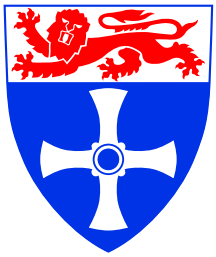
School of Electrical, Electronic & Computer Engineering

UNIVERSITY OF
NEWCASTLE UPON TYNE



Direct mapping of low-latency asynchronous controllers from STGs

D.Sokolov, A.Bystrov, A.Yakovlev

Technical Report Series

NCL-EECE-MSD-TR-2006-110

January 2006

Contact:
danil.sokolov@ncl.ac.uk
a.bystrov@ncl.ac.uk
alex.yakovlev@ncl.ac.uk

EPSRC supports this work via GR/S12036 (STELLA) and GR/S81421 (SCREEN).

NCL-EECE-MSD-TR-2006-110

School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

Direct mapping of low-latency asynchronous controllers from STGs

D.Sokolov, A.Bystrov, A.Yakovlev

January 2006

Abstract

A method for automated synthesis of low latency asynchronous controllers using direct mapping is presented. The idea of direct mapping is that a graph specification of a system is translated into a circuit netlist by mapping the graph nodes into circuit elements and the graph arcs into circuit interconnects. The key feature of this approach is its low algorithmic complexity and direct correspondence between the elements of the initial specification and the components of the resultant circuit. In out method the synthesis starts from an initial specification in form of a Signal Transition Graph (STG). This STG is split into a device and an environment, which synchronise via a communication net that models wires. The device is represented as a tracker and a bouncer. The tracker follows the state of the environment and provides reference points to the device outputs. The bouncer interfaces to the environment and generates output events in response to the input events according to the state of the tracker. This two-level architecture provides an efficient interface to the environment and is convenient for subsequent mapping into a circuit netlist. A set of optimisation heuristics are developed to reduce the latency and size of the control circuit. As a result of this work, a software tool called OptiMist has been developed. Its low algorithmic complexity allows large specifications to be synthesised, which is not possible in acceptable time for the tools based on state-space exploration. OptiMist successfully interfaces conventional EDA design flow for simulation, timing analysis and place-and-route.

1 Introduction

Two main approaches to design of asynchronous controllers are logic synthesis [7] and direct mapping [15, 18].

Logic synthesis works with the low-level system specifications which capture the behaviour of the system at the level of signal transitions. In this approach boolean equations for the output signals of the circuit are derived using the *next state functions* [5]. In order to find the next state functions all possible orders of the events must be explored. Such an exploration may result in a state space which exponentially large w.r.t. the initial specification. The circuit optimisation often involves analysis and recalculation of the whole state space.

The logic synthesis approach is now well developed and supported by public tools (Petrify [7], Minimalist [12], 3D [4]). However, this approach suffers from excessive computation complexity and memory requirements, thus it cannot be applied to large specifications. There is no transparent correspondence between the elements of the original specification, the intermediate representation of the state space and the components of the resultant circuit, which complicates the checking of circuit functionality.

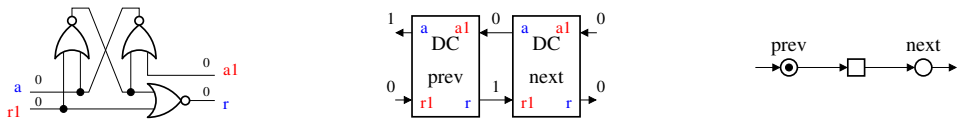
The main idea of the *direct mapping* approach is that a graph specification of a system is translated into a circuit netlist in such a way that the graph nodes correspond to the circuit elements and graph arcs correspond to the interconnects. Direct mapping can typically be divided into three independent operations: *translation*, *optimisation* and *mapping*. Firstly, a system specification is translated into an intermediate graph representation convenient for subsequent mapping. Then, peephole optimisation is usually applied to the intermediate representation of a system. Finally, the optimised graph is mapped into a circuit netlist implementation. In a practical design flow, however, some operations can be merged together or not present at all, e.g. optimisation is often performed together with mapping and there are cases when the circuit implementation is obtained directly from the initial specification without converting it into an intermediate form.

The key feature of the direct mapping approach is its low algorithmic complexity. The use of heuristic-based local optimisation (as opposed to state-space global optimisation in a logic synthesis approach) also facilitates the computational simplicity of the method. The transparent correspondence between the elements of the initial specification and the components of the resultant circuit is advantageous for checking the functional correctness of the implementation. Notwithstanding all advantages, this approach is insufficiently studied and existing techniques for direct mapping often produce large circuits with inefficient interface to the environment.

The direct mapping approach originates from [16], where a method of *the one-relay-per-row* realisation of an asynchronous sequential circuit is proposed. This approach is further developed in [36] where the idea of the *1-hot state assignment* is described. The 1-hot state assignment is then used in the method of concurrent circuit synthesis presented in [15]. The underlying model in this method is an Augmented Finite State Machine (AFSM), which is an FSM with added facilities, including timing mechanisms for the delay of state changes. These circuits have inputs that are logic values (signal levels as opposed to signal transitions), which is advantageous for low-level interfacing. These circuits use a separate set-reset flip-flop for every local state, which is set to 1 during a transition into the state, and which in turn resets to 0 the flip-flops of all its predecessor's local states. The main disadvantages of this approach are the fundamental mode assumptions and the use of local state variables as outputs. The latter are convenient for implementing event flows but require an additional level of flip-flops if each of those events controls just one switching phase of an external signal (either from 0 to 1 or from 1 to 0).

Another direct mapping method proposed in [28] works for the whole class of 1-safe Petri nets. However, it produces control circuits whose operation uses a 2-phase (no-return-to-zero) signalling protocol. This results in lower performance than what can be achieved in 4-phase circuits.

The approach of [18] is based on *distributors* and also uses the 1-hot state assignment, though a different implementation of local states. In this method every place of a Petri net is associated with a *David cell* (DC) [9]. The circuit diagram of a single DC is shown in Figure 1(a). The state of its output r denotes the marking of an associated Petri net place. DCs can be coupled using a 4-phase handshake protocol, so that the interface $\langle a1,\ r\rangle$ of the previous stage DC is connected to the interface $\langle a,\ r1\rangle$ of the next stage as shown in Figure 1(b). This DC structure corresponds to a Petri net shown in Figure 1(c). The circuits built of DCs by this approach are speed independent [24] and do not need fundamental mode assumptions. On the other hand, these circuits are autonomous (no inputs/outputs). The only way of interfacing them to the environment is to represent each interface signal as a set of abstract processes, implemented as request-acknowledgement handshakes, and to insert these handshakes into the breaks in the wires connecting DCs. This restricts the use of DCs to high-level design.



