case that there is another path between ${}^\bullet q$ and $q^\bullet$, a contradiction to v_i^m being non-forking. Then, Proposition 4.1.8(2) implies that $N' \backslash p'$ is the gyroscope insertion of $N \backslash p'$. Choose $v_i^{m+1} = v_i^m$ $(i = 1, 2)$ and $t^{m+1} = t^m$.

Deletion of a redundant transition t Observe that a transition on $v_1^m\, v_2^m$ is adjacent to an MG-place on $v_1^m\, v_2^m$, since $v_1^m\, v_2^m$ leads from t_2 to $t_4 \neq t_2$; hence these transitions are not redundant. Also, this observation implies that, in R''_m, t is not adjacent to p (nor is its duplicate if it exists). Now, Proposition 4.1.4(4) implies that t is also redundant in R''_m and R'_m, p is implicit in $R''_{m+1} = R''_m \setminus t$ by Proposition 4.1.4(3), and R'_{m+1} is obtained from the gyroscope insertion in R_{m+1}.

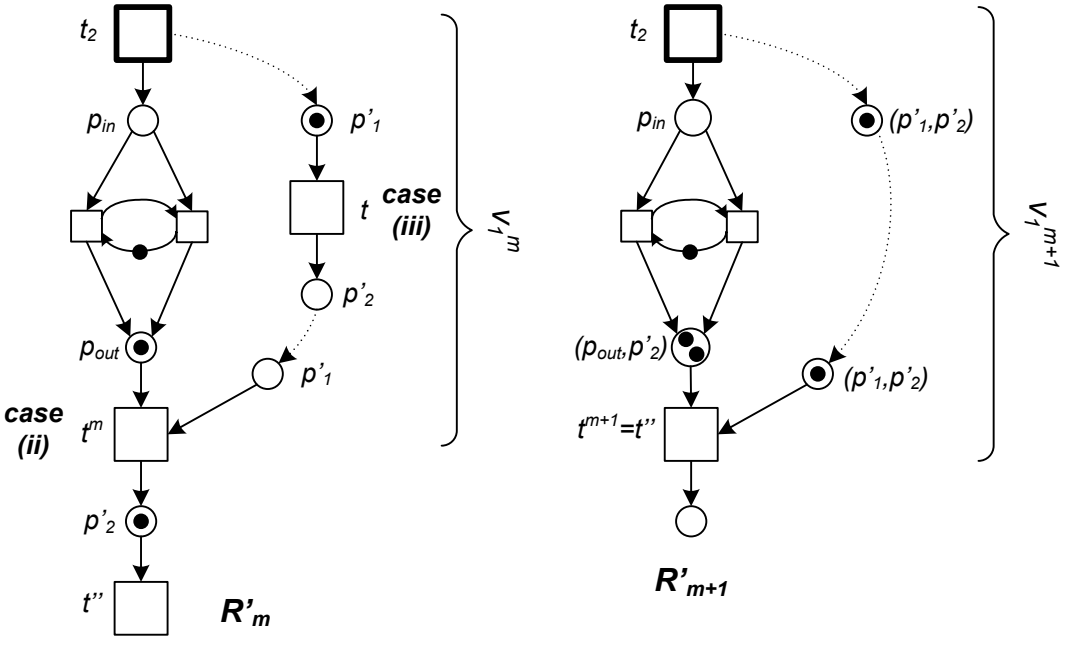


Figure 4.15: For the proof of Theorem 4.4.1. Cases (ii) and (iii) of transition contraction.

Contraction of a transition t Figure 4.15 shows the cases (ii) and (iii), both in one figure; some elements appear twice, once for each case.

(i) $t \notin v_1^m, v_2^m$: claim obviously fulfilled. For (3) observe that in the intermediate R''_m and hence also in R''_{m+1}, p is a shortcut place w.r.t v_1^m.

In all other cases, the path $v_1^m\, v_2^m$ is modified. Here, it is important that all transitions on $v_1^m\, v_2^m$ (except t_4) are non-forking; this property also holds for the modified path.

(ii) $t = t^m$: by assumption, there is only one place p'_2 in the postset of t^m ($t = t_4$ is not possible since t_4 is not λ-labelled). The contraction of t^m is possible in R'_m since the additional place $p_{out} \in {}^\bullet t_m$ does not render the contraction insecure. Let $v_1^m = t_2 v p'_1 t^m$ for some path v and $v_2^m = t^m p'_2 v'$ for some path v'. The places p'_1, p'_2 and p_{out} are replaced in R'_{m+1} by the places (p'_1, p'_2), (p_{out}, p'_2); for simplicity the latter is identified with p_{out}. Then, t^{m+1} is the single transition in $p'^\bullet_2$, $v_1^{m+1} = t_2 v (p'_1, p'_2) t^{m+1}$ and $v_2^{m+1} = v'$ are non-forking, and furthermore:

$$
\begin{aligned}
& M_{R'_{m+1}}((p_{out}, p'_2)) \\
(\text{definition}) \quad &= M_{R'_m}(p_{out}) + M_{R'_m}(p'_2) \\
(\text{induction}) \quad &= M_N(w_3) - M_{R'_m}(v_2^m) + M_{R'_m}(p'_2) \\
&= M_N(w_3) - (M_{R'_m}(t^m p'_2 v') - M_{R'_m}(p'_2)) \\
&= M_N(w_3) - M_{R'_m}(v') \\
&= M_N(w_3) - M_{R'_{m+1}}(v_2^{m+1})
\end{aligned}
$$

p_{in} is not touched by the contraction and therefore (3) follows. Observe that the initial marking of $\{p_{in}, p_{out}\}$ increases by $M_{R'_m}(p'_2)$ and so does the marking of v_1^{m+1} compared to that of v_1^m; hence, the gyroscope insertion leading to R'_{m+1} is indeed based on a shortcut place p.

(iii) $t \in v_1^m \setminus \{t^m\}$: let $v_1^m = t_2 v' p'_1 t p'_2 v'' t^m$ for some paths v' and v''. Since t is not adjacent to the inserted gyroscope, the contraction is also possible in the intermediate R''_m leaving p being a shortcut place in R''_{m+1}, and also in R'_m.

In R'_{m+1}, choose $v_1^{m+1} = t_2 v' (p'_1, p'_2) v'' t^m$, $v_2^{m+1} = v_2^m$ and $t^{m+1} = t^m$.

(iv) $t \in v_2^m \setminus \{t^m\}$: similar to (iii)

In R'_n, $t^n = t_4$, $v_1^n = t_2 p_{st} t_4$ and $v_2^n = t_4$ since all transitions of $w_3 \setminus \{t_4\}$ have been contracted. Hence, $M_{R'_n}(p_{in}) = M_N(w_2)$ and $M_{R'_n}(p_{out}) = M_N(w_3)$. The gyroscope insertion first inserts a place p resulting in an intermediate STG R''_n such that $M_{R''_n}(p) = M_{R'_n}(p_{in}) + M_{R'_n}(p_{out}) = M_N(w_2 \cup w_3) = M_N(w) = M_N(p_{st})$. Therefore, p_{st} is implicit in R''_n and due to Proposition 4.1.7(3) also implicit in R'_n. Deleting it there results in the final component C'_r returned by the algorithm.

If C'_d is recalculated from N', it is obviously correct. Otherwise, the proof is analogous to the previous one. Observe that this time p_{out} stays untouched while p_{in} is combined with places from w_1; in particular, w_1 has to be *non-joining* to avoid the 'duplication' of p_{in}. □

With this result, Avoid-NoRecalc can be applied repeatedly, as described for Avoid-0 above, and this gives a correct decomposition in the end.

The Theorem above shows that an output has been inserted into the self-trigger that records the occurrence of a^+. This removes the original irreducible CSC conflict, where the circuit for C_r (CC in Figure 4.2) had to produce y^+ after a^- but could not distinguish this state from the one before a^+. However, it could very well be the case that there is still some CSC conflict: the first a^+ is recorded by ic^+, the second one by ic^-; so the state vector before the first a^+ could easily be the same as the one after the second a^-, where y^+ has to be produced. It has been achieved that this CSC conflict can now be solved by inserting internal signals into C_r alone as described in [CKK+02] and proven correct in the sense of Definition 3.2.3, see [SV07].

In fact, the irreducible CSC conflict can also vanish completely if the two state vectors just mentioned differ in other signals. For the SPT-benchmarks presented in Figure 4.4 and reported in the next section, this is actually always the case; here, the reason is that always at least two gyroscopes are inserted 'non-concurrently'.

To understand this more in detail, consider the case that ic_a is inserted between a^+ and a^-, and ic_b between some b^+ and b^-. If ic_a and ic_b cannot fire concurrently, but only in some fixed order, the following firing sequence is possible (instead of the markings, the state vectors are shown projected to the relevant signals, which are ordered according to occurrence): (0000) a^+ (1000) ic_a^+ (1100) a^- (0100) ... b^+ (0110) ic_b^+ (0111) b^- (0101) ... a^+ (1101) ic_a^- (1001) a^- (0001) ... b^+ (0101) ic_b^- (0001) b^- (0000). The two states after a^- are pairwise different from the two states before a^+, and similarly for b. The situation discussed is somewhat reminiscent of the so-called *lock relation* [VGCM92] or *coupledness relation* [YP90] resp. (cf. also Section 5.4).

4.5 Experimental Results

The algorithms InsertImplicit, CheckTransition and Avoid-0, Avoid-1 as well as Avoid-NoRecalc have been integrated in our decomposition tool DesiJ (see Chapter 6). Avoid-1 is used as default algorithm for conflict avoidance, since it can handle the most general structured specifications; now all benchmarks in Table 4.1 can be synthesised. The optimisation Avoid-NoRecalc can only be applied very rarely for 'real-life' benchmarks, because they usually have more complicated structures than depicted in Figure 4.14 (here, only from the simple structured benchmarks SPT-6 and SPT-7[9] a synthesis is possible); furthermore, Avoid-NoRecalc does not offer the desired performance improvements for the considered benchmarks. Note that DesiJ-based logic synthesis would be impossible for any benchmark in Table 4.1 if neither Avoid-0 nor Avoid-1 nor Avoid-NoRecalc is applied.

For identifying a delay candidate as well as the paths w_1 and w_2, so far a simple solution is implemented. There is work in progress on integrating the ideas of Section 4.3.1. For irreducible CSC conflict detection the following approximation is used: for each component STG one can identify sub-structures as depicted in Figure 4.6 for which several entry/exit transition pairs per conflict can be identified (cf. Section 4.2.1).

Decomposition-based logic synthesis via DesiJ and Petrify as synthesis backend has been applied to the benchmark examples in Table 4.1. (In fact, the final logic synthesis using Mpsat might be more efficient, but the component STGs usually have complicated structures such that the structural methods of Mpsat sometimes fail to synthesise some of the components.)

Now these results will be compared with the ones from Moebius [CCCGV06], Mpsat [KKY04] and the pure application of Petrify [CKK+02]. Note that Moebius also implements an entire decomposition-based logic synthesis approach by:

1. encoding of specifications that do not satisfy CSC by solving complex ILP problems,

2. decomposing them (i.e. computing the projection[10] of the encoded specification for each single output signal, again by applying ILP), and

3. synthesising each component STG by applying state-based logic synthesis.

[9] These specifications result from control resynthesis of Sequence-Parallel-Trees having 6 or 7 levels resp., cf. also SPT-2 from Figure 4.4.

[10] another but restricted STG decomposition method

Benchmark	Size	DesiJ	Moebius	Mpsat	Petrify
	$\|P\|/\|T\|$	*Dec+Int+Syn/#*	*Enc+Proj+Syn/#*	*time/#*	*time/#*
2pp_wk.06	47/26	0+0+1/7	1/1	1/1	2/1
2pp_wk.09	71/38	0+0+1/7	1+1+0/1	4/1	13/1
2pp_wk.12	95/50	0+0+1/7	1+2+1/1	15/1	149/1
3pp_wk.06	70/38	0+0+2/12	2+2+0/2	5/2	90/3
3pp_wk.09	106/56	0+0+3/12	3+4+1/2	19/2	1466/3
3pp_wk.12	142/74	0+0+3/12	6+6+6/2	60/2	>**12h**
tsend-csm	34/29	0+0+8/9	0+0+1/2	1/2	2/2
Shifter	67/51	0+0+9/32	4+1+0/5	64/5	1480/5
ArbTree	83/72	0+1+4/31	75+7+1/16	196/6	1117/8
SPT-6	403/260	2+7+788/208	2160+360+240/56	m.o.	m.o.
SPT-7	659/516	5+20+822/272	**9.5h**+2280+0/115	m.o.	m.o.

Table 4.1: Comparison of different logic synthesis approaches w.r.t. synthesis time (in seconds) and number of inserted internal signals.

Initially, no considered benchmark example satisfies CSC, so the results cannot be compared to a decomposition-based synthesis with Nutas [YM07].

The benchmark computations, except for the Moebius results, were performed on a VMware® Linux guest system with 1 GB of RAM. The host machine was an Intel® Core™ 2 Duo E8400 CPU with 3 GHz and 4 GB of RAM running a Windows XP® 32-Bit version. The Moebius results were performed by Josep Carmona on an Intel® Xeon® CPU X3363 with 2.83 GHz and 24 GB of RAM running a Linux OS.

The first two columns of Table 4.1 report the benchmark names and their corresponding sizes in terms of place count ($|P|$) and transition count ($|T|$). The remaining columns report the synthesis times for each benchmark in seconds and the (cumulated) number of inserted internal communication signals (#) during synthesis. Note that in the DesiJ-column the synthesis time is split into decomposition time (*Dec*), the time for inserting internal communication signals as described in this paper (*Int*) and the cumulated time for Petrify synthesis of the component STGs (*Syn*). In the Moebius-column the times are also split into the encoding time to solve CSC for the specification N (*Enc*), the time to compute the projection of N for each output signal (*Proj*), and the 'Petrify-like' logic synthesis for the projected components (*Syn*). Observe that the encoding time for SPT-7 is 9.5 hours and Petrify cannot synthesise a circuit for 3pp_wk.12 after 12 hours of computation; all the other times are reported in seconds.

All benchmarks in the lower half of Table 4.1 arise when resynthesising handshake circuits consisting of control components, cf. Section 2.3.3 or [FC08], while the 2pp_wk.??

and 3pp_wk.?? specifications are not related to resynthesis.

All benchmarks up to 100 places or transitions can be handled within a few seconds by DesiJ. For instance, every other tool needs at least 83 seconds to synthesise the ArbTree circuit. To the best of my knowledge, DesiJ-based synthesis does not only enable the synthesis of circuits by orders of magnitude faster than any other approach, but it even enables the synthesis of circuits which are impossible to synthesise with pure logic synthesis: e.g. Petrify and Mpsat cannot synthesise a circuit for SPT-6 and SPT-7 due to memory overflows (m.o.); Moebius takes approximately 10 hours to synthesise a circuit from SPT-7, while the presented approach can manage this in less than 15 minutes.