(a) Circuit (b) Coupling (c) Model

Figure 1: David cell

The controllers and interfaces are traditionally specified by timing diagrams and STGs. However, the majority of direct mapping techniques work with high-level Petri nets and cannot process low-level specifications. An attempt to apply direct mapping method at a low-level, where the circuit behaviour is captured at the level of signal events, is made in [37]. In this approach DC structures are used to capture the state of the system and to control flip-flops which are associated to each output signal. Inputs, however, are still represented as abstract processes and free-choice nets are not supported.

Direct mapping from STGs and the problem of device-environment interface were addressed in [3]. This paper

presents a method based on the idea of [3] and extends it by a set of optimisation algorithms and heuristics. In proposed method a system specification is, firstly, split into a device STG and an environment STG. These are synchronise via a communication net, which model wires. The device STG is considered separately. It consists of a *tracker* and a *bouncer*. The *tracker* follows the state of the environment and is used as a reference point by the device outputs. The *bouncer* interfaces the environment and generates output events in response to the input events according to the state of the tracker. This two-level device architecture provides an efficient interface to the environment and is convenient for subsequent mapping into a circuit netlist. These are implemented in a software tool called OptiMist. The speed-independent circuits obtained by this method have a two-level architecture, which contributes to a low-latency interface to the environment. The OptiMist tool exhibits the computation time growth linear to the specification size which allows to apply the method to large STGs.

The rest of the paper is organised as follows. Firstly, Section 2 defines terminology and behavioural models which are used in the paper. Secondly, the our direct mapping method and its justification are presented in Section 3. The optimisation heuristics and algorithms implemented in OptiMist software tool are described in Section 6. The OptiMist design flow is considered on a simple example in Section 7. The method and the optimisation heuristics are evaluated in Section 8 using a set of benchmarks.

2 Background

This section provides an introduction to asynchronous circuits, their delay models, operation modes, classes and common signalling protocols. A behavioural Petri nets model which is widely used for specification, verification and synthesis of asynchronous circuits is also presented in this section.

2.1 Asynchronous circuits

A category of circuits containing no global clock is called *asynchronous circuits [36]*. These circuits may make use of timing assumptions both within the circuit and in its interaction with environment. Based on these assumptions the asynchronous circuits can be divided into several classes. This section overviews the asynchronous circuits using a classification presented in [19, 10].

2.1.1 Delay models

An asynchronous circuit can be considered as an interconnection of two types of components, *gates* and *delay elements*, by means of *wires*. A gate computes a set of output variables (often a single output variable) as a discrete logical function of its input variables. A delay element produces a single output that is a delayed version of its input. Each wire connects an output of a single gate or delay element to inputs of one or more gates or delay elements. Primary inputs and outputs of a circuit can be considered as gates computing the identity function.

There are two major models of a delay element: *pure delay* model and *inertial delay* model. A pure delay element transmits each signal event on its input to its output with some delay regardless the shape of the signal's waveform. On the contrary, an inertial delay element alters the shape of its input waveform by attenuating short pulses, i.e. it filters out pulses of a duration less than some threshold period.

The delay elements are also characterised by their timing models. In a *fixed delay* model, a delay is assumed to have a fixed value. In a *bounded delay* model, a delay may have any value in a given timing interval. In an *unbounded delay* model, a delay may take an arbitrary finite value.

2.1.2 Operation modes

An interaction of a device circuit with its environment can be characterised by circuit operation mode. The *device* and its *environment* together form a *close system*. If the environment is allowed to respond to a device's outputs

without any timing constraints, the system is said to interact in *input-output mode*. Otherwise, environmental timing constraints are assumed. The most common example is *fundamental mode* where the environment must wait for the device to stabilise before producing new inputs.

Depending on the restrictions to the input changes, the fundamental mode is divided into several subclasses. If a single input is allowed to change at a time, the operation mode is called *Single Input Change (SIC)* fundamental mode. SIC mode forces the inputs to be sequential, which may restrict the speed of circuit operation. Another approach which allows one or more inputs to change after the circuit stabilisation, is called *Multiple Input Change* (MIC) fundamental mode. The speed of a circuit operating in this mode improves compared to SIC mode, however it may be difficult to implement a circuit operating MIC mode.

A trade-off between SIC and MIC fundamental modes is a *Burst Mode* (BM) which only allows inputs to change in groups, called bursts. Inputs in a burst may arrive in any order and at arbitrary time. A set of inputs in a burst cannot be a subset of another burst. This restriction helps a circuit to distinguish bursts one from another. The circuit waits until all inputs in a burst change before producing its outputs. The outputs must be allowed to settle before another input burst starts.

2.1.3 Classes of asynchronous circuits

The most obvious model to use for asynchronous circuits is the same as for synchronous circuits. This model is followed in *Huffman* circuits [16], which are designed to work correctly in the fundamental mode of operation. A bounded delay is assumed for both gates and wires.

Delay-Insensitive (DI) circuits are designed to operate correctly in input-output mode with unbounded gate and wire delay. These circuits are most robust with respect to manufacturing processes and environmental variations. The concept of delay-insensitive circuits originates from [6] and is formalised in [35]. The class of DI circuits built out of simple gates is quite limited. It has been proven that almost no useful DI circuits can be built if one is restricted to a class of simple gates [21]. However, many practical DI circuits can be built using complex gates [11]. A complex gate is constructed out of several simple gates. Externally a complex gate operates in a delay-insensitive manner, however internally it may rely on some timing assumptions.

In order to build practical circuits out of simple gates a relaxation of the requirements to the DI circuits is necessary. This can be achieved by introducing an *isochronic fork*, which is a forked wire where the difference in delays between the branches is negligible [2]. Asynchronous circuits with isochronic forks are called *Quasi-Delay-Insensitive* (QDI) circuits [20]. In contrast, in DI circuits, delays on the different fork branches are completely independent, and may vary considerably.

Speed-Independent (SI) circuits are guaranteed to work correctly in input-output mode regardless of gate delays, assuming that wire delays are negligible. This means that whenever a signal changes its value all gates it is connected to will see that change immediately. SI circuits introduced in [24] only considered deterministic input and output behaviour. This class has been extended to include circuits with a limited form of non-determinism in [1].

Self-timed circuits, described in [33], are built out of a group of elements. Each element may be an SI circuit, or a circuit whose correct operation relies on local timing assumptions. However, no timing assumptions are made on the communication between elements and the circuit operates in input/output mode. If both internal and external timing assumptions are used to optimise the designs, then such circuits are called *timed* [26].

2.1.4 Signalling protocols

Asynchronous circuit signalling schemes are based on a protocol called *handshake*, involving *requests*, which are used to initiate an action, and corresponding *acknowledgements*, used to signal completion of that action. These control signals provide all of the necessary sequence controls for computational events in the system.