For the presented benchmarks it does not matter whether applying Avoid-0 or Avoid-1, since the specifications have almost marked graph structures. At the moment, it is not easy to exploit the superiority of Avoid-1; this will be discussed later in Section 4.6. The difference in time consumption for Avoid-0 and Avoid-1 is negligible, since both approaches work purely structural. Avoid-NoRecalc can only be applied for SPT-6 and SPT-7. However, for these benchmarks it consumes more time for internal communication insertion than Avoid-0 or Avoid-1 resp.: for SPT-6 it needs 13 seconds and for SPT-7 30 seconds, since for each conflict three gyroscope insertions (for the specification, critical and delay component) instead of only one gyroscope insertion (for the specification) have to be done.

Observe that DesiJ-based synthesis is not always the best approach to exploit the advantages of logic synthesis. In particular, for a small specification like tsend-csm a pure Mpsat or Petrify synthesis is better suited. DesiJ often introduces more internal signals than a pure logic synthesis would do. E.g. for the 2pp_wk.?? benchmarks, each of Petrify, Mpsat or Moebius only inserts one internal signal to solve CSC. With total decomposition (i.e. constructing one component for each output), DesiJ must avoid two self-triggers by introducing two internal signals, and during logic synthesis of the components Petrify introduces 5 CSC signals – hence, the DesiJ-based synthesis needs 7 internal signals to solve CSC for each 2pp_wk.?? benchmark; this might eventually consume more chip area.

As a consequence, DesiJ shall only be applied for very large specifications where logic synthesis with other tools is impossible or takes unacceptably long time. Furthermore, total decomposition is not always the best approach, since it leads to many component STGs which potentially require many internal signal insertions to avoid irreducible CSC conflicts (and so leads to larger circuits). In Chapter 5 the decomposition will be tuned to find output partitions that result in component STGs having fewer irreducible CSC

conflicts; indeed, these component STGs might be larger, but the partitioning must be done carefully such that the component STGs are still synthesisable with PETRIFY, PUNF&MPSAT or MOEBIUS. This might maximally exploit the opportunities of logic synthesis, and might lead to the most efficient circuit implementations that can be achieved.

4.6 Limitations and Opportunities

In this chapter, an approach to avoid irreducible CSC conflicts by introducing internal communication between the components (by applying so-called gyroscope insertion) was presented. The gyroscope insertion usually transforms an irreducible CSC conflict of a so-called critical component into a reducible one. Sometimes the considered conflict could even vanish completely by a suitable introduction of the gyroscopes, cf. the discussion at the end of Section 4.4.

Although the benchmarks presented in the latter section are promising there are still examples which are not completely synthesisable, but a remarkable progress for them was reached anyway. For example, there is specification which models a part of the control behaviour for a BALSA-specified Stack circuit. It consists of 206 transitions and 272 places. This specification is too complex for applying pure logic synthesis. With DESIJ-based synthesis, but without applying internal communication insertion, only 10 of 47 components can be synthesised. Instead, with internal communication insertion the synthesis of 32 of 47 components is possible. The remaining 15 components still have irreducible conflicts, because of the following limitations of the presented approach.

Figure 4.16 shows specifications that can serve as an example for the limitations of the solution presented in Section 4.3: choose the a^+-labelled transition as entry transition t_{en} and the a^--labelled one as exit transition t_{ex} yielding a self-trigger for the critical component producing y.

When applying INSERTIMPLICIT (Figure 4.8) and AVOID-1 (Figure 4.14) or AVOID-0 resp., the pending self-trigger for the critical component resulting from the reduction of the specification in Figure 4.16(a) can be avoided by inserting the gyroscope at the cloud's position. However, for (b) the conflict *cannot* be avoided by inserting the gyroscope at the position of the cloud, although the application of INSERTIMPLICIT would suggest to insert an implicit place q at this position. Even AVOID-1 allows the generation of distinct component STGs for producing x_1 and x_2. However, the problem is that t_1 or t_2 cannot be identified as delay transitions or delay candidates according to Definition 4.3.1.

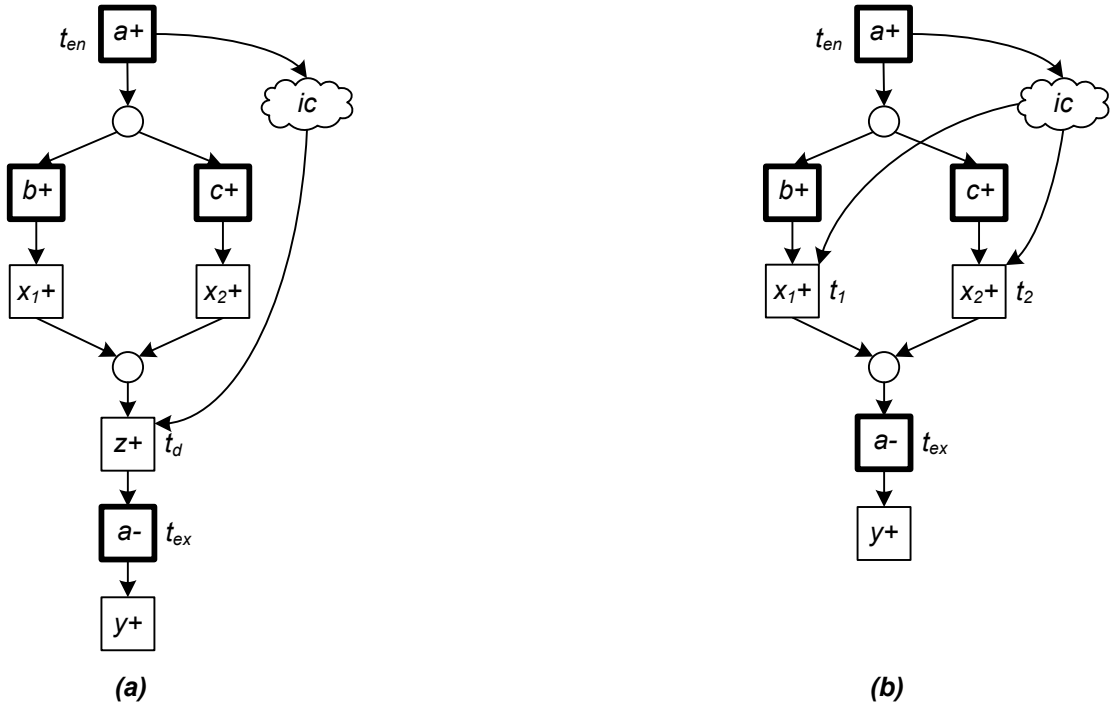


Figure 4.16: Specification with branching and merging resulting in two delay components.

A solution could be the relaxation of the requirement that a delay transition has *always* to fire 'between' t_{en} and t_{ex}. Thus, Definition 4.2.1 could be replaced by:

Definition 4.6.1 (Delay Transition)
A transition t_d of N is called *delay transition* w.r.t. a transition pair (t_{en}, t_{ex}) if $t_d \neq t_{en}$ is labelled with a local (i.e. non-input) signal edge and t_d occurs in *some* transition sequence t_{en} ... t_{ex} enabled under a reachable marking of N. △

Then one could define a quality criterion for delay transitions, such that a delay transition which occurs in more sequences t_{en} ... t_{ex} than another one should be preferred.

This would not only enable the handling of specification structures, as sketched in Figure 4.16(b), but this would enable the handling of the most general STG structures, because the requirement of having non-merging paths w_2, as in Figure 4.7, would not be necessary anymore.

Another problem is that the inserted gyroscopes sometimes do not avoid a *real* irreducible CSC conflict. In the current DesiJ implementation irreducible CSC conflicts are merely *approximated* by identifying sub-structures, as sketched in Figure 4.6, for each component STG. Indeed, this is very rough and sometimes these structural CSC conflicts are not dynamic ones. Thus, unnecessary gyroscopes will sometimes be inserted, yielding inefficient circuit implementations. To overcome the latter drawback,

irreducible CSC conflict detection could be done by using reachability analysis, e.g. with PUNF&MPSAT. This should not suffer from complexity problems, since it is done for the (usually) small component STGs, instead of the large specification.

Furthermore, the gyroscope insertion is not as efficient as possible, so far. On the one hand, if there are two 'overlapping' irreducible CSC conflicts (i.e. for two conflicts A and B the transition sequences v_1 or v_2 corresponding to A, cf. Figure 4.6, share some transitions with v_1 or v_2 corresponding to B), then there is no optimisation implemented so far in order to insert only one gyroscope for avoiding both conflicts. On the other hand, this can lead to the problem that one conflict will be avoided with more than one gyroscope. If the conflict A is already solved by a gyroscope insertion and for the conflict B there is a transition pair (t_{en}, t_{ex}) for which another gyroscope will be inserted to avoid B, but (t_{en}, t_{ex}) are part of v_1 or v_2 corresponding to A as well, then A will be avoided via two gyroscopes. Obviously, this is undesired and DESIJ should be improved to avoid this. Solutions to rather similar problems w.r.t. Petri net unfoldings have been presented in [KMY06].

In Section 4.2 two further possibilities to avoid irreducible CSC conflicts in the components were mentioned. First, an appropriate partitioning of the output signals was suggested; a solution will be presented later in Section 5.3. Second, the relevant signal detection procedure can be improved for critical components: one can use transitions t that fire between t_{en} and t_{ex} such that they will not be contracted during decomposition, but backtracking will be performed and $l(t)$ will be added as an *input* for the critical component. Related work can be found in [KS07].

To exploit the advantages of all these possibilities an overall heuristic might be developed in the following way:

1. A suitable output partitioning should be chosen in order to substantially reduce the number of irreducible CSC conflicts in the component STGs. However, this partitioning must be done carefully, since it must not lead to such complex components which are not synthesisable anymore (cf. also Chapter 5).

2. Remaining conflicts should be avoided by backtracking on signals which are at first glance not relevant for the critical component. Again this should be carefully done, since it could lead to an uncontrollable growth of the critical component, because in the worst case each 'backtracked' signal could lead to new irreducible CSC conflicts, preventing the final logic synthesis of this component.

3. If there are still remaining conflicts or if the critical component grows uncontrollably, then new internal communication signals, as presented in this chapter,

should be introduced in order to avoid the remaining conflicts.

An important issue for the success of control resynthesis is the performance improvement of the resulting circuits, in comparison to a synthesis based on syntax-directed translation. For this, the presented approach could be optimised by investigating how the choice of the delay candidate (among several possible delay candidates), cf. Section 4.3.1, can influence the synthesis time and the resulting circuit area and performance. Further research is needed to investigate how the order of the gyroscopes for a sequential gyroscope insertion, as in Section 4.3.3, influences the resulting circuit performance and its synthesis time.

Another possibility for optimisation might be the replacement of the gyroscope insertion by another operation: Indeed, a gyroscope insertion is easy but potentially doubles the state space of the respective component (since the state of the new signal is independent of all other signals), which can prevent synthesis. Alternatives are:

- the insertion of a two-way communication according to a 4-phase handshake protocol: the critical component sends ic^+_{req} to the delay component, which answers with ic^+_{ack} and then ic^-_{req} and ic^-_{ack} follow. This doubles the number of inserted signals but could keep the state space small, and thus might be needed for the synthesis of larger specifications.

- the insertion of a single signal edge ic^+, and the insertion of ic^- somewhere else such that consistency is preserved.[11] In particular, if two gyroscopes are inserted into one component as in the SPT-benchmarks from Section 4.5, one could be replaced by ic^+ and the other by ic^-. However, this might require a state space analysis of the component, since the respective signal edges must not be enabled concurrently to guarantee consistency. Hence, this might only be suitable for small components.

[11] This is quite similar to ordinary CSC solving, except that the constraints differ somehow. Hence, the same techniques used there might be helpful.

5 Partitioning Heuristics

This chapter presents heuristics for finding suitable output partitions, as a guidance for the STG decomposition, in order to improve the quality of the resulting circuits.[1] First notions about partitioning heuristics can be found in [Him99].

In the sections 5.1 to 5.4 four new heuristics are presented and experimental results are given in Section 5.5. Section 5.6 discusses the limitations of the presented approach, gives an outlook for further research and gives some first promising results when combining the considered partitioning heuristics.

The starting point for each heuristic is always an unimproved *feasible* partition and the outcome is an improved *feasible* one.

First, each heuristic will be introduced with the help of Figure 5.1. Then an algorithmic solution will be presented for each of them. Figure 5.1 shows just fragments of safe, live and consistent STGs: (a) shows a part of a large specification STG, while (b), (c), (d), (e) and (f) show (the resp. parts of) the components resulting from total decomposition of (a). Until Section 5.4 forget about place q and the dashed arcs in (a) – it will explicitly be mentioned when they are needed. For simplicity, assume that one can do reasoning just on these parts of STGs, i.e. that the unseen parts do not contradict this reasoning.

[1] The content of this chapter is based on [WVW11].