For example, consider an interaction of two modules, a sender A and a receiver B. A request is sent from A to

B indicating that A is requesting some action from B. When B completes the action, it acknowledges the request by sending an acknowledge signal from B to A. Most asynchronous signalling protocols require a strict alternation of request and acknowledge events. These ideas can be extended to interfaces shared by more than 2 subsystems.

There are several ways of how the handshake events are encoded onto specific control wires. The most commonly used handshake protocols are the *four-phase* and *two-phase*. In *four-phase protocol*, also called *return-to-zero*, four signal transitions (two on the request and two on the acknowledgement) are required to complete a handshake. In *two-phase protocol*, also called *non-return-to-zero*, every request-acknowledgement pair of transitions indicates a new handshake.

2.2 Behavioural models

This section introduces the formal models used for the specification and verification of asynchronous circuits. First, the basic concept of Petri nets (PNs) model is presented. PNs extend the Finite State Machines (FSMs) model with a notion of concurrency, which makes them especially convenient for the specification and verification of asynchronous circuits. The formal definitions and notations in this section are based on the work introduced in [8, 25, 27, 30].

2.2.1 Petri nets

A Petri nets model, first defined in [29], is a graphical and mathematical representations of discrete distributed systems. Petri nets are used to describe and study concurrent, asynchronous, distributed, parallel and non-deterministic systems. As a graphical tool, PNs can be used as a visual communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is allows to set up state equations, algebraic equations, and other mathematical models governing the behaviour of systems.

A *Petri Net* (PN) is formally defined as a tuple $PN = \langle P, T, F, M_0 \rangle$ comprising finite disjoint sets of *places* P and *transitions* T, *arcs* denoting the flow relation $F \subseteq (P \times T) \cup (T \times P)$ and *initial marking* M_0.

There is an arc between $x \in P \cup T$ and $y \in P \cup T$ iff $(x, y) \in F$. An arc from a place to a transition is called *consuming arc*, and from a transition to a place - *producing arc*. The *preset* of a node $x \in P \cup T$ is defined as $\bullet x = \{y \mid (y, x) \in F\}$, and the *postset* as $x\bullet = \{y \mid (x, y) \in F\}$. It is assumed that $\bullet t \neq \emptyset \neq t\bullet$, $\forall t \in T$. The *pre-preset* of a node $x \in P \cup T$ is defined as $\bullet\bullet x = \bigcup_{y \in \bullet x} \bullet y$, and the *post-postset* as $x\bullet\bullet = \bigcup_{y \in x\bullet} y\bullet$.

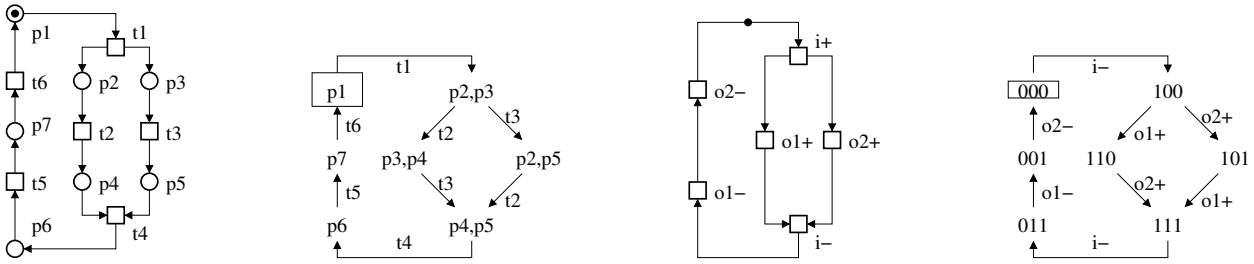
A place p such that $|p\bullet| > 1$ is called *choice place*, i.e. it has more than one transition in its postset. A choice place p is called *free choice* if $\forall t \in p\bullet : |\bullet t| = 1$, i.e. each transition in its postset has exactly one preset place. A choice place p is called *controlled choice* if $\exists t \in p\bullet : |\bullet t| > 1$, i.e. there is at least one transition in its postset which has more than one preset place. Note that a controlled choice whose all postset transitions have the same preset places can be transformed into a free choice. A place p such that $|\bullet p| > 1$ is called *merge place*. A transition t such that $|t\bullet| > 1$ is called *fork* and a transition t such that $|\bullet t| > 1$ is called *join*.

The dynamic behaviour of a PN is defined as a *token game*, changing markings according to the enabling and firing rules its transitions. A *marking* is a mapping $M : P \rightarrow \mathbb{N}$ denoting the number of *tokens* in each place, $\mathbb{N} = \{0, 1\}$ for *1-safe* PNs. A transition t is *enabled* iff $M(p) > 0, \forall p \in \bullet t$. The evolution of a PN is possible by *firing* the enabled transitions. *Firing* of a transition t results in a new marking M' such that $\forall p \in P : M'(p) = \begin{cases} M(p) - 1 & if\, p \in \bullet t, \\ M(p) + 1 & if\, p \in t\bullet, \\ M(p) & otherwise \end{cases}$, i.e. for an enabled transition t one token is removed from each preset place and one token is produced to each postset place.

A marking M' is *reachable* from a marking M if there exists a *firing sequence* $\sigma = t_0 \ldots t_n$ starting at marking M and finishing at M'. A set of reachable markings from M is denoted by $[M\rangle$. A set of markings reachable from the initial marking M_0 is called a *reachability set* of a PN.

The set of markings reachable in a PN from its initial marking can be represented as a reachability graph, whose nodes are labelled with PN markings and arcs are labelled with PN transitions. Formally, a *Reachability Graph* (RG) of a $PN = \langle P, T, F, M_0 \rangle$ is a labelled directed graph $RG = \langle S, A, l, s0 \rangle$, where $S = [M_0\rangle$ is a *reachability set*, $A = S \times T \times S$ is a set of *arcs* between these states, $l : A \rightarrow T$ is a *labelling function* indicating transitions between markings, and s_0 is the *initial state* corresponding to the initial marking of the PN.

Graphically, places of a PN are represented as circles (○), transitions as boxes (□), consuming and producing arcs are shown by arrows (⟶), and tokens of the PN marking are depicted by dots in the corresponding places (⦿). A simple PN is shown using this graphical notation in Figure 2(a). This example illustrates that, unlike FSM model, PNs model can capture concurrent actions. If two transitions are enabled in the same marking and the firing of one does not interfere with the enabling of the other, then both transitions will eventually fire. The fact that transitions *t2* and *t3* are concurrent means that both firing sequences *t2, t3* and *t3, t2* are possible. This is captured by the RG in Figure 2(b). The RG nodes are labelled with the reachable PN markings, arcs are labelled with the corresponding PN transitions and its initial state is marked with a box.



(a) PN (b) RG (c) STG (d) SG

Figure 2: Simple examples of PN, RG, STG and SG

Transitions in a PN can be involved in different ordering relations. Two PN transitions are in *direct conflict* if there exists a reachable marking in which both of them are enabled but firing of one of them disables the other. *Conflict* relations can be generalised by considering the transitive successors of directly conflicting transitions. If two transitions are enabled in some reachable marking but are not in direct conflict, they are *concurrent*. Transitions which are not concurrent and are not in a transitive conflict are *ordered*.

Important properties of a PN are safeness, liveness and deadlock-freeness. A PN is said to be k-*bounded* if the number of tokens in every place of a reachable marking does not exceed a finite number k. A 1-bounded PN is also called *1-safe*. A PN is *deadlock-free* if, no matter what marking has been reached, it is possible to fire at least one transition of the net. A PN is *live* if for every reachable marking M and every transition t it is possible to reach a marking M' that enables t.

An extension of a PN model is a *contextual net* [23]. It uses additional elements such as *non-consuming arcs*, which only control the enabling of transitions and do not influence their firing. A PN extended with a type of non-consuming arcs, namely *read-arcs,* is defined as $PN = \langle P, T, F, R, M_0 \rangle$. A set of read-arcs R is defined as $R \subseteq (P \times T)$, there is a read-arc between p and t iff $(p, t) \in R$. The *read-preset* of a transition $t \in T$ is defined as $\star t = \{p \mid (p, t) \in R\}$, and the *read-postset* of a place $p \in P$ as $p\star = \{t \mid (p, t) \in R\}$. Place p *controls* transition t by means of a read-arc iff $p \in \star t$. A transition t *reads* the state of a place p iff $t \in p\star$. A transition t is enabled iff $M(p) \neq 0, \forall p \in \bullet t \cup \star t$. The rules for firing of the transitions are preserved. A read-arc is depicted as a line without arrows.

The following are three most common subclasses of PNs. A PN is called a *Marked Graph* (MG) iff $\forall p \in P : |\bullet p| \leq 1 \wedge |p\bullet| \leq 1$, i.e. each place has at most one preset and one postset transition. The nets of this subclass represent deterministic concurrent systems. Dually, a PN is called a *State Machine* (SM) iff $\forall t \in T : |\bullet t| = 1 \wedge |t\bullet| = 1$, i.e. each transition has exactly one preset and one postset place. This subclass allows to represent non-deterministic

sequential systems. A PN is called *Free Choice* (FC) net iff for any choice place $p\ \forall t \in p\bullet : |\bullet t| = 1$, i.e. each transition in the postset of a choice place has exactly one preset place. Free choice nets model both non-determinism and concurrency but restricts their interplay. The former is necessary for modelling choice made by the environment whereas the latter is essential for asynchronous behaviour modelling.

The two modelling extensions of PNs are Labelled PNs and Coloured PNs. A *Labelled Petri Net* (LPN) is a PN whose transitions are associated with a labelling function [38]. The extension of non-consuming arcs is also applicable to the LPN definition. A *Coloured Petri Net* (CPN) is a formal high-level net where places are associated with data types, tokens are associated with the data values and transitions denote the operations on that data [17]. This allows the representation of data path in a compact form, where each token is equipped with an attached data value.

2.2.2 Signal transition graphs

The Signal Transition Graph (STG) model was introduced independently in [5] and [31] to formally model both the circuit and the environment. The STG can be considered as a formalisation of the widely used timing diagrams. It describes the causality relations between transitions on the input and output signals of a specified circuit. It also allows the explicit description of data-dependent choices between various possible behaviours. STGs are interpreted Petri nets, and their close relationship to Petri nets provides a powerful theoretical background for the specification and verification of asynchronous circuits.

An STG is a 1-safe LPN whose transitions are labelled by signal events, i.e. $STG = \langle P, T, F, M_0, \lambda, Z, v_0 \rangle$, where λ is a *labelling function*, Z is a set of *signals* and $v_0 = \{0, 1\}^{|Z|}$ is a *vector of initial signal values*.

The set of signals Z is divided into two disjoint sets of *input signals* Z_I and *output signals* Z_O, $Z = Z_I \cup Z_O$, $Z_I \cap Z_O = \emptyset$. Input signals are assumed to be generated by the environment, whereas output signals are produced by the logic gates of the circuit. *Internal signals* may also be included in the Z_O set.

The labelling function $\lambda : T \rightarrow Z \pm \cup \Theta$ maps transitions into *signal events* $Z\pm = Z \times \{+, -\}$ and *dummies* Θ, $Z \pm \cap \Theta = \emptyset$. The signal events labelled $z+$ and $z-$ denote the transitions of signals $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. Dummy transitions are *silent events* that do not change the state of any signal. The labelling function does not have to be 1-to-1, i.e. transitions with the same label may occur several times in the net. In order to distinguish between transitions with the same label and refer to them from the text an index $i \in \mathbb{N}$ is attached to their labels as follows: $\lambda(t)/i$, where i differs for different transitions with the same label.

In order to be implementable as a circuit an STG must satisfy the property of consistency. An STG is *consistent* if for each signal $z \in Z$ transitions labelled $z-$ and $z+$ alternate in any firing sequence starting from M_0. In this work it is assumed that all the considered STGs are consistent.

A *vector of signal change* $v_\sigma = \left(v_\sigma^1, ..., v_\sigma^{|Z|}\right)$ can be associated with a finite sequence of transitions σ, so that each v_σ^i is the difference between the number of rising and falling edges of signal z_i in σ. The *vector of signal values* $v = v_0 + v_\sigma$ defines the states of all STG signals after some sequence of transitions σ. Note that for consistent STGs the vectors v_0, $v_\sigma$and v are binary.

A *projection* of a firing sequence σ onto a set of signals $X \subseteq Z$ is defined as $\sigma \downarrow X = \{t \in \sigma : \lambda(t) \in X\pm\}$, i.e. it only includes transitions of signals in X. A *silent sequence* θ is a firing sequence (possibly empty) such that $\theta \downarrow Z = \emptyset$, i.e. its projection on the set of signals is empty. Similarly, a firing sequence whose the projection on the set of output (input) signals is empty and projection on the set of input (output) signals is not empty is called *input (output) sequence*.

STGs inherit the operational semantics of their underlying PNs, including the notations of transition enabling and firing. Likewise, STGs also inherit the various structural (marked graph, free-choice, etc.) and behavioural properties (boundedness, liveness, etc.). Note that a set of read-arcs can be included into the model of STG, which is an enhancement w.r.t. [31].