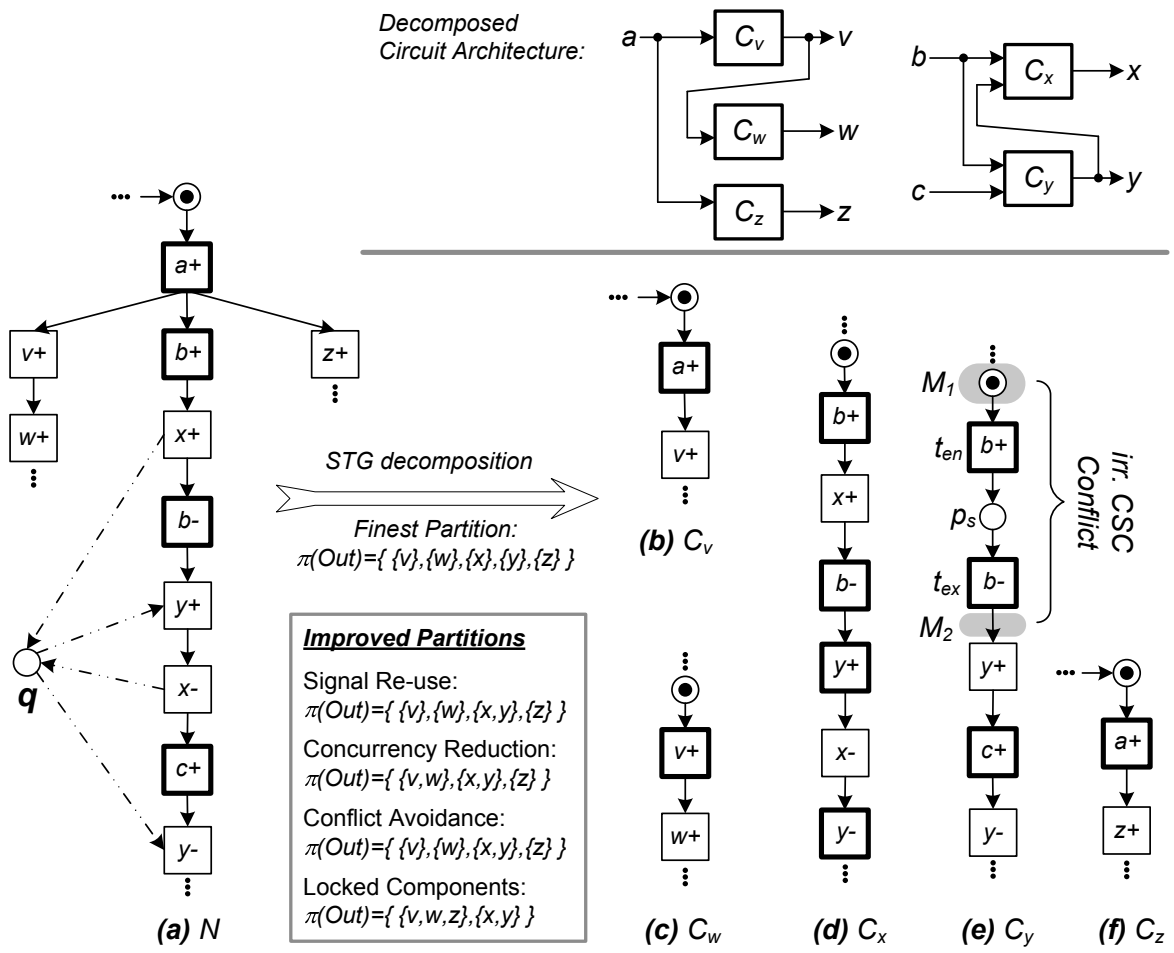


Figure 5.1: Fragments of consistent, live and safe STGs: a specification (a) and the components (b), (c), (d), (e) and (f) after total decomposition. In (a) ignore place q and the dashed arcs until explicitly mentioned.

5.1 Signal Re-use

Other decomposition-based methods, e.g. [CCCGV06, YM07], always apply total decomposition. Our STG decomposition is more general, and this allows us to take advantage of the following observation. After decomposition, any two components C_i and C_j model the occurrence of different outputs, but the two sets of external signals sometimes overlap to a large extent – or one might even be contained in the other. In case of a large overlap, it seems to be better to determine just one component C_{ij} with outputs $Out_i \cup Out_j$, because it will have only slightly more signals than each of C_i and C_j. Thus, logic synthesis will often be just as possible for C_{ij}, and the resulting circuit is usually more efficient (in terms of area) than the two circuits resulting from synthesising C_i and C_j separately. (This efficiency gain is caused by a kind of 'internal signal sharing', i.e. a CSC conflict from C_i and another one from C_j might be solved by the same internal signal in C_{ij}.)

For a feasible partition, two of its members are called *compatible*, if their external signals overlap to a large extent; the idea is to improve the feasible partition by merging such members.

To *merge* two members (In_1, Out_1) and (In_2, Out_2) of a partition π simply means to replace both members in π by $((In_1 \cup In_2) \setminus (Out_1 \cup Out_2), Out_1 \cup Out_2)$. It is easy to see that the resulting partition will be feasible again; note that, for each output in Out_1 and Out_2 resp., all essential or relevant signals (i.e. its trigger signals as well as its conflicting signals) must be in the merged partition member; this is the case since these signals are already in $In_1 \cup Out_1$ and $In_2 \cup Out_2$ resp.

Practically, compatibility is determined w.r.t. an initial feasible partition without computing the final components. On the one hand, this saves time; on the other, the final components usually contain more relevant inputs, but these might be unnecessary for the combined components.

Definition 5.1.1 (r-Compatible)
Two initial partition members (In_1, Out_1) and (In_2, Out_2) are *r-compatible* if the fraction of the signals in Ext_1 that are also contained in Ext_2 is at least r, or vice versa. $\triangle$

'r-compatible' means that the external signals $(In_i \cup Out_i)$ of one initial component are covered by the external signals of another one to a certain extent; 'r' defines the fraction of overlapping and can be chosen arbitrarily.[2]

[2]It turned out that at least for the benchmarks (in Section 5.5) $r = 0.66$ was a good choice.

After identifying all r-compatible pairs of partition members, we try to find large *compatible sets* of members which are pairwise r-compatible, and then merge these members. A maximal compatible set is shortly called an *MC*. Note that r-compatible 1 and 2 as well as r-compatible 2 and 3 will not be merged if 1 and 3 are not r-compatible. This could lead to a large component C_{123} which might have too many signals, because 1 and 3 do not have many signals in common.

Regarding the specification and its component STGs in Figure 5.1, one can see that the inputs for the initial components are as follows: $In_v = \{a\}$, $In_w = \{v\}$, $In_x = \{b, y\}$, $In_y = \{b, c\}$ and $In_z = \{a\}$ (note that e.g. In_v specifies the inputs of the initial component producing output v). The fraction of overlapping of Ext_x and Ext_y (and vice versa) is $\frac{2}{3}$; according to Definition 5.1.1 and by using $r = 0.66$ one has to produce just one component C_{xy}. Consequently, a better partition of the outputs might be $\{\{v\}, \{w\}, \{x, y\}, \{z\}\}$.

For an algorithmic solution, a graph G has to be constructed with the partition members as vertices, and two vertices are adjacent if they are r-compatible. The vertices of this graph are tried to be covered by a minimum number of compatible sets, i.e. cliques of this graph; in other words, we try to partition the vertex set into a minimum number of cliques. So far, a fast greedy algorithm has been implemented that iteratively chooses an MC Q_0 in G and then an MC Q_1 in $G \setminus Q_0$ etc., until no more vertices are left. In future work, better approximations will be investigated. Obviously, this will be computationally more complex, but since the graphs G are usually not very large, it might not become a bottleneck for the decomposition algorithm.

Since too large components cannot be synthesized efficiently in time, one could specify an upper bound of external signals per compatible set instead of searching for an MC; however for the benchmarks presented in Section 5.5, this has not been necessary, because the new components have been small enough for synthesis.

5.2 Concurrency Reduction

Improving a partition by merging partition members (as described in the last section) might result in component STGs that are not synthesisable anymore because of state space explosion. The main reason for the latter is a large concurrency degree of the improved components. Consequently, a heuristic will be presented that merges members that increase the concurrency degree of the resulting merged components by only a small percentage.

For efficiency, the concurrency or sequentiality resp. of two partition members will be

defined w.r.t. the signals of the corresponding initial components $(C_i)_{i \in I}$ again, instead of the final ones.

Definition 5.2.1 (Sequential Signal Pair)
Two signals s_1 and s_2 of an STG N are *sequential* iff all transition pairs of $\{(t_1, t_2) \mid l(t_1) = s_1^{\pm}$ and $l(t_2) = s_2^{\pm}\}$ are sequential, i.e. the two transitions are not enabled concurrently under any reachable marking; otherwise s_1 and s_2 are *concurrent*. △

Definition 5.2.2 (r-Sequential)
Two partition members (In_1, Out_1) and (In_2, Out_2) are *r-sequential* iff the fraction of sequential signal pairs in $\{(s_1, s_2) \mid s_1 \in Ext_1$ and $s_2 \in Ext_2\}$ is at least r. △

Considering the specification in Figure 5.1(a), all signals of the initial components C_v and C_w (i.e. $\{a, v\}$ and $\{v, w\}$ resp.) are pairwise sequential. Thus, it might be a good idea to produce only one component C_{vw} which has the same concurrency degree as either component. A similar situation applies for the initial components C_x and C_y. Consequently, an improved output partition (even for $r = 1$) – yielding components having a small concurrency degree – is $\{\{v, w\}, \{x, y\}, \{z\}\}$.

As a first step for an implementation all pairs of partition members which are r-sequential have to be identified – calling them compatible – and then proceed as in Section 5.1. Again, to avoid components having too many signals, one can specify an upper bound of signals per compatible set, i.e. resulting partition member. Here, this upper bound might be really necessary: components having a large number of signals – even though (almost) all of them are pairwise sequential – might not be synthesisable by a logic synthesiser like Petrify (note: this might not caused by state explosion, but the Boolean minimisation problem could suffer from complexity issues).

For checking the r-sequentiality of two partition members, one has to check the sequentiality of the respective signal pairs (see Definition 5.2.2); thus, one has to check whether the corresponding transition pairs are sequential or not, cf. Definition 5.2.1. In fact, this is $\mathcal{EXPSPACE}$-hard for arbitrary nets and $\mathcal{PSPACE}$-complete for safe ones [CEP95]. However, it can be approximated by using linear programming (instead of ILP):

Proposition 5.2.3
Let N be an STG and I be the incidence matrix of N. Two transitions t_1 and t_2 of N are sequential if the following LP problem is infeasible:

$$M_1 = M_N + I \cdot x_1$$
$$M_1[t_1 t_2\rangle \quad \textit{and} \quad M_1[t_2 t_1\rangle$$
$$M_1, x_1 \geq 0$$

Proof. If t_1 and t_2 are concurrent, i.e. not sequential, then there is a marking M_1 reachable via $v_1 \in T^*$ such that t_1 and t_2 can fire concurrently, hence $M_1[t_1 t_2\rangle$ and $M_1[t_2 t_1\rangle$. This gives a non-negative integer solution to the (in)equalities when taking x_1 as the Parikh vector of v_1. If there is no solution, t_1 and t_2 are definitely sequential. □

Of course, the infeasibility as an ILP problem is a better approximation for sequentiality than the infeasibility as an LP problem. However, solving ILP problems is more complex than solving LP problems. For the benchmarks in Section 5.5, the approximation with LP delivers promising results.

A different approach to check whether two transitions are not sequential is to compute the entire concurrency relation for N in advance. There is a polynomial algorithm for approximating the concurrency relation which is exact for free choice nets [KE96]. In future work, this approach could be compared to the linear programming approach presented here.

5.3 Conflict Avoidance

After applying STG decomposition, the final components might have irreducible CSC conflicts; e.g. component C_y in Figure 5.1(e) shows a self-trigger between the markings M_1 and M_2. The problem is that the environment of the component circuit for C_y might produce the input edge b^- so fast after producing b^+ that the circuit is not able to register b^+ completely; this can result in malfunctioning. But in fact the environment of the complete circuit has to wait for x^+ before it produces b^-, as shown by the specification in Figure 5.1(a). Thus, the x^+-labelled transition is a *delay transition* in sense of Definition 4.2.1, i.e. if t_{ex} fires after t_{en}, then t_d must fire in between.

W.r.t. our example the entry transition t_{en} is labelled with b^+, the exit transition t_{ex} with b^- and t_d with x^+. If C_y would additionally produce x, the self-trigger will be avoided. Hence, in our example a good output partition for avoiding irreducible CSC conflicts might (again) be $\{\{v\}, \{w\}, \{x, y\}, \{z\}\}$.

Often, for each irreducible CSC conflict, one can find at least one pair of transitions (t_{en}, t_{ex}) such that a delay transition resolves the conflict; for self-triggers there is only

one pair (t_{en}, t_{ex}), cf. the example above in Figure 5.1(e). For general irreducible CSC conflicts, one can identify more than one of these pairs, cf. Section 4.2.1.

Towards an algorithmic solution, a first decomposition pass using the original partition has to be applied. In contrast to Section 5.1, the final components are needed instead of the initial ones in order to detect irreducible CSC conflicts (or fragments that can be identified structurally and often represent *real* irreducible conflicts). Then, a respective transition pair (t_{en}, t_{ex}) has to be found for each conflict; this is easy for structural fragments such as syntactic self-triggers.

Next, one delay transition per conflict of some critical component C_i has to be identified. Each such delay transition corresponds to an output of another component. Thus, instead of computing separate components, one has to merge C_i to all these other components as in Section 5.1 (i.e. one has to merge all the respective partition members); this avoids irreducible conflicts of C_i. Here, this can easily lead to large partition blocks, and this would not alleviate the state explosion problem anymore. Hence, an upper bound of external signals per partition member should be specified again; remaining conflicts might be resolved by different approaches, cf. Chapter 4 and [KSVW09].

The last open question is how to identify a delay transition w.r.t. a transition pair (t_{en}, t_{ex}) in a way that hopefully avoids complexity problems arising from state space exploration, huge SAT problems related to Petri net unfoldings, or large ILP problems for state space approximation. One approach was already presented in Section 4.3.1: A *delay candidate* was defined purely on the graph structure of the specification. Indeed, this approximation was neither a sufficient nor a necessary condition for a delay transition, but for many benchmarks it was sufficient. An efficient algorithm to find these delay candidates was also presented.

Here a new approach is presented to find a delay transition by applying linear programming techniques again.

Proposition 5.3.1
Let N be a consistent specification STG and I be the incidence matrix of N. $t \in T$ is a delay transition w.r.t. the transition pair (t_{en}, t_{ex}) if the following LP problem is infeasible:

$$\begin{aligned} M_1 &= M_N + I \cdot x_1 \quad \textit{and} \quad M_1[t_{en}\rangle M_1' \\ M_2 &= M_1' + I \cdot x_2 \quad \textit{and} \quad M_2[t_{ex}\rangle \\ 0 &= x_2(t) \\ 0 &\leq x_1, x_2, M_1, M_1', M_2 \end{aligned}$$

Proof. If t is not a delay transition, then there exist a marking M_1 reachable via $v_1 \in T^*$ such that $M_1[t_{en}\rangle M_1'$ and $v_2 \in T^*$ without t and M_2 with $M_1'[v_2\rangle M_2[t_{ex}\rangle$. This gives a non-negative integer solution to the (in)equalities when taking x_1 and x_2 as the Parikh vectors of v_1 and v_2. □

To determine a delay transition for a transition pair (t_{en}, t_{ex}) one can check the above LP problem for infeasibility for each output transition t of the specification N. However, in case of a self-trigger, one can expect that the delay transition is an output transition that was contracted 'into' some place $p_s \in t_{en}{}^\bullet \cap {}^\bullet t_{ex}$ of the critical component, cf. Figure 5.1(e); i.e. the delay transition lies on some path from t_{en} to t_{ex} in N that does not contain a transition of the critical component in its interior. For efficiency, one can check the LP problem from Proposition 5.3.1 at first just for those output transitions t of N which are reachable by a depth first search (DFS) from t_{en} that terminates at transitions labelled with a signal of the critical component and furthermore t_{ex} should be reachable from t by applying a reverse DFS from t_{ex}. If no delay transition can be identified by this way, then the other output transitions of N have to be checked for being delay transitions.