For graphical representation of STGs a short-hand notation is often used, where a transition can be connected to another transition if the place between those transitions has one incoming and one outgoing arc as illustrated in

Figure 2(c).

A state of an STG without dummies is a pair $\langle M, v\rangle$, where M is a reachable marking and v is vector of signal values corresponding to this marking. Note that the vector of signal values along does not uniquely identify the STG state. Also in general case the same marking can correspond to different states of the STG (e.g., if it is not live or not consistent). The extension of an STG with the notion of dummy transitions complicates the definition of its state. As a dummy transition does not correspond to a signal event, firing of a dummy does not actually change the state of the system described by the STG. That is why the states of an STG before and after firing a dummy are considered equal, though the marking is different. In this work we assume that a dummy (or in a more general case a silent sequence) is a part of preceding signal transition.

In the same way as an STG is an interpreted PN with transitions associated with binary signals, a state graph is the corresponding binary interpretation of an RG in which the events are interpreted as signal transitions. Formally, a *State Graph* (SG) of an $STG = \langle P, T, F, M_0, \lambda, Z\rangle$ is a quadruple $SG = \langle S, A, l, C, s_0\rangle$, where S is a set of *reachable states*, $A = S \times T \times S$ is a set of *arcs* between these states, $l : A \rightarrow T$ is a *labelling function* for the arcs, $C : S \rightarrow \{0, 1\}^{|Z|}$ is a *state assignment function*, which is defined as $C(\langle M, v\rangle) = v$, and $s_0 = \langle M_0, v_0\rangle$ is the *initial state*. Note that dummies are

The SG of the STG in Figure 2(c) is shown in Figure 2(d). Each SG state corresponds to a marking of the STG and is assigned a binary vector. Each SG arc corresponds to firing of a signal transition. For readability the SG arcs are indicated by the transition labels. The signal order in the binary vectors is $\langle i, o1, o2\rangle$. The initial state (marked with a box) corresponds to the marking $\{p1\}$ with the signal values vector 000.

A property of an STG which simplifies its hardware implementation is persistency. An STG is *persistent* if no transition can be disabled by another transition unless they both are events of different input signals. This means that all non-deterministic behaviour is part of the environment and the arbitration is avoided in the device.

Properties of an STG specific for a logic synthesis approach are *unique state coding* and *complete state coding*. The former is sufficient condition and the latter is necessary condition of a circuit implementability by logic synthesis. Two distinct states of an SG are in a *Unique State Coding* (USC) conflict if they are assigned to the same code. Two distinct states of a SG are in a Complete State Coding (CSC) conflict if they are assigned to the same code and the set of enabled output signals is different in these states. An STG satisfies the USC (CSC) property if no two states of its SG are in USC (CSC) conflict. Note that neither USC nor CSC is required in a direct mapping approach. The properties of an STG which are specific for the proposed direct mapping method are considered in Section 3.1.

2.2.3 Bisimulation

Bisimulation, originally introduced in [14, 22], is an equivalence relation between STGs, associating systems which behave in the same way, in the sense that one system simulates the other and vice-versa. Intuitively two systems are bisimular if they match each other's moves, i.e. each of the systems cannot be distinguished from the other by an observer.

Two systems described by $STG = \langle P, T, F, R, M_0, \lambda, Z, v_0\rangle$ and $STG' = \langle P', T', F', R', M'_0, \lambda', Z', v'_0\rangle$ are *(strongly) bisimular*, notation $STG \sim STG'$, iff:
(i) $M_0 \sim M'_0$;
(ii) if $M \sim M'$ and $M\,[t\rangle\, M_1$ then $\exists\, t' \in T'$ such that $\lambda(t) = \lambda'(t')$, $M'\,[t'\rangle\, M'_1$ and $M_1 \sim M'_1$;
(iii) as (ii) but with roles of STG and STG' reversed.

The notion of strong bisimulation requires a system to be capable of matching each transition that an equivalent system may perform. However, sometimes internal and external (observable) behaviour of a system are distinguished. In this sense two systems are equivalent if they exhibit the same external behaviour, irrespective of any intermediate internal behaviour that may occur. For example, if the system STG includes a notion of silent actions (dummies), then bisimulation can be relaxed to ignore these dummies.

Two systems represented by $STG = \langle P, T, F, R, M_0, \lambda, Z, v_0\rangle$ and $STG' = \langle P', T', F', R', M'_0, \lambda', Z', v'_0\rangle$

are *weakly (observationally) bisimular*, notation $STG \approx STG'$, iff:
(i) $M_0 \approx M'_0$;
(ii) if $M \approx M'$ and $M\,[t\rangle\,M_1$ then either $\lambda(t) = \tau$ and $M_1 \approx M'$ or $\exists\, t' \in T'$, $\lambda(t) = \lambda'(t')$ and silent sequences θ_1, θ_2 such that $M'\,[\theta_1\rangle\,M'_{\bullet t}\,[t'\rangle\,M'_{t\bullet}\,[\theta_2\rangle\,M'_1$ and $M_1 \approx M'_1$;
(iii) as (ii) but with roles of STG and STG' reversed.

Still, the notion of weak bisimulation cannot be regarded as the natural generalisation of strong bisimulation for STGs with silent events. The reason for this is that an important feature of bisimulation is missing for weak bisimulation. Namely the property that any firing sequence in one STG corresponds to a firing sequence in the other, in such a way that all intermediate states of these STGs correspond as well. However, according to the definition of the weak bisimulation one may fire arbitrary many silent transitions in an STG without worrying about the markings that are passed through in the meantime. For example, the STGs in Figure 3 are weakly bisimilar, however in the right STG there is a trace which does not enable the $b+$ transition, while $b+$ is enabled in all traces of the left STG. Thus, the observational equivalence does not preserve the branching structure of STGs and hence lacks one of the main characteristics of bisimulation semantics.

Figure 3: Observation bisimulation

An alternative definition of observational equivalence which preserves the branching structure of STGs was proposed in [13]. This equivalence, called *branching bisimulation*, requires all intermediate markings in silent sequences θ_1 and θ_2 of STG' to be related with markings M and M_1 of STG respectively. Note that STGs in Figure 3 are not branching bisimilar.

Obviously, branching bisimulation is stronger than weak bisimulation. However, one can see that for the class of persistent STGs weak bisimulation becomes equivalent to branching bisimulation.

3 Method

A distinctive characteristic of the proposed direct mapping technique is that the system STG is converted into a form convenient for mapping into circuit netlist. It is achieved by associating groups of places and transitions to the state holding elements and by modelling connections between circuit components with arcs.

The initial specification describes the behaviour of both device and environment as a complete system. Usually, only the device needs to be synthesised which requires the extraction of the device model from the system STG. In order to do this the system STG is split into a device model and an environment model, which are connected by an intermediate net. Only the device model is subsequently optimised and mapped into a circuit netlist.

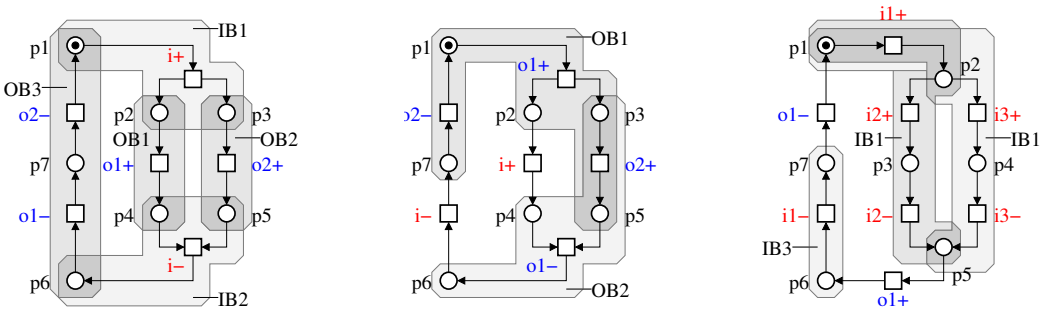
3.1 Requirements to the initial specification

There are several limitations on the class of STGs which can be synthesised using our direct mapping method. Similarly to the requirements of logic synthesis methods the STG must be consistent and persistent. The STG consistency is essential for any hardware implementation due to the nature of binary signals whose rising and falling transitions alternate. Persistency is required to avoid arbitration in the device by letting the environment make all the choices.

Unlike logic synthesis methods in our approach neither USC nor CSC is necessary for the whole STG. The limitation on the state encoding is more relaxed and is defined using the notion of bursts: a maximally connected subgraph of

an STG which only includes transitions of an input (output) sequence and places incident to them is called *input (output) burst*. Two bursts are said to be in *conflict* if there is a transition in one burst which is in a direct conflict with a transition from another burst. A burst *B1* is said to be *covered* by burst *B2* if they are in conflict and all signal events of *B1* also exist in *B2* possibly in different order. Note that in a persistent STG only input bursts can be in conflict and covered.

The notion of bursts is illustrated in Figure 4. The STG in Figure 4(a) contains two input and two output bursts (*IB1*, *IB2* and *OB1*, *OB2* respectively). Note that even though an output sequence *o1+, o2+* is possible from a reachable marking $\{p2, p3\}$, the output bursts *OB1* and *OB2* are separate because their graphs are not connected. The example in Figure 4(b) shows two output bursts *OB1* and *OB2* (input bursts are trivial and are hidden for simplicity). These output bursts are overlapping, however they cannot be merged into one burst because there is no output sequence which contains both *o1+* and *o1-*. In Figure 4(c) three input bursts are shown (trivial output bursts are hidden). Note that conflicting input bursts *IB1* and *IB2* are separated in this STG even though their graphs are overlapping because these bursts belong to different input sequences.



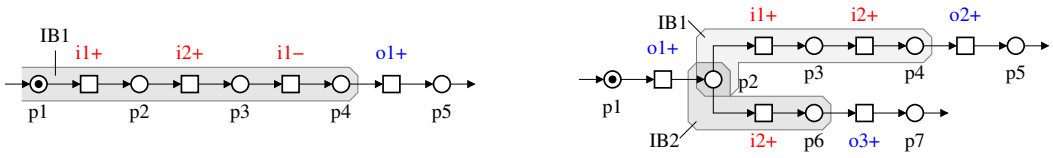
(a) Simple bursts (b) Concurrency (c) Choice

Figure 4: Input and output bursts

All the requirements to encoding of the system states in the proposed direct mapping method are due to the delay-insensitive nature of the device-environment interface. Usually a designer can control the wire delays inside a relatively small device and build it speed-independent. However, the delays of connections between the device and the environment often cannot be guaranteed. The uncontrollable interconnect delays on the device-environment interface may result in a situation when two signals issued in sequence by a sender reach a receiver simultaneously or even in a reversed order. This means that while the device itself can be speed-independent, its interface to the environment should be built under delay-insensitive timing assumptions.

One of the ways to ensure the delay-insensitive interface is to apply *order relaxation* [32] to the initial STG. This approach, however, may complicate the structure of the STG, which is disadvantageous for a direct mapping technique. In our direct mapping approach the order relaxation is not applied explicitly. Instead, a device distinguishes the end of an input burst by catching an encoding in which all inputs comprising the burst have switched. The unique identification of such encoding is only possible if all states corresponding to the input burst are coded uniquely. The opposite is also true for output bursts: in order to uniquely identify the end of an output burst the encodings of its states should be unique. For example, the STG in Figure 5(a) has an input burst *IB1* which can cause problems. If due to interconnect delays *i2+* reaches the device before *i1+* then the device can produce *o1+* by mistake even without waiting for *i1+* and *i1-*.

Thus, in our method each input and output burst of the system STG must have USC. An STG is said to satisfy *burst USC* property if there is no USC conflict in any input or output burst, i.e. the state encoding is unique within each individual burst. In order to satisfy the burst USC property it is sufficient for a consistent and persistent STG to



(a) Non-unique burst encodding

(b) Conflicting bursts

Figure 5: Bursts in a delay insensitive interface

have no more than one transition of each signal in every input and output burst. The uniqueness of a signal transition in each burst implies monotonic change of the code and hence no repetition of the state encoding within the burst.

Another type of ambiguity introduced by the delays in the device-environment interconnect may occur when choosing between conflicting branches. For example, the STG in Figure 5(b) has two input bursts *IB1* and *IB2* which are in conflict. The transitions in the direct conflict are different (*in1*+ and *in2*+) and the choice is unambiguous. However, if the transitions of the burst *IB1* reach the device in the reverse order (*in2*+ first), then the device will be confused which conflicting branch the environment selected. The device still can recognise the branch selection if the following condition holds: the state encodings after each input burst is different from all encodings in the bursts it conflicts with. In order to satisfy this condition it is sufficient for a consistent and persistent STG with burst USC to contain only non-covered bursts; such an STG is called *non-covered*. Indeed, the state of all STG signals is the same before conflicting bursts and the change of encoding is monotonic within each burst. If none of the bursts is covered by the others then the encoding after a burst is not repeated in any burst it conflicts with.