5.4 Locked Components

Finally, a heuristic will be studied which is related to the approach in the last section to a certain extent. Established approaches for resolving CSC conflicts concentrate on the introduction of constraints among the signals of an STG, using lock relation or coupledness relation resp. as a guidance, cf. [VGCM92, YP90]. In order to get efficiently synthesisable final components, so-called r-locked members should only be merged, yielding an improved partition.

According to [VGCM92], the signals $s_1 \neq s_2$ of a consistent STG N are called *locked* if the occurrences of s_1 and s_2 alternate, i.e. between the occurrence of two edges of s_1 there always occurs an edge of s_2 and vice versa, for all traces of N. If s_1 and s_2 are locked, one can write $(s_1, s_2), (s_2, s_1) \in LS$; this defines the so-called *lock graph* G on the signals of N with edge set LS. For suitable subgraphs of G, the connected components are called *lock classes*.[3] Such components can also be determined in linear time with depth first search. Now the concept of r-locked partition members will be defined:

[3] This corresponds to the concept of *strongly coupledness* in [YP90].

Definition 5.4.1 (r-Locked)
Two partition members (In_1, Out_1) and (In_2, Out_2) are *r-locked* if some lock class L of the subgraph of G induced by $Ext_1 \cup Ext_2$ contains at least a fraction r of the signals of $Ext_1 \cup Ext_2$. △

Merging of r-locked members (by applying a high percentage of locking) may alleviate the CSC solving during synthesis of the corresponding merged components [VGCM92, YP90].

One can improve the finest partition for the specification N in Figure 5.1(a) by merging the members corresponding to components C_x and C_y: note that $Ext_x \cup Ext_y = \{b, c, x, y\}$, and that $(x, y), (b, x)$ and (c, y) are locked (remember it is assumed that the unseen STG part does not contradict our reasoning); $(b, c), (b, y)$ and (c, x) are not locked, but still the lock class is $\{b, c, x, y\} = Ext_x \cup Ext_y$, i.e. the partition members are 1-locked. A similar situation holds for merging the components C_v, C_w and C_z. Thus, an improved output partition is $\{\{v, w, z\}, \{x, y\}\}$.

Similarly as in previous sections, a set of pairwise r-locked partition members is called *lock-compatible*; such sets are merged as above, starting again (for more efficiency) from the initial components $(C_i)_{i \in I}$ instead of computing the final ones.

It remains to show how to check whether two signals x and y of an STG N are locked; for this, a similar approach as used for consistency checking in [GVC97] can be used. One can introduce a new place q in N – as in Figure 5.1(a) – which fulfills some properties as follows:

Proposition 5.4.2
Let N' be a consistent STG and $x, y \in Sig$. Let place $q \in P$ be connected to transitions $t \in T$ as follows: $W(t, q) = 1$ if $l(t) = x^{\pm}$, and $W(t, q) = 0$ otherwise; $W(q, t) = 1$ if $l(t) = y^{\pm}$, and $W(q, t) = 0$ otherwise.

If q is safe and implicit, then the signals x and y are locked in N which is obtained from N' by removing q.

Proof. Since q is safe, some y-transition must fire between any two x-transitions in all firing sequences of N', and these are the same in N since q is implicit. Whenever a y-transition fires, it empties q by safeness; hence, an x-transition must fire before the next y-transition. □

To check whether x and y are locked, a place q is added to N obtaining N' as described in the proposition; then q has to be checked whether it is safe and implicit. The only choice is whether to put a token on q or not. If, in each firing sequence of N, there

is an x-transition before a y-transition, there must be $M_{N'}(q) = 0$; if there is always a y-transition before an x-transition, there must be $M_{N'}(q) = 1$. Since one cannot know a priori which case is available, one has to try both possibilities.

To approximate the safeness and implicitness of a place q of a (very large) STG N linear programming can be used again.

Proposition 5.4.3
Let N be an STG and I be the incidence matrix of N. The place $q \in P$ of N is safe if the following LP problem is infeasible:

$$M = M_N + I \cdot x$$
$$M(q) \geq 2$$
$$M, x \geq 0$$

Proof. If q is unsafe, then there is a firing sequence $v \in T^*$ reaching a marking M where q has two or more tokens; then the Parikh vector x of v (and M) give a non-negative integer solution of the problem. □

For approximating implicitness one could check q for structural redundancy. If the ILP problem in Definition 3.2.2 has a solution, q is structurally redundant, i.e. q is implicit. Alternatively, here an idea inspired from [CC08] is used to approximate implicitness for a place q by solving LP problems instead of complex ILP problems.

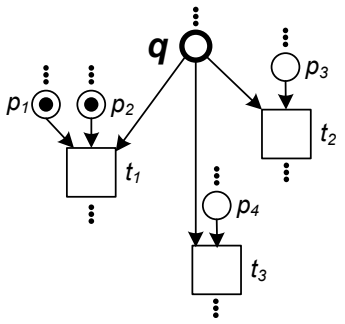


Figure 5.2: STG fragment showing a part of a reachable marking. The place q is not implicit.

The marking of an implicit place q is never the only restriction that prevents any transition t with $q \in {}^\bullet t$ from its firing. In Figure 5.2, the place q is not implicit, since all places of ${}^\bullet t_1 \setminus q$ are sufficiently marked, i.e. p_1 and p_2 have one token each; the insufficient marking of q is the only condition that prevents t_1 from firing. If $\forall t \in q^\bullet$ no such reachable marking exists, then q is implicit. This will be approximated by using

linear programming in the same way as above, where the undesired property was to have two tokens on q (a proof can be found in [STC98]):

Proposition 5.4.4
Let N be an STG and I be the incidence matrix of N. The place $q \in P$ of N is implicit if the following LP problem is infeasible $\forall t \in q^\bullet$:

$$M_1 = M_N + I \cdot x_1$$
$$\forall p \in {}^\bullet t \setminus q : M_1(p) \geq W(p,t)$$
$$M_1(q) \leq W(q,t) - 1$$
$$M_1, x_1 \geq 0$$

So far, it is required that a certain fraction of signals is covered by one lock class for r-locked partition members, cf. Definition 5.4.1. Since the results for this approach were not so promising as expected, it will be investigated in the future how the quality of the results changes if one replaces the applied lock class criterion by considering either cliques or weaker structures than connected components as suggested in [VGCM92, YP90]. Furthermore, an approximation will be considered where the implicitness check from Proposition 5.4.4 will be replaced with an LP check for Definition 3.2.2.

5.5 Experimental Results

Now, the partitioning heuristics will be demonstrated for the benchmark examples given in Table 5.1. For each benchmark, its size in terms of transition ($|T|$) and place count ($|P|$) is given. The bottom half presents specifications derived from handshake component circuits in the context of control resynthesis (cf. Section 2.3.3), while the top half presents some standard benchmarks – not related to control resynthesis. In order to compare the presented approach to different logic synthesis approaches, Table 5.1 reports the logic synthesis results for PUNF&MPSAT [KKY04], PETRIFY [CKK+02] and MOEBIUS [CC08]. For each benchmark and logic synthesiser, the synthesis *time* (in seconds) as well as an approximation for the circuit *area* in terms of literals of the Boolean equations resulting from complex gate synthesis is reported. Without overall CSC solving, it is impossible to compare the results with NUTAS [YM07], since none of the benchmarks has CSC initially.

Sometimes logic synthesis gave no result after two hours of computation ('timeout'), a memory overflow occurred ('mem. ov.'), or logic synthesis was impossible ('imposs.') due to either structural limitations of the unfolding-based synthesis approach or due to

	Benchmark	Size $\lvert T\rvert$–$\lvert P\rvert$	MPSAT *time/area*	PETRIFY *time/area*	MOEBIUS *time/area*
1	2pp_wk.09	38–71	4/89	21/90	3/90
2	2pp_wk.12	50–95	15/119	236/120	5/120
3	3pp_wk.06	38–70	5/90	89/99	4/93
4	3pp_wk.09	56–106	19/135	1467/144	9/138
5	3pp_wk.12	74–142	61/180	timeout	18/183
6	FifoStage	12–15	1/46	2/37	imposs.
7	MstRead	52–74	imposs.	timeout	imposs.
8	Shifter	72–98	64/76	842/79	7/76
9	ArbTree	104–115	196/128	2028/119	84/126
10	SMPS4	204–309	187/47	216/54	imposs.
11	SMPS5	418–636	mem. ov.	mem. ov.	imposs.
12	SMPS6	844–1285	mem. ov.	mem. ov.	imposs.
13	SeqTree5	252–372	mem. ov.	mem. ov.	688/183
14	SeqTree6	508–756	mem. ov.	mem. ov.	timeout
15	SeqTree7	1020–1524	mem. ov.	mem. ov.	timeout

Table 5.1: SI logic synthesis results.

incompleteness of the ILP-based CSC solving method.

The benchmark computations, except for the MOEBIUS results, were performed on a VMware® Linux guest system with 1 GB of RAM. The host machine was an Intel® Core™ 2 Duo E8400 CPU with 3 GHz and 4 GB of RAM running a Windows XP® 32-Bit version. The MOEBIUS results were performed by Josep Carmona on an Intel® Xeon® CPU X3363 with 2.83 GHz and 24 GB of RAM running a Linux OS.

Table 5.2 reports the results for decomposition-based logic synthesis using the tool DESIJ (see Chapter 6) and PETRIFY for logic synthesis of the final components. Each of the heuristics presented in Section 5 was integrated into DESIJ (cf. the header of Table 5.2) by using the free tool LPSOLVE [BEN04] as linear programming solver and the results are compared with total decomposition. Note that for each benchmark the introduction of internal communication between components as presented in Chapter 4 is necessary to enable logic synthesis of the final components. Again the synthesis *time* (i.e. the cumulated seconds for decomposition and PETRIFY logic synthesis) and circuit *area* (i.e. the cumulated literals count of the final component circuits using complex gate implementation) is reported. Furthermore, the number of resulting components for each benchmark and heuristic is given ($\lvert C\rvert$).

The diagram in Figure 5.3 visualizes the area reduction achievements for the bench-

marks 1 to 12 when applying DesiJ-based synthesis using the distinct partitioning heuristics. Each application of a heuristic can be compared with total decomposition, which is usually very area consuming, as well as with logic synthesis, which usually produces the most efficient circuits – hence the logic synthesis results can be seen as a lower bound for the results from DesiJ-based synthesis.[4]

B.	Total $\|C\|$–*time/area*	5.1 Re-use $\|C\|$–*time/area*	5.2 RedConc $\|C\|$–*time/area*	5.3 AvoidCSC $\|C\|$–*time/area*	5.4 Lock $\|C\|$–*time/area*
1	19–1/130	10–1/124	19–1/130	18–1/90	6–3/124
2	25–1/160	13–1/151	25–2/160	24–1/120	7–4/151
3	19–2/154	10–2/137	19–3/154	17–1/93	7–3/137
4	28–3/208	15–5/196	28–4/198	26–2/138	10–6/208
5	37–3/243	19–3/227	37–6/243	35–3/183	10–10/228
6	4–0/70	2–0/37	4–0/70	2–0/37	4–0/70
7	14–1/327	8–7/294	14–1/326	4–118/261	14–2/328
8	13–30/208	11–3/191	8–36/159	7–41/182	8–38/159
9	18–3/225	17–2/219	9–13/200	4–62/196	18–9/225
10	2–6/74	1–2/54	1–2/54	1–2/54	2–5/74
11	2–99/80	1–2/54	1–2/54	1–2/54	2–86/66
12	2–1952/126	mem. ov.	mem. ov.	mem. ov.	2–1818/126
13	33–1/289	33–1/289	7–72/295	7–81/315	32–30/289
14	65–3/577	65–3/577	13–477/572	14–122/629	64–318/577
15	129–6/1153	129–6/1153	26–5707/1162	27–265/1272	128–3703/1153

Table 5.2: DesiJ-based SI logic synthesis using different partitioning heuristics.

DesiJ-based synthesis includes the following steps: at first, one of the decomposition heuristics is applied or just the finest partition is used for total decomposition. Then decomposition is carried out and internal communication signals are inserted structurally – as *gyroscope insertions* – in order to avoid possibly still remaining irreducible CSC conflicts for the final components, cf. Chapter 4. Finally, Petrify synthesises the final component STGs; for this, it solves the remaining reducible CSC conflicts, again by internal signal insertion. This signal insertion is done on the state graph level; thus, it is more efficient w.r.t. circuit area but more time consuming. To sum up, the encoding step for DesiJ-based logic synthesis is split into the fast but area consuming gyroscope insertion and the slow but area efficient Petrify signal insertion.

Note that this two-step CSC solution usually yields circuits with more area than the circuits resulting from synthesis using the approaches from Table 5.1 (see also

[4] Note that the results for 'Logic Synthesis' represent the average values for the final circuit area when doing logic synthesis with Punf&Mpsat, Petrify or Moebius for each benchmark.

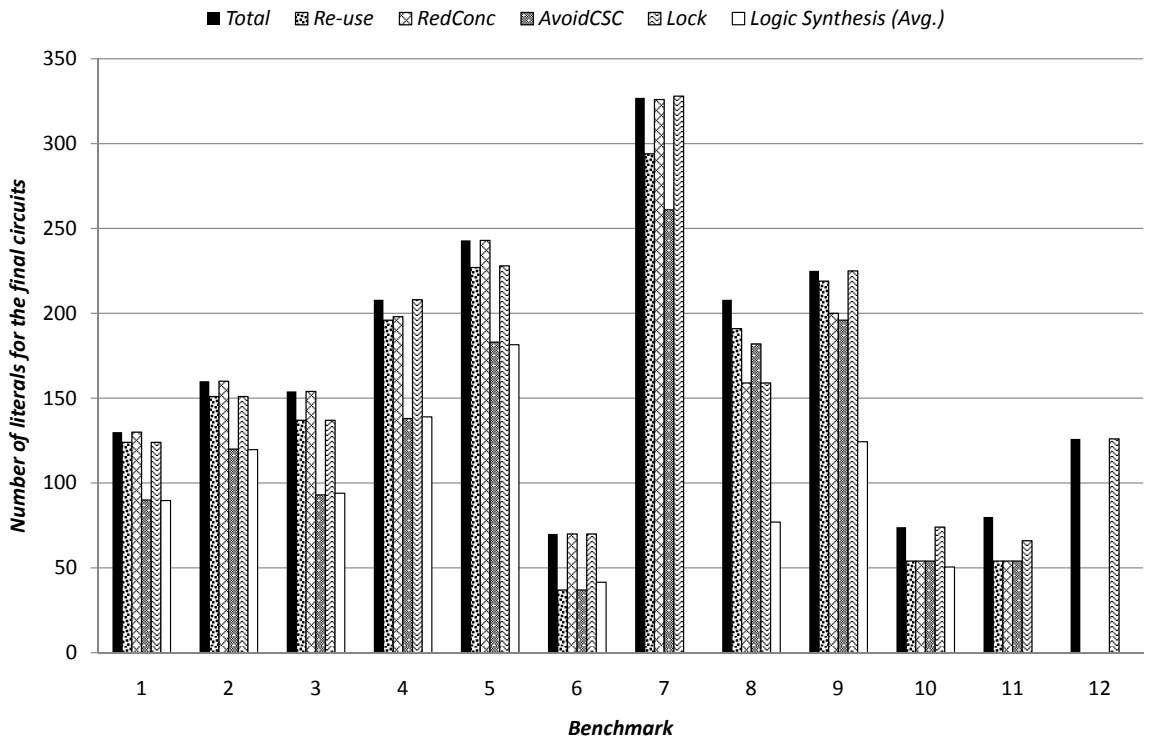


Figure 5.3: Comparison of area reduction results from Tables 5.1 (w.r.t. 'Logic Synthesis (Avg.)') and Table 5.2 for the benchmarks 1 to 12.