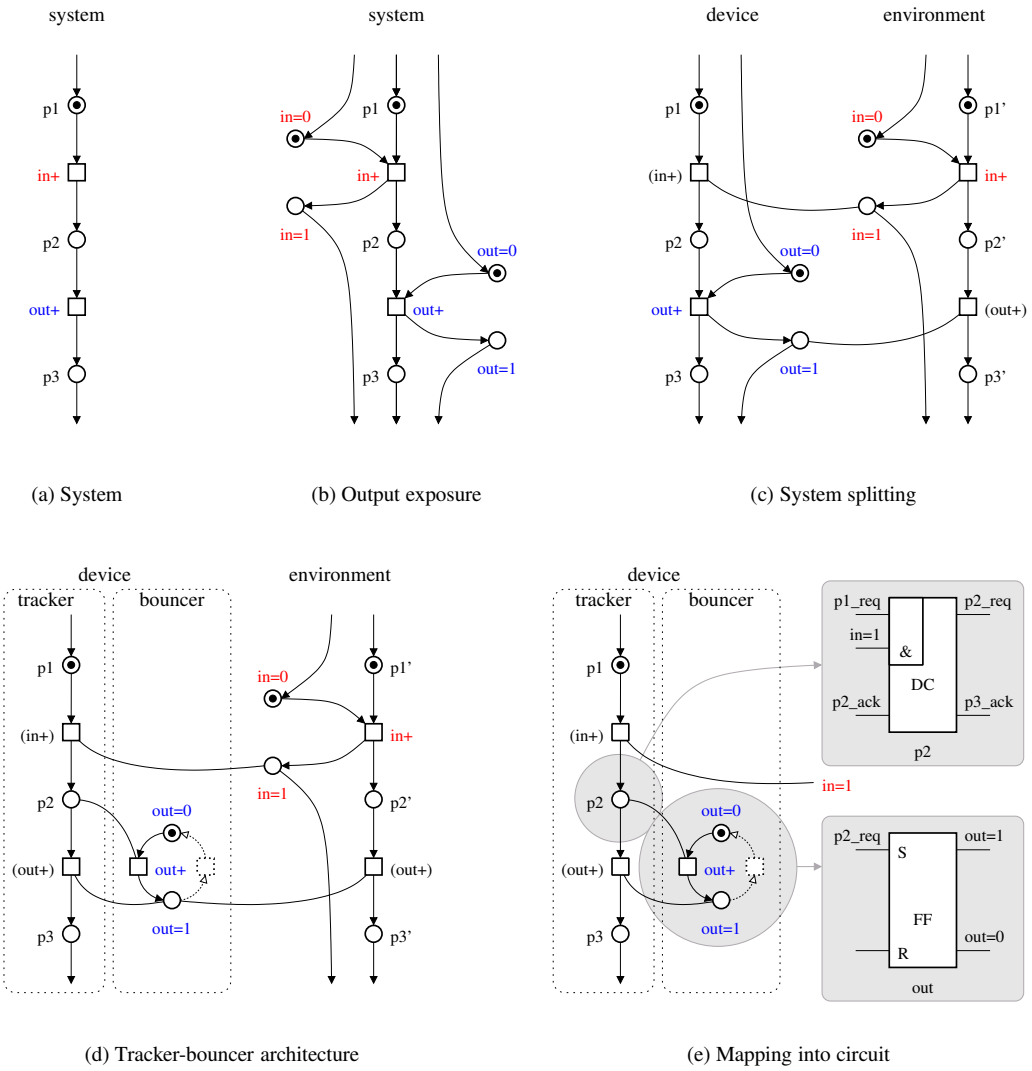
To summarise, in order to be mappable into a circuit using our direct mapping method a system STG must be consistent, persistent, non-covered and must have burst USC. Checking these properties is computationally hard problem which does not feet into a direct mapping design flow aiming at low algorithmic complexity. Instead it is assumed that the control path STG is supplied by a high-level synthesis tool which insures the above properties by construction. All the transformations presented in the following sections preserve the behavioural equivalence if the original STG satisfying these properties.

3.2 Transformation

The idea of the our direct mapping method is illustrated on a basic example whose STG is partially shown in Figures 6(a). The depicted slice of the specification contains the *in*+ input event causing the *out*+ output event.

The first step in extracting the device model is the exposure of the signal states as shown in Figure 3.2(b). For this each signal z is associated with a pair of complementary places $z = 0$ and $z = 1$ representing low and high levels of the signal. These places are inserted as transitive places between positive and negative transitions of z, thus expressing the property of signal consistency. Note that the transitive places do not change the behaviour of the system and weak bisimulation is preserved on this stage of transformation.

The second step of the transformation is splitting the system specification into *device* and *environment* parts as shown in Figure 3.2(c). For this the STG obtained in the first step is duplicated. In the first copy, corresponding to the device, the transitive places associated to the inputs are removed. Similarly, in the second copy, corresponding to the environment, the transitive places associated to the outputs are removed. The behaviour of the device and the environment parts is synchronised by means of read-arcs as follows. In the environment part, each transitive place associated to low (high) level of an input signal z_I is connected by read-arcs to all negative (positive) transitions of z_I in the device. After that the transitions of input signal z_I in the device part are replaced by dummies. This way the device follows (or tracks) the behaviour of environment. Similar procedure applies to all output signals but with the roles of device and environment changed. In the device part, each transitive place associated to low (high) level



(a) System (b) Output exposure (c) System splitting

(d) Tracker-bouncer architecture (e) Mapping into circuit

Figure 6: Method for the direct mapping from STGs

of an output signal z_O is connected by read-arcs to all negative (positive) transitions of z_O in the environment. The transitions of z_O in the environment part are replaced by dummies. Now the environment also tracks the operation of the device. For convenience each dummy introduced in this step are labelled by the original transition name put in parenthesis.

The transformation of the second step splits each transition of an output (input) signal into the signal transition itself which belongs to the device (environment) and a dummy in the environment (device). The firing of these two transitions are ordered by read-arcs, so that the interface signal transition is enabled first and only after this transition fires the corresponding dummy is enabled. The dummy transition cannot be disabled until it fires because of the burst USC property of the original STG. Thus the behavioural equivalence is preserved on this step of transformation.