Figure 5.3), which solve CSC overall for the entire specification. However, the aim of DesiJ-based synthesis is basically to enable logic synthesis for very large specifications which are impossible to synthesise (as a whole) with any other logic synthesis approach, instead of producing the most efficient circuits. And indeed, larger specifications than with Moebius can even be handled (cf. benchmarks 14 and 15). This is because Moebius uses $\mathcal{NP}$-complete ILP techniques to solve CSC for the entire specification.

For the partitioning heuristics Re-use, RedConc, and Lock resp. the following values for parameter r were used: 0.66, 0.9 and 0.8 resp. A signal number restriction of 20 signals per components was applied for the heuristics AvoidCSC and Lock. These numbers have been determined empirically in extended experiments.

When applying partitioning heuristics instead of using total decomposition, in most cases a remarkable area reduction of the final circuits (up to a half, cf. benchmark 6 – the FifoStage) can be reached, cf. also Figure 5.3. This is achieved, either because one needs in total fewer encoding signals to solve CSC conflicts for the final components, or because in total fewer gyroscope insertions are needed even though Petrify inserts further signals, but in a more efficient way. On the downside, synthesis with partitioning heuristics usually lasts longer than with total decomposition, since Petrify usually

takes more time for signal insertion compared to the DESIJ gyroscope insertion which is purely structural; note that computing a partition and the decomposition itself is usually not a bottleneck.

In contrast, for the SeqTree benchmarks (13, 14 and 15), the partitioning heuristics cause no area improvement. The reason is that in these cases, the gyroscope insertions do not only turn irreducible CSC conflicts into reducible ones but actually solve all the conflicts as efficient as pure logic synthesis would do; hence, it does not hurt that total decomposition leads to more gyroscope insertions. This phenomenon might be related to the fact that the SeqTree benchmarks have a very simple structure; this relationship will be studied in the future. Furthermore, for these complex SeqTree examples even the LP solution becomes a bottleneck, because almost the entire synthesis time is used for computing the improved partition. Observe that DESIJ-based synthesis is the only approach that delivers a circuit for benchmarks 7, 11, 12, 14 and 15 within two hours.

AvoidCSC turned out to be a very good heuristic: it usually produces circuits with a similar area as the implementations resulting from pure logic synthesis (PETRIFY and PUNF&MPSAT), provided the latter works at all; cf. the benchmarks in Tables 5.1 and 5.2. AvoidCSC is better than total decomposition because fewer internal signal insertions are needed to solve CSC problems.

However, the results for the lock heuristic turned out to be worse than expected, because logic synthesis of the final components often needed many more internal signals than with AvoidCSC. The reason might be that the lock heuristic was applied just to initial partition members instead of final components. Recall that the applied decomposition method can also add relevant signals to the initial partition members; due to these added signals, the final components might not be locked so well anymore, i.e. the added signals might destroy the good CSC properties in the final components. In future work, this will be investigated in more detail.

In many cases (cf. examples 1 to 7), RedConc was not able to merge any partition members (even for parameter values down to 0.6) and, thus, delivers the same components as total decomposition. However, for real-life specifications from resynthesis context – like 8 and 9 – RedConc delivers the best results.

The strategy Re-use shows some promise over all examples; possibly, the results can be slightly improved by tweaking the parameter r and maybe choosing an upper signal bound per partition member. It might also be useful in combination with other heuristics.

As usual, the time for DESIJ-based synthesis in comparison to pure logic synthesis is either slightly better for small benchmarks or clearly superior for very large ones,

cf. the large benchmarks 9 to 15. Except for the Shifter benchmark, DesiJ is also always faster than decomposition-based synthesis with Moebius – the bottleneck for the Shifter benchmark is again the Petrify synthesis (i.e. CSC solving) of the final components and not the partitioning and decomposition. However, Moebius produces sometimes much better results for circuit area because of the superiority of its overall CSC solution approach, cf. the benchmarks 8, 9 and 13.

For benchmark 12, the merged components determined by Re-use, RedConc and AvoidCSC are too complex for Petrify synthesis.

It was also investigated how the results changed when using ILP instead of LP for the heuristics RedConc, AvoidCSC and Lock. In fact, the resulting components (and thus the final circuit area) were always the same; unexpectedly, ILP solving often needed a similar amount of time as LP solving:

One reason might be the size of the specifications, i.e. for small examples like FifoStage the complexity of ILP solving does not have a noticeable effect. Another reason might be that the very large benchmarks, e.g. SeqTree6, have a simple structure (often marked graphs) such that even LP delivers non-negative integer solutions for their reachable markings; note that an ILP solver using the branch-and-bound technique, like lpSolve, often first tries to transform the ILP problem by removing the integrality constraints on the variables, called its LP relaxation, and then tries to solve this problem; this LP relaxation might already deliver the correct results for the large 'simple structured' benchmarks.

However, for the benchmarks 1 to 5 and the Lock heuristic, ILP takes considerably longer than LP: for benchmark 4, the partition computation by using LP or ILP resp. needs 2.1 or 311 seconds resp. and for the fifth benchmark 5.3 or 1603 seconds resp.; but (as mentioned above) the resulting improved partitions were the same.

5.6 Limitations and Opportunities

The results presented in the latter section are promising, but there are still benchmarks which are not synthesisable. As already mentioned in Section 4.6, after decomposition and insertion of internal communication to avoid irreducible CSC conflicts there might be components which still have unresolvable conflicts, especially due to the current limitations of the approach presented in Chapter 4.

However, recent experiments show that if total decomposition fails, i.e. several components are not SI implementable, the proposed partitioning heuristics at least significantly increase the fraction of synthesisable components.

In particular for building 'better' components w.r.t. CSC satisfaction the 'locked components' heuristic should deliver better results. In future work, the use of different criteria for building lock classes will be studied and how the reasoning on final components, instead of initial partition members, influences the results.

In order to further improve the final circuit area, the effect of parameter tuning for each of the presented heuristics should be investigated as well as the presented algorithmic solutions should be improved, as suggested above.

Another open question is how these heuristics would combine. First experiments were conducted using a new heuristic which combines Re-use, RedConc, AvoidCSC and Lock: It basically works like Re-use from Section 5.1, but compatibility is defined by majority voting[5] based on the distinct criteria whether two partition members are compatible or 'mergable'. For example, if there are two partition members 1 and 2 and according to Re-use and RedConc resp. both are r-compatible and r-sequential resp., then 1 and 2 are compatible according to the new combined heuristic as well.

This combined heuristic was incorporated into DesiJ; it delivered first promising results. Figure 5.6 depicts these results by comparing them with total decomposition and logic synthesis for the benchmarks 1 to 11 from Section 5.5.

An area reduction w.r.t. total decomposition is always reached when using the combined heuristic, but in comparison to the isolated application of each heuristic the combined heuristic does not always deliver the best results. (The combined heuristic computes for the benchmarks 6 and 8, 9, 10, 11 the best results.) Remarkably, for benchmark 9 the combined heuristic delivers even a better result (177 literals) than each other heuristic in isolation. It might be interesting whether this effect can be confirmed by further experiments. The time consumption for computing this combined partitioning heuristic is roughly the cumulated time as if each heuristic is computed in isolation. Note, this has only a big impact when synthesising very complex benchmarks such as SeqTree7, where merely the computation of the partition needs 8739 seconds.

So far, the presented partitioning heuristics concern the improvement of CSC satisfaction for the final components as well as area reduction. However, in the important field of control resynthesis (Section 2.3.3) area consumption does not really matter, but performance. It is crucial to obtain controllers that react as fast as possible. Therefore, factors like cycle time must guide both the partition construction and the decomposition. In [CC08] some preliminary work in this direction can be found.

[5] Here, 2 out of 4 is already considered a majority, because the distinct partitioning criteria are rather orthogonal.

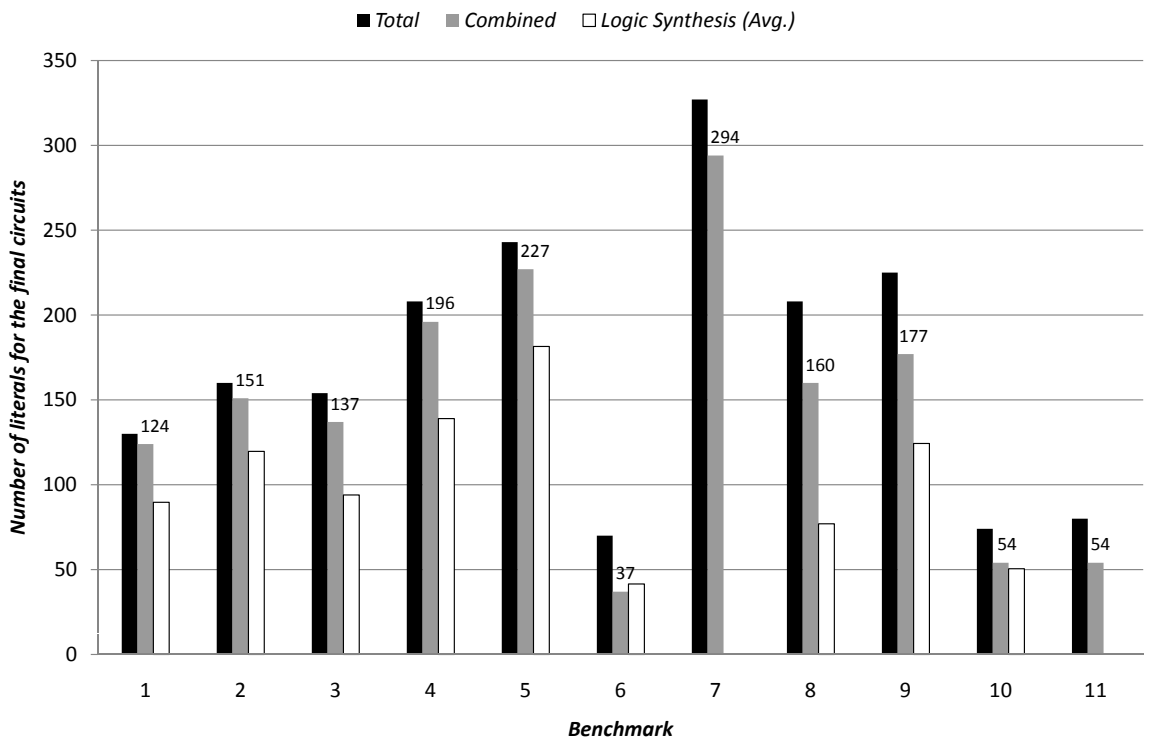


Figure 5.4: Comparison of area reduction results when applying total decomposition, decomposition using the combined partitioning heuristic, and logic synthesis for the benchmarks 1 to 11 from Section 5.5.

The grey column is labelled with the precise values for the combined partitioning heuristic results. Recall, the precise values for the application of the different logic synthesis methods or total decomposition can be found above in the tables 5.1 or 5.2 resp.

Concerning the partition construction, e.g. the 'locked components' heuristic may increase the cycle time if many signals are locked. Thus, when using the combined heuristic the r-locked criterion must not have such a big impact as the other compatibility criteria or it can even be omitted for the matter of performance improvement.

Furthermore, one can develop new basic heuristics for improving performance, e.g. a heuristic which reduces the amount of internal communication between the components such that the components are able to work as independent as possible (i.e. as parallel as possible); in particular, this avoids slow communication paths, e.g. starting from C_1 to C_2 ... to C_n. However, such a performance-driven heuristic might destroy CSC satisfaction though. Thus, it is crucial to find a sound combined heuristic which can be tuned to find a good trade-off.

6 The Decomposition Tool DesiJ

This chapter presents the decomposition tool DesiJ.[1] It is a *de*composer for *si*gnal transition graphs which is implemented in *J*ava. To cope with complexity in logic synthesis of asynchronous controllers, it decomposes a large STG specification into several smaller component STGs – as depicted by the architecture diagram in Figure 6.1(a), cf. also Figure 1.1(b).

After decomposition, SI logic synthesis from each component STG can be done by using Petrify or Punf&Mpsat. In Figure 2.12 it was already shown how DesiJ could be incorporated into a design-flow based on control resynthesis.

Decomposition with DesiJ is guided by an output partition and for each partition member a component STG is generated which should be SI implementable, i.e. it must not have irreducible CSC conflicts. Thus, DesiJ does not only consist of the *Decomposer* agent, see Figure 6.1(b), which implements several decomposition strategies [SVWK06, KSVW09], but also needs an *Irreducible CSC handler* to avoid irreducible CSC conflicts for the resulting component STGs. Chapter 4 describes in detail how the latter agent works, in particular how it modifies the specification and the former output partition (note that the Irreducible CSC handler has read/write-access – i.e. modifiable access – to a copy of the specification and to the output partition as well). The *Partitioning agent* computes suitable output partitions to improve the resulting circuits, as explained in Chapter 5.

Figure 6.2 shows a typical interaction of these agents to get SI implementable com-

[1] The content of this chapter is inspired by [SWW09].

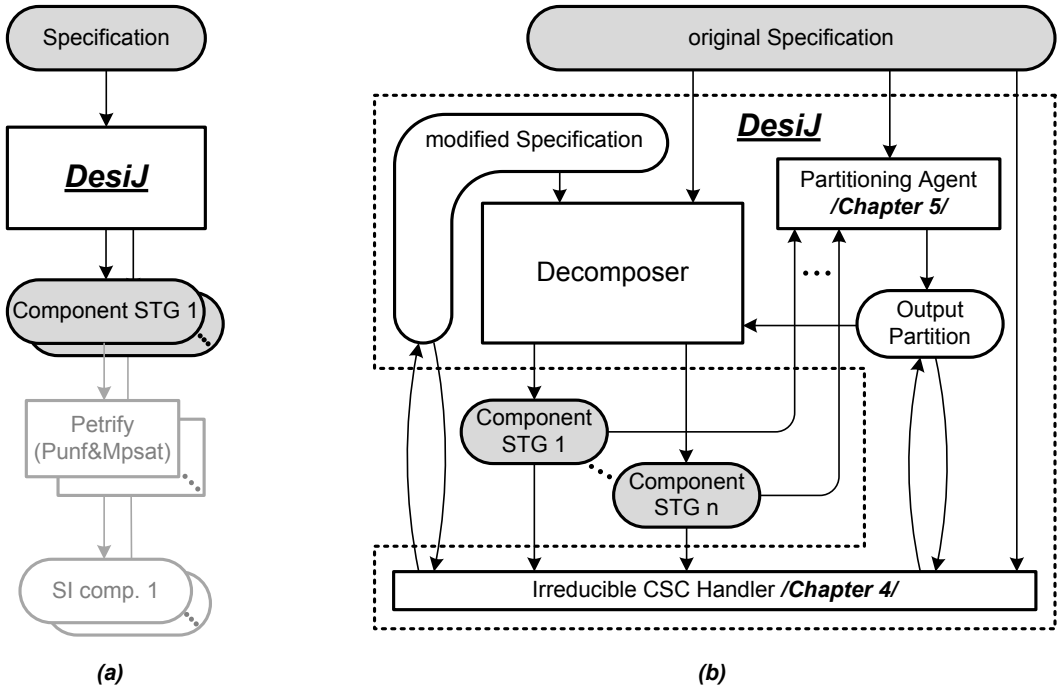


Figure 6.1: System Architecture of DesiJ.
(The grey storage components are realised as .g-files.)

ponent STGs. It is modelled as a safe Petri net, where transitions have operational characteristics, i.e. the firing of a transition represents the execution of the operation with which the considered transition is labelled. The small transitions without any label represent NOP-transitions, i.e. their firing represent *no op*eration; they are necessary due to syntactical reasons.

Before decomposition, the Partitioning agent computes an improved partition which is usually coarser than the finest one. Then the Decomposer computes the final components according to this chosen output partition. The Irreducible CSC handler checks for each component whether it has irreducible CSC conflicts. If no conflicts can be identified, the decomposition procedure is finished, otherwise the conflicts are tried to be avoided by introducing new internal communication signals and the modified specification is decomposed anew. Then one could try to identify new irreducible CSC conflicts for the modified components, resolve them, and decompose the modified specification again, until no new irreducible CSC conflicts can be identified.

However, the implemented approximation for identifying irreducible CSC conflicts of the components (which is represented by the grey-shaded transition) is not very mature yet – see also Section 4.6. On the one hand, there might be several conflicts identified as irreducible which are actually not such conflicts. On the other, there might be other

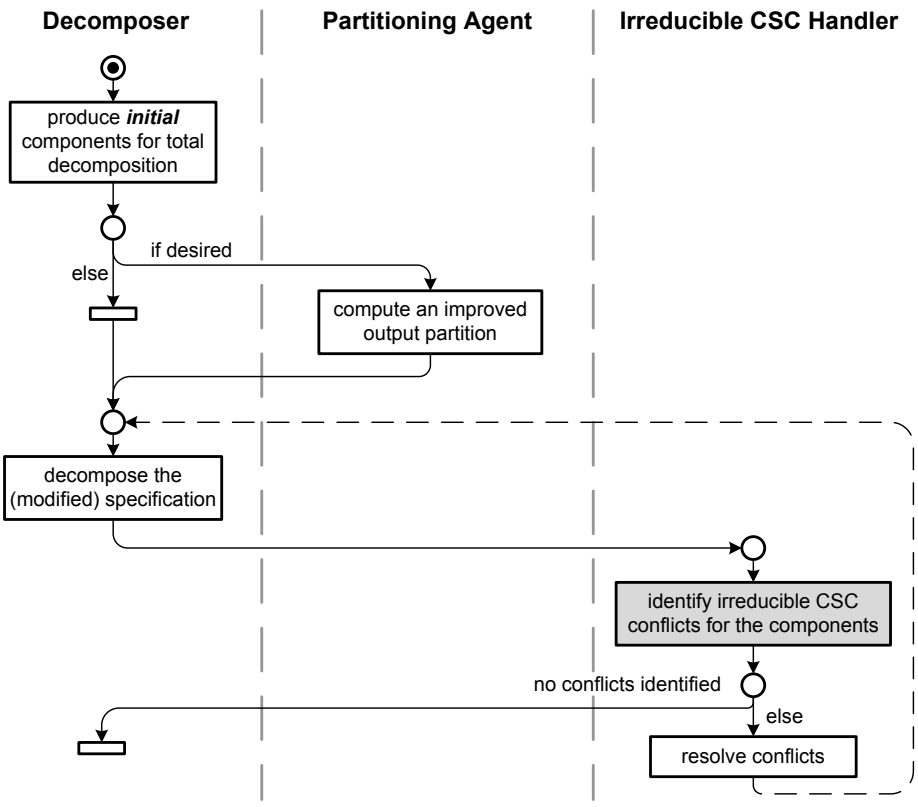


Figure 6.2: Typical Decomposition Sequence of DesiJ

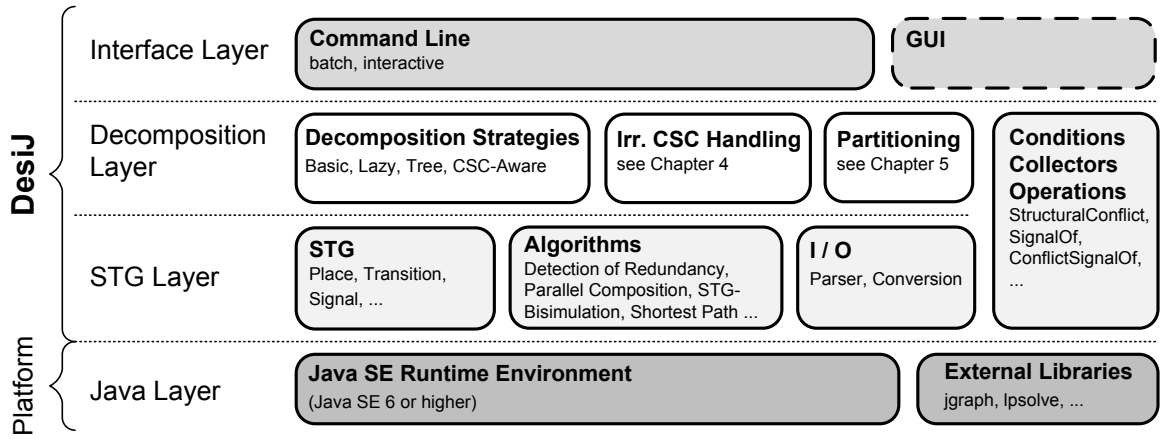


Figure 6.3: Layered Software Architecture of DesiJ (adapted from [SWW09])

'real' irreducible CSC conflicts which are not identifiable by the current implementation of the Irreducible CSC handler. Thus, the current DesiJ implementation executes this loop (represented by the dashed arc) only once; i.e. after the first run of decomposition it checks for irreducible conflicts, resolves them, applies a second decomposition pass and eventually tries to synthesise each component. This already succeeds for many benchmarks.

6.1 Software Architecture

DesiJ was started to develop in 2004 by Mark Schäfer as a decomposition framework for STGs [Sch07,SWW09]. The present author started to improve the tool in 2007 in order to enable decomposition-based *SI* logic synthesis. Java was chosen as the programming language, because a relatively short, easy extensible and maintainable source code was preferred over a high-performance implementation (e.g. using C/C++). In particular, performance was not the prime objective, because the implemented algorithms work structurally, i.e. it is explicitly avoided to perform expensive operations, e.g. building reachability graphs.

To support extensibility, DesiJ was developed as a layered architecture based on the Java SE Runtime Environment (JRE) and some external libraries, see Figure 6.3:

- The STG layer provides basic functionality for STG handling independent from decomposition. In particular, it implements the STG class and a read operation for STGs from Petrify '.g-files' by using the JavaCC parser generator [Jcc]. In order to view particular STGs conversion operations using Graphviz [Gra] are implemented.

To avoid generating many copies of component STGs during backtracking an *undo mechanism* is implemented which can restore previous versions of an STG very efficiently. It works as a stack like in most text or image editors. 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.

- The decomposition layer implements the various decomposition strategies, see [KSVW09]. In particular, a heuristic to approximate optimal *decomposition trees* for the strategy TREE is implemented in order to avoid the $\mathcal{NP}$-complete calculation of optimal decomposition trees [KK01].

  Furthermore, the decomposition layer implements the avoidance of irreducible CSC conflicts for the components, see Chapter 4, and the various partitioning heuristics, see Chapter 5, as well.

- The interface layer actually provides three access possibilities for the implemented functionality: a batch command line mode as the most powerful mode with the most options, an interactive command line mode and a graphical user interface (GUI) in order to visually track the decomposition process. At the moment, the GUI is merely working in an older version of DESIJ, but not for the current implementation (hence, the GUI block is depicted with a dashed border in Figure 6.3). The batch mode of DESIJ has been designed in order to use the tool as a back-end by some other tool or framework (cf. Figure 2.12). For example, there is already an interface for using DESIJ within the framework WORKCRAFT [Wor] which handles interpreted graph models. The accessible features via the command line are presented in Section 6.2.

- The *C*onditions, *C*ollectors and *O*perations (*CCO*) block is something special: these operations are independent from decomposition, but provide useful features for it; however, the CCO block belongs to the STG layer on a functional level. The CCO concept was introduced to enable the fast (sometimes prototype) implementation of necessary algorithms. It allows to implement abstract pseudo-code like descriptions of an algorithm in a straightforward way [Sch07].

6.2 Functionality

As explained, the primary task of DESIJ is the decomposition of complex STGs into several smaller component STGs by applying structural techniques only. Furthermore,

DesiJ provides many additional *external* functions. Some of them are used during the decomposition process as well. These additional external functions are:

- deletion of redundant places,
- dummy transition contraction (i.e. λ-labelled transitions),
- reachability graph generation,
- a verification operation to find an STG-bisimulation between the specification and its components.

All functions are accessible via the command line interface. The desired function can be chosen via the command line parameter `operation`, e.g. `desij operation=reddel some_stg` activates the deletion of all redundant places of `some_stg`. (Note that on Windows command lines the '`=`' characters are usually interpreted and replaced by spaces before the command execution happens. To avoid this interpretation use quotation marks such as `operation"="reddel`.) If no function or operation resp. is specified the default function is STG decomposition. A list of all external available functions can be found in [Sch07].

In addition to the functions, several *options* and *parameters* are available to configure the STG decomposition process. There are options and parameters for:

1. controlling the decomposition process,
2. handling of redundant places,
3. supporting the synthesis of components.

The first group contains options and parameters to affect the decomposition itself. The user can specify the decomposition strategy, i.e. Basic, LazyBack, Tree, Aggregation or CSC-Aware [KS07]. If Tree is chosen, the user can also determine the method used to generate the decomposition tree. Furthermore, he can define output partitions which are different from the finest one, or the user can specify to apply one of the heuristics presented in Chapter 5. Additionally, he can even configure the order or the execution of several decomposition operations during the decomposition (e.g. postponing failed transition contractions or perform safe contractions only).

The second group includes options to configure the deletion of redundant places during the decomposition. DesiJ can check for easy special cases (e.g. a place with an empty postset is always redundant), loop-only places, duplicate places, shortcut

places and implicit places [STC98]; the latter needs support from external tools such as PUNF&MPSAT or LPSOLVE.

The last group contains options and parameters for the synthesis of component STGs. Since DESIJ is only a decomposition tool but not a logic synthesiser, it provides interfaces to call PUNF, MPSAT and PETRIFY for SI logic synthesis after decomposition – cf. Figure 6.1(a). This usually requires the avoidance of irreducible CSC conflicts for the components (Chapter 4).

Besides, DESIJ provides various further functions, options and parameters to provide more user-friendliness:

- the conversion of STGs into postscript files by using the GraphViz package [Gra],
- the automatic generation of special complex benchmark STGs (e.g. SPT-?, see Figure 4.4 and Table 4.1), and
- operations to affect the logging process.

All options and parameters are accessible via the command line interface. Options can be given in a long or in a short form. For instance, the option for allowing safeness preserving transition contractions can be given as `--safe-contractions` or shortly as `-f`. The short forms can be combined, i.e. instead of `-f -T`, just `-fT` can be used[2]. Furthermore, options have a default value, either true or false, which is switched by the presence of the option in the command line call. In contrast, parameters must always be given in the form `parameter=value`, cf. the parameter `operation` mentioned above.

The default values for the command line options and parameters are chosen such that in most cases only typing `desij some_stg` results in a fast and sufficient decomposition of `some_stg`. Note that only STGs specified in the *.g-file format*, i.e. the input format of PETRIFY and PUNF, are processable by DESIJ.

A complete list of all command line options and parameters, accompanied by short descriptions including their default values, will be printed on the standard output by `desij --help`.

6.3 Installation

The basic requirement for running DESIJ is a working installation of the Java SE Runtime Environment, version 6 or higher.

[2]The option `-T` affects the removal of redundant transitions during the decomposition process.

DesiJ as well as some STG examples can be downloaded from [Des]. There are two versions: an older one with a working GUI and a current development snapshot without a GUI. It comes within an archive that has to be extracted to a local folder. Besides the binary of DesiJ (a zip- or jar-archive resp.) and maybe another library for the old version (called `freehep` [Fre]), the folder contains two shell scripts: one for running DesiJ on Windows machines (a *batch* script) and the other one for Linux systems (a *bash* script). After adapting the environment variables `DESIJ_DIR`[3] and `PATH` to the absolute path of DesiJ's local folder, one can call it at the command line, e.g. by simply typing `desij --help`.

The interaction with other tools such as Petrify and Mpsat works for Linux and Windows installations, but the Windows version of Petrify does not work as good as the Linux version; thus, it is recommended to use DesiJ on Linux machines. Of course, DesiJ requires the correct installation of these external tools as well as the adaptation of the `PATH` variable to the folders containing the binaries of them.

[3] as done in the shell scripts

7 Conclusions

The STG decomposition of [Wol97,VW02,VK07] was significantly improved by enabling decomposition-based logic synthesis of SI circuits, using the decomposition tool DesiJ, cf. Figure 1.1(b). An SI circuit synthesis from very large 'real-life' STG specifications of a size not reached before is possible. The method presented here is an important step towards taking further advantage of the optimisation space that logic synthesis provides.

It was shown that irreducible CSC conflicts resulting from STG decomposition can often be avoided by introducing internal communication between the components, without modifying the overall behaviour. The presented approach is purely structural which enables SI logic synthesis from very complex STGs, in particular resulting from control resynthesis of handshake circuits.

Decomposing a specification with CSC might generate components without CSC or even with *irreducible* CSC conflicts; the presented method shows a way how to deal with the latter problem. Remarkably, the applied STG decomposition can handle specifications without CSC, which is an advantage over other decomposition methods (e.g. the one in [YM07]).

From another point of view, the presented STG decomposition method contributes to the CSC solution for very complex specifications in a different way than former approaches (such as [CC08]). CSC conflicts are not solved for the entire specification, but they are broken down to CSC solving for the smaller components. Thus, CSC solving is split into two steps: First, the specification is decomposed into several smaller

component STGs by avoiding irreducible CSC conflicts via new internal communication signals, as presented in Chapter 4. Then CSC is solved for each component STG in isolation by using well-known and optimised techniques [CKK+02], which would suffer from complexity issues if they were applied to the complex specification. This downscaling of the CSC encoding problem to smaller component STGs enables SI logic synthesis from *very complex* specifications.

Furthermore, heuristics are presented as a guidance to find better output partitions for the applied STG decomposition method. This is an important step to improve the quality of the resulting circuits. In particular, the presented partitioning heuristics aim at the improvement of the final component's CSC satisfaction as well as chip area reduction. Very efficient techniques have been used ensuring that the computation of the partition does not become a bottleneck for the entire synthesis process.

7.1 Future Work

Many suggestions for further research have been made during chapters 4 and 5 for the improvement of the proposed methods. In particular, both sections 4.6 and 5.6 give an outlook for future research for each approach presented in the respective chapter. These ideas should not be repeated here, but the interested reader is referred to these sections. Here, more general ideas for future work will merely be given which are particularly not related to the content of either chapter 4 or 5. The first idea is a general improvement for the presented STG decomposition method, while the others are related to control resynthesis (cf. Section 2.3.3).

Relevant Signal Detection

An important issue for STG decomposition is the detection of the relevant signals to produce the component's output signals. During backtracking, input signals are added to the initial partition member or component resp. in order to avoid non-determinisms (e.g. auto-conflicts, see Section 3.2.3). Sometimes, this backtracking-driven process might not lead to a minimal final set of relevant signals for a particular component, which might eventually be too large for an efficient logic synthesis. It might be better to approximate a complete and minimal set of relevant signals *a priori* which ideally makes backtracking unnecessary; it should merely be used in exceptional cases. Recall that a complete set of relevant signals consists of the trigger signals of the component's outputs as well as all signals which are in conflict with them and maybe some *additional* signals which are added during backtracking.[1] In order to determine some

[1] Details can be found in Section 3.2.3.

small set of *really necessary* additional signals a priori, lock relations could be used as a guidance. According to [VGCM92], two signals of a consistent STG are called *locked* if the occurrences of their corresponding signal edges always alternate. Lock relations are used to introduce constraints among the signals of an STG to resolve CSC conflicts (see e.g. [CC08]). For efficiency, the lock relation w.r.t. all signal pairs of some (complex) specification N can be approximated by using LP techniques (instead of ILP), as explained in Section 5.4. Furthermore, lock classes can efficiently be determined using graph algorithms on the lock graph G of N (see also [VGCM92, YP90]); G's vertices represent signals of N and its edge set is determined by the (approximated) lock relation.

An important field for applying decomposition-based logic synthesis is control resynthesis of handshake circuits. As explained in Section 2.3.3, HC-STGs are composed by using the conventional parallel composition (cf. also Figure 2.10) with inter-component communication signals being hidden and contracted (cf. also Figure 2.11), resulting in a so-called cluster STG. For such (usually large) cluster STGs decomposition-based logic synthesis should be applied to get an efficient cluster circuit implementation, in terms of chip area, performance and/or energy consumption.

The following suggestions for future work are particularly related to control resynthesis.

Improving Parallel Composition

The application of conventional parallel composition (as defined in Section 3.2.2) to the HC-STGs almost always produces 'messy' cluster STGs with many implicit places, even if the HC-STGs did not have them. Some of these places are easy to remove by using efficient structural methods, e.g. duplicate places (which have the same pre- and postsets), but removing other implicit places usually requires full-blown model checking, which is infeasible for very large cluster STGs. Although implicit places do not have noticeable effect on tools based on state space exploration, such as Petrify, the results from tools that are based on structural methods, such as DesiJ, often deteriorate. An operation where implicit places matter is transition contraction (as defined in Definition 3.2.1), which is a crucial operation during STG decomposition. Implicit places in the preset and/or postset of a transition can prevent its contraction, although a contraction would be possible after removing these places.

The structure of the cluster STG can be optimised by improving the standard parallel composition such that the composed STG structures are better suited for applying structural operations, e.g. transition contraction. The idea is to add a pre-processing

step to the conventional parallel composition by removing places from the components that are guaranteed to be implicit in the result. The precise conditions that allow to remove a particular place from a component can be found in [AKM+11]; at this point it is enough to know that they are structural and thus can efficiently be checked. Finally, the resulting net usually contains less implicit places in comparison with the net obtained by conventional parallel composition.

Subnet Contraction

Another aspect for the improvement of the results might be the application of subnet contraction (instead of transition contractions only). Since the building of cluster STGs follows a strict set of rules (i.e. parallel composition and inter-component signal contraction) applied to a small set of possible HC-STGs[2], the resulting cluster STG structures might not be totally arbitrary. It might be a reasonable assumption that one can identify (maybe 'messy') recurring substructures in these cluster STGs. If such an entire substructure becomes irrelevant for some component, then its contraction as a whole could be useful to get a small final component; in particular, if an inappropriate contraction order for the transitions of this substructure prevents some of these contractions.

Obviously, subnet contraction is not only related to control resynthesis, but it would be a general improvement for the presented STG decomposition. A first simple example for subnet contraction was introduced in Definition 4.1.6 which enables the contraction of a dummy gyroscope.

Performance Improvements

In the field of control resynthesis, high performance really matters, i.e. controllers having a small cycle time and short communication paths between their components are preferred over circuits which are e.g. area efficient. Consequently, partitioning could be guided to improve performance, as already sketched in Section 5.6. Furthermore, the decomposition strategy itself might be improved in this direction:

As mentioned, backtracking during decomposition is crucial to dissolve non-determinisms by declaring a former irrelevant signal as relevant during the reduction process for some component. If non-determinisms, such as auto-conflicts, occur, then there are usually options between two or more signals to declare one of them as relevant. Thus, one could prefer to choose an input from the environment instead of an output of the specification, which is produced by another component. Note that the choice of an output might introduce long communication paths through several components and

[2] approximately 10 control components

hence slows down the performance of the resulting system.

Another way to influence the decomposition strategy w.r.t. performance gain concerns the reordering of applied transition contractions, cf. also [SVWK06]. For some (initial) component a list of irrelevant signals, which have to be contracted (or their corresponding transitions resp.), could be reordered, such that those signals have to be contracted first which would slow down the final component if they were backtracked.

The clustering of a handshake circuit, as explained in Section 2.3.3, could also be improved w.r.t. performance gain. In particular, the control component clusters (see Figure 2.9) should be chosen along the longest (i.e. slowest) signal paths. By this way, decomposition-based logic synthesis or even pure logic synthesis definitely tackles the performance bottlenecks of the system's control path, especially when implementing the resulting cluster circuits in fast 2-level logic.

Bibliography

[Ach] Achronix. URI: http://www.achronix.com.

[AKM+11] Arseniy Alekseyev, Victor Khomenko, Andrey Mokhov, Dominic Wist, and Alex Yakovlev. Improved parallel composition of labelled Petri nets. In *ACSD'11: Proceedings of the 11th International Conference on Application of Concurrency to System Design*, 2011. (to appear).

[AN07] Melinda Y. Agyekum and Steven M. Nowick. A cycle-based decomposition method for burst-mode asynchronous controllers. In *In ASYNC'07: Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 129–142, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[And83] Charles André. Structural transformations giving b-equivalent PT-nets. In *Selected Papers from the 3rd European Workshop on Applications and Theory of Petri Nets*, pages 14–28. Springer, 1983.

[Bal] Balsa – An Asynchronous Synthesis System. URI: http://apt.cs.man.ac.uk/projects/tools/balsa/.

[Bar98] Andrew Bardsley. Balsa: An asynchronous circuit synthesis system. Master's thesis, University of Manchester, Faculty of Science and Engineering, Department of Computer Science, 1998.

[Bar00] Andrew Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, University of Manchester, Faculty of Science and Engineering, School of Computer Science, 2000.

[BCK+04] Ivan Blunno, Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Kelvin Lwin, and Christos P. Sotiriou. Handshake protocols for de-synchronization. In *ASYNC'04: Proceedings of the 10th International*

Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 149–158. IEEE Computer Society Press, 2004.

[BE97] Andrew Bardsley and Doug Edwards. Compiling the language Balsa to delay insensitive hardware. In *CHDL'97: Proceedings of the IFIP TC10 /WG 10.5 international conference on hardware description languages and their applications : specification, modelling, verification and synthesis of microelectronic systems*, pages 89–91, London, UK, 1997. Chapman & Hall, Ltd.

[BEN04] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. GNU LGPL (Lesser General Public Licence), May 2004. URI: http://lpsolve.sourceforge.net /5.5/.

[Ber87] Gérard Berthelot. Transformations and decompositions of nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986-Part I: Central Models and Their Properties*, LNCS 254, pages 359–376, London, UK, 1987. Springer.

[BEW99] Jochen Beister, Gernot Eckstein, and Ralf Wollowski. From STG to extended-burst-mode machines. In *ASYNC'99: Proceedings of the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 145–158, Washington, DC, USA, 1999. IEEE Computer Society.

[BL00] Ivan Blunno and Luciano Lavagno. Automated synthesis of micro-pipelines from behavioral verilog HDL. In *ASYNC'00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 84–92, Washington, DC, USA, 2000. IEEE Computer Society.

[BSKY04] Frank Burns, Delong Shang, Albert Koelmans, and Alex Yakovlev. An asynchronous synthesis toolset using verilog. In *DATE'04: Proceedings of the conference on Design, automation and test in Europe*, pages 724–725, Washington, DC, USA, 2004. IEEE Computer Society.

[BTE09] Andrew Bardsley, Luis Tarazona, and Doug Edwards. Teak: A token-flow implementation for the balsa language. In *ACSD'09: Proceedings*

of the 9th International Conference on Application of Concurrency to System Design, pages 23–31. IEEE Computer Society, 2009.

[BY02] Alexandre Bystrov and Alexandre Yakovlev. Asynchronous circuit synthesis by direct mapping: Interfacing to environment. In *ASYNC'02: Proceedings of the 8th International Symposium on Asynchronus Circuits and Systems*, pages 127–136. IEEE, 2002.

[CC03] Josep Carmona and Jordi Cortadella. ILP models for the synthesis of asynchronous control circuits. In *ICCAD'03: Proceedings of the IEEE/ACM international conference on Computer-aided design*, Washington, DC, USA, 2003.

[CC06] Josep Carmona and Jordi Cortadella. State encoding of large asynchronous controllers. In *DAC'06: Proceedings of the 43rd annual conference on Design automation*, pages 939–944, New York, USA, 2006.

[CC08] Josep Carmona and Jordi Cortadella. Encoding large asynchronous controllers with ILP techniques. *Proceedings of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(1):20–33, 2008.

[CCCGV06] Josep Carmona, José Manuel Colom, Jordi Cortadella, and F. García-Vallés. Synthesis of asynchronous controllers using integer linear programming. *Proceedings of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(9):1637–1651, 2006.

[CEP95] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. *Theor. Comput. Sci.*, 147(1-2):117–136, 1995.

[Chu87] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[CKK+97] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.

[CKK+02] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Springer, ISBN: 3-540-43152-7, 2002.

[CKLS06] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, and Christos P. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1904–1921, 2006.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition edition, 2001.

[CNBE02] Tiberiu Chelcea, Steven M. Nowick, Andrew Bardsley, and Doug Edwards. A burst-mode oriented back-end for the balsa synthesis system. In *DATE'02: Proceedings of the conference on Design, automation and test in Europe*, pages 330–337, Washington, DC, USA, 2002.

[CNBE03] Tiberiu Chelcea, Steven M. Nowick, Andrew Bardsley, and Doug Edwards. Balsa-cube: an optimising back-end for the balsa synthesis system. In *Proceedings of 14th UK Async. Forum*, 2003.

[Des] DesiJ: A Tool for STG Decomposition. URI: http://www.informatik.uni-augsburg.de/en/chairs/swt/ti/research/tools/desij.

[DN97] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical report, Technical Report UUCS-97-013, Computer Science Department, University of Utah, 1997.

[EB02] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[Ebe92] Jo C. Ebergen. Arbiters: an exercise in specifying and decomposing asynchronously communicating components. *Sci. Comput. Program.*, 18(3):223–245, 1992.

[EBJ+06] Doug Edwards, Andrew Bardsley, Lilian Janin, Luis Plana, and Will Toms. *Balsa: A Tutorial Guide*, version 3.5 edition, 2006.

[Ela] Elastix. URI: http://elastixcorp.com.

[EM00] Javier Esparza and Stephan Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods in System Design*, 16:159–189, 2000.

[ERV96] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. In *TACAs'96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 87–106, London, UK, 1996. Springer.

[Esp98] Javier Esparza. Decidability and complexity of Petri net problems — An introduction. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, LNCS 1491, pages 374–428. Springer, 1998.

[FC08] Francisco Fernández and Josep Carmona. Logic synthesis of handshake components using structural clustering techniques. In *PATMOS 2008: Proceedings of the 18th International Workshop on Power and Timing Modeling, Optimization and Simulation*, LNCS 5349, pages 188–198, 2008.

[FNT+99] Robert M. Fuhrer, Steven M. Nowick, Michael Theobald, Niraj K. Jha, Bill Lin, and Luis Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Computer Science Dept., Columbia University, 1999.

[Fre] FreeHEP Java Libraries. URI: http://java.freehep.org/.

[Gei85] Jürgen Geissler. *Zerlegung von diskreten Systemen mit Petrinetzen*. PhD thesis, University of Kaiserslautern, Electrical Engineering, 1985.

[Gra] Graphviz – Graph Visualization Software. URI: http://www.graphviz.org/.

[Gre] GreenArrays, Inc. URI: http://greenarraychips.com.

[GVC97] Fernando García-Vallés and José-Manuel Colom. Structural analysis of signal transition graphs. In *PNSE'97: Proceedings of the Workshop on Petri Nets in System Engineering*, Report FBI-HH-B-205, pages 123–134, 1997.

[HBB95] Henrik Hulgaard, Steven M. Burns, and Gaetano Borriello. Testing asynchronous circuits: A survey. *Integration, The VLSI Journal*, 19:111–131, 1995.

[Him99] Peter Himmighöfer. Module zur Schrittgraphgenerierung und Dekomposition des Signalflankengraphen für ein CAD-System zur Synthese asynchroner Schaltwerksverbünde. Master's thesis, Fachbereich EIT, University of Kaiserslautern, 1999.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 3 edition, 2006.

[Hoa85] Tony Hoare. *Communicating sequential processes.* Prentice/Hall International, April 1985.

[Int] intellaSys. URI: http://www.intellasys.net.

[Jcc] Java Compiler Compiler (JavaCC) – The Java Parser Generator. URI: http://javacc.java.net/.

[KE96] Andrei Kovalyov and Javier Esparza. A polynomial algorithm to compute the concurrency relation of free-choice signal transition graphs. In *Proc. of the International Workshop on Discrete Event Systems (WODES)*, pages 1–6, UK, 1996. IEE.

[KGT06] Andreas Knöpfel, Bernhard Gröne, and Peter Tabeling. *Fundamental Modeling Concepts: Effective Communication of IT Systems.* Wiley, 2006.

[KK01] Victor Khomenko and Maciej Koutny. Towards an efficient algorithm for unfolding Petri nets. In *CONCUR'01: Proceedings of the 12th International Conference on Concurrency Theory*, LNCS 2154, pages 366–380. Springer, 2001.

[KKT93] Alex Kondratyev, Michael Kishinevsky, and Alexander Taubin. Synthesis method in self-timed design. Decompositional approach. In *ICVC'93: Proceedings of the IEEE International Conference on VLSI and CAD*, pages 324–327, 1993.

[KKY04] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Logic synthesis for asynchronous circuits based on Petri net unfoldings and incremental SAT. In *ACSD'04: Proceedings of 4th International Conference on Application of Concurrency to System Design*, pages 16–25, Hamilton, Ontario, Canada, 2004. IEEE.

[KMY06] Victor Khomenko, Agnes Madalinski, and Alex Yakovlev. Resolution of encoding conflicts by signal insertion and concurrency reduction based on STG unfoldings. In *ACSD'06: Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 57–68, Washington, DC, USA, 2006. IEEE.

[KNI08] Nobuo Karaki, Takashi Nanmoto, and Satoshi Inoue. An asynchronous circuit design technique for a flexible 8-bit microprocessor. *IEICE Transactions*, 91-C(5):721–730, 2008.

[KS07] Victor Khomenko and Mark Schäfer. Combining decomposition and unfolding for STG synthesis. In *ATPN'07: Proceedings of the 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, LNCS 4546, pages 223–243, 2007.

[KSVW09] Victor Khomenko, Mark Schäfer, Walter Vogler, and Ralf Wollowski. STG decomposition strategies in combination with unfolding. *Acta Informatica*, 46:433–474, September 2009.

[KVL96] Tilman Kolks, Steven Vercauteren, and Bill Lin. Control resynthesis for control-dominated asynchronous designs. In *ASYNC'96: Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 233–243, Washington, DC, USA, 1996. IEEE Computer Society.

[KWVB03] Ben Kangsah, Ralf Wollowski, Walter Vogler, and Jochen Beister. DESI: a tool for decomposing signal transition graphs. In *3rd ACiD-WG Workshop*, Crete, 2003.

[Man06] Rajit Manohar. Reconfigurable asynchronous logic. In *CICC'06: Custom Integrated Circuits Conference*, pages 13–20, 2006.

[Mar89] Alain J. Martin. Programming in vlsi: From communicating processes to

delay-insensitive circuits. Technical report, California Institute of Technology, Pasadena, CA, USA, 1989.

[Mar90] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the 6th MIT conference on Advanced research in VLSI*, pages 263–278. MIT Press, 1990.

[MB59] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. *The Annals of the Computation Laboratory of Harvard University*, 29:204–243, 1959.

[McM93] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV'92: Proceedings of the 4th International Workshop on Computer Aided Verification*, pages 164–177, London, UK, 1993. Springer.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MLM+97] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous mips r3000 microprocessor. In *Conference on Advanced Research in VLSI*, pages 164–181, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[MLM99] Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. Projection: A synthesis technique for concurrent systems. In *ASYNC'99: Proceedings of the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society, 1999.

[MM98] Rajit Manohar and Alain J. Martin. Slack elasticity in concurrent computing. In *MPC'98: Proceedings of the Mathematics of Program Construction*, LNCS 1422, pages 272–285, London, UK, 1998. Springer.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[Now93] Steven Mark Nowick. *Automatic Synthesis of Burst-mode Asynchronous Controllers*. PhD thesis, Computer Systems Laboratory Departments of Electrical Engineering and Computer Science at the Stanford University, 1993.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[PKY09] Ivan Poliakov, Victor Khomenko, and Alex Yakovlev. Workcraft — a framework for interpreted graph models. In *ATPN'09: Proceedings of the 30th International Conference on Applications and Theory of Petri Nets*, LNCS 5606, pages 333–342, 2009.

[Poh80] Walter Pohl. *Petrinetz-Modelle der Dynamik diskreter, technischer Systeme.* PhD thesis, University of Kaiserslautern, Electrical Engineering, 1980.

[PtBdWM10] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An implementation style for data-driven compilation. In *ASYNC'10: Proceedings of the 16th International Symposium on Asynchronous Circuits and Systems*, pages 3–14. IEEE Computer Society, 2010.

[PTE05] Luis A. Plana, Sam Taylor, and Doug Edwards. Attacking control overhead to improve synthesised asynchronous circuit performance. In *ICCD'05: Proceedings of the 2005 International Conference on Computer Design*, pages 703–710, Washington, DC, USA, 2005. IEEE Computer Society.

[Rei85] Wolfgang Reisig. *Petri Nets.* EATCS Monographs on Theoretical Computer Science 4. Springer, 1985.

[Rei10] Wolfang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien.* Leitfäden der Informatik. Vieweg+Teubner, 15 July 2010. ISBN 978-3-8348-1290-2.

[RY85] Leonid Ya. Rosenblum and Alexandre Yakovlev. Signal graphs: From self-timed to timed ones. In *International Workshop on Timed Petri Nets*, pages 199–206, Torino, Italy, 1985.

[SBY03] Danil Sokolov, Alex Bystrov, and Alex Yakovlev. STG optimisation in the direct mapping of asynchronous circuits. In *DATE'03: Proceedings of the conference on Design, Automation and Test in Europe*, pages 932–937, Washington, DC, USA, 2003. IEEE Computer Society.

[Sch07] Mark Schäfer. Desij – a tool for STG decomposition. Technical Report 2007-11, Institute of Computer Science, University of Augsburg, 2007.

[Sch08] Mark Schäfer. *Advanced STG Decomposition*. PhD thesis, University of Augsburg, 2008.

[SKC+99] Hiroshi Saito, Alex Kondratyev, Jordi Cortadella, Luciano Lavagno, and Alexander Yakovlev. What is the cost of delay insensitivity? In *ICCAD'99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 316–323, Piscataway, NJ, USA, 1999. IEEE Press.

[Spa01] Jens Sparsø. Asynchronous circuit design – a tutorial. In *Chapters 1-8 in "Principles of asynchronous circuit design - A systems Perspective"*, pages 1–152. Kluwer Academic Publishers, Boston / Dordrecht / London, 2001.

[Sta90] Peter H. Starke. *Analyse von Petri-Netz-Modellen*. Teubner, 1990.

[STC98] Manuel Silva, Enrique Teruel, and José Manuel Colom. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. *Lectures on Petri Nets I: Basic Models*, LNCS 1491:309–373, 1998.

[SV07] Mark Schäfer and Walter Vogler. Component refinement and CSC solving for STG decomposition. *Theoretical Computer Science*, 388(1–3):243–266, 2007.

[SVJ05] Mark Schäfer, Walter Vogler, and Petr Jančar. Determinate STG decomposition of marked graphs. In G. Ciardo and P. Darondeau, editors, *ATPN'05: In Proceedings of the 26th International Conference On Application and Theory of Petri Nets and Other Models of Concurrency*, LNCS 3536, pages 365–384. Springer, 2005.

[SVWK06] Mark Schäfer, Walter Vogler, Ralf Wollowski, and Victor Khomenko. Strategies for optimised STG decomposition. In *ACSD'06: Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 123–132, Washington (D.C.), USA, 2006. IEEE.

[SWW09] Mark Schäfer, Dominic Wist, and Ralf Wollowski. Desij – enabling decomposition-based synthesis of complex asynchronous controllers. In

ACSD'09: Proceedings of the 9th International Conference on Application of Concurrency to System Design, pages 186–190, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[SY05] Danil Sokolov and Alex Yakovlev. Clockless circuits and system synthesis. *IEE Proceedings – Computers & Digital Techniques*, 152(3):298–316, 2005.

[Tay07] Samuel M. Taylor. *Data-Driven Handshake Circuit Synthesis*. PhD thesis, University of Manchester, Faculty of Engineering and Physical Sciences, School of Computer Science, 2007.

[TEP08] Sam Taylor, Doug Edwards, and Luis Plana. Automatic compilation of data-driven circuits. In *ASYNC'08: 14th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 3–14, 2008.

[Ung69] Stephen H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, New York, USA, 1969.

[Val98] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, LNCS 1491, pages 429–528, London, UK, 1998. Springer.

[vB93] Kees van Berkel. *Handshake circuits: an asynchronous architecture for VLSI programming*. Cambridge University Press, New York, NY, USA, 1993.

[vBJN99] C. H. Kees van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, 1999.

[vBKR+91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *EURO-DAC'91: Proceedings of the conference on European design automation*, pages 384–389, Los Alamitos, CA, USA, 1991.

[VGCM92] Peter Vanbekbergen, Gert Goossens, Francky Catthoor, and Hugo J. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. *Proceedings of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(11):1426–1438, 1992.

[vGvBP+98] Hans van Gageldonk, Kees van Berkel, Ad Peeters, Daniel Baumann, Daniel Gloor, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. In *ASYNC'98: Proceedings of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, Washington, DC, USA, 1998.

[VK07] Walter Vogler and Ben Kangsah. Improved decomposition of signal transition graphs. *Fundamenta Informaticae*, 78:161–197, 2007.

[VW02] Walter Vogler and Ralf Wollowski. Decomposition in asynchronous circuit design. In *Concurrency and Hardware Design, LNCS 2549*, pages 152–190. Springer, 2002.

[WB00] Ralf Wollowski and Jochen Beister. Comprehensive causal specification of asynchronous controller and arbiter behaviour. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 3–32. Kluwer Academic Publishers, 2000.

[Wen77] Siegfried Wendt. Using Petri nets in the design process for interacting asynchronous sequential circuits. In *Proceedings of the IFAC-Symposium on Discrete Systems*, volume 2 of *Dresden*, pages 130–138, 1977.

[WM01] Catherine G. Wong and Alain J. Martin. Data-driven process decomposition for circuit synthesis. In *ICECS 2001: Proc. of the IEEE Conference on Electronic Circuits and Systems*, 2001.

[WM03] Catherine G. Wong and Alain J. Martin. High-level synthesis of asynchronous systems by data-driven decomposition. In *DAC'03: Proceedings of the 40th conference on Design automation*, pages 508–513, New York, NY, USA, 2003. ACM.

[Wol97] Ralf Wollowski. *Entwurfsorientierte Petrinetz-Modellierung des Schnittstellen-Sollverhaltens asynchroner Schaltwerksverbünde*. PhD thesis, University of Kaiserslautern, Electrical Engineering, 1997.

[Wor] Workcraft – A Framework for Interpreted Graph Models. URI: http://workcraft.org/.

[WSVW10] Dominic Wist, Mark Schäfer, Walter Vogler, and Ralf Wollowski. STG decomposition: Internal communication for SI implementability. Tech-

nical Report 32, HPI, University of Potsdam, 2010. URI: http://www.hpi.uni-potsdam.de/forschung/publikationen/technische_berichte.html.

[WSVW11] Dominic Wist, Mark Schäfer, Walter Vogler, and Ralf Wollowski. Signal transition graph decomposition: Internal communication for speed independent circuit implementation. *IET Computers & Digital Techniques*, 5:440–451, 2011.

[WV09] Ralf Wollowski and Walter Vogler. OptaCon: Optimierung hochsprachen-spezifizierter asynchroner Controller mittels STG-Dekomposition. Proposal for the DFG (Deutsche Forschungsgemeinschaft), May 2009. GZ: VO 615/10-1 and WO 814/3-1.

[WVW11] Dominic Wist, Walter Vogler, and Ralf Wollowski. STG decomposition: Partitioning heuristics. In *ACSD'11: Proceedings of the 11th International Conference on Application of Concurrency to System Design*, pages 141–150, 2011.

[WWSV09] Dominic Wist, Ralf Wollowski, Mark Schäfer, and Walter Vogler. Avoiding irreducible CSC conflicts by internal communication. *Fundamenta Informaticae*, 95(1):1–29, 2009.

[YD99a] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits. II.(automatic synthesis). *Proceedings of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(2):118–132, 1999.

[YD99b] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits. I.(specification and hazard-free implementations). *Proceedings of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(2):101–117, 1999.

[YK98] Alexandre Yakovlev and Albert Koelmans. Petri nets and digital hardware design. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets II: Applications*, volume 1492 of *LNCS*, pages 154–236. Springer Berlin / Heidelberg, 1998.

[YM07] Tomohiro Yoneda and Chris J. Myers. Synthesis of timed circuits based on decomposition. *Proceedings of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(7):1177–1195, 2007.

[YMKM05] Tomohiro Yoneda, Atsushi Matsumoto, Manabu Kato, and Chris Myers. High level synthesis of timed asynchronous circuits. In *ASYNC'05: Proceedings of the 11th International Symposium on Asynchronous Circuits and Systems*, pages 178–189, 2005.

[YOM04] Tomohiro Yoneda, Hiroomi Onda, and Chris Myers. Synthesis of speed independent circuits based on decomposition. In *ASYNC'04: Proceedings of the 10th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 135–145, Los Alamitos, CA, USA, 2004.

[YP90] Alexandre Yakovlev and Alexei Petrov. Petri nets and asynchronous bus controller design. In *ICATPN'90: International Conference on Application and Theory of Petri Nets*, LNCS 524, pages 244–262, Paris, 1990. Springer.