The third step of the transformation is splitting the device into *tracker* and *bouncer* parts as shown in Figure 3.2(d). There is no need to further transform the environment part as only the device will be subsequently implemented. The tracker-bouncer splitting starts from representing each output signal by an elementary cycle. An *elementary cycle* of a signal z consists of two places $z = 0$ and $z = 0$ (these are transitive places added in the first transformation step), and several positive and negative transitions of z connecting these places. The positive transitions of z are inserted after $z = 0$ and before $z = 1$. Similarly, the negative transitions of z are inserted after $z = 1$ and before $z = 0$. The number

of positive and negative transitions of z is equal to the number of corresponding events in the device specification and can be more than one. The set of elementary cycles for all output signals forms the device bouncer and the rest is the device tracker. The elementary cycles of the bouncer are synchronised with the tracker part by means of read arcs as shown in Figure 3.2(d). Each positive (negative) transition t_B of signal z in the bouncer is uniquely associated to a positive (negative) transition t_T of the same signal z in the tracker. A transition t_T is called a *prototype* of t_B. All places in the preset of t_T are connected by read-arcs to t_B and the only place in the postset of t_B is connected by a read-arc to t_T. After that the prototype transition t_T in the tracker is replaced by a dummy which is labelled by the original transition name in parenthesis.

The transformation described in the third step basically splits each output signal transition $z\pm$ into the signal transition $z\pm$ itself (in the bouncer) and a dummy $(z\pm)$ (in the tracker). The transition $z\pm$ is enabled only when all the places in the preset of $(z\pm)$ have tokens. These tokens cannot propagate further because the dummy is disabled by a read-arc from the postset of $z\pm$. The only place p_z in the post of $z\pm$ is either $z = 0$ or $z = 1$ depending on the polarity of $z\pm$. As soon as $z\pm$ fires the dummy $(z\pm)$ becomes enabled and the tokens continue their move in the tracker. It is also necessary that the token does not leave p_z until $(z\pm)$ fires. This condition is ensured by the signal consistency of the initial STG. Thus the transformation of this step preserves the behavioural equivalence of the system.

From this point the device model is considered separately and the environment is assumed to produce inputs in response to device outputs according to the system protocol. The elementary cycles of the device bouncer are subsequently implemented as set-reset Flip-Flops (FF) and the places of the device tracker are mapped into DCs, see Figure 3.2(e).

3.3 Optimisation

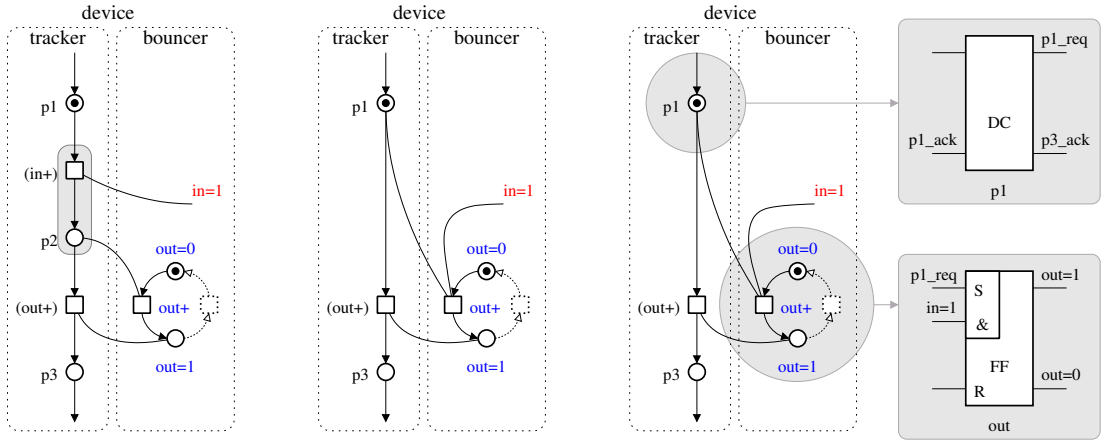
It is often possible to control outputs by the directly preceding interface signals without using intermediate states. Many places and preceding dummies can thus be removed, provided that the system behaviour is preserved w.r.t. input-output interface (weak bisimulation). Such places are called *redundant*. Note that the notion of redundant places in our method is different from the redundant transitive places in the structural theory of Petri nets, thus the structural theory cannot be applied to remove them. This way $p2$ is redundant in the considered example, Figure 7(a). It can be removed from the device tracker together with the preceding dummy $(in+)$ as shown in Figure 7(b). Now the input $in = 1$ controls the output $out+$ transition directly, which results in latency reduction when the STG is mapped into the circuit, see Figure 7(c). Before the optimisation the output flip-flop was set by the $p2\_req$ signal, which was generated in response to the input in, see Figure 7(e). In the optimised circuit the output flip-flop is triggered directly by the in input and the context signal $p1\_req$ is calculated in advance, concurrently with the environment action.

4 Coding conflicts

The elimination of places is restricted by potential *coding conflicts* which may cause tracking errors. There are two types of conflicts: *Mark Graph-specific* and *State Machine-specific*. The former conflicts may appear in a non-conflicting branch of an STG, the latter may appear in the conflicting branches after a choice place.

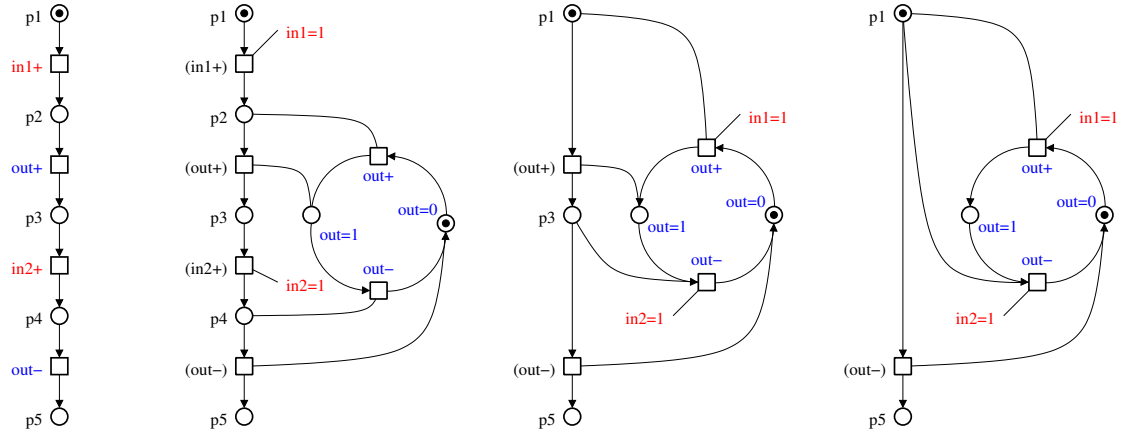
4.1 Mark Graph-specific coding conflicts

For the idea of an Mark Graph-specific coding conflict, consider the system whose STG is depicted in Figure 8(a). The device specification extracted from this STG by applying the above method is shown in Figure 8(b). The tracker part of the device can be further optimised. The removal of redundant places $p2$ and $p4$ does not cause any conflicts of the tracker, Figure 8(c). However, if the place $p3$ is eliminated as shown in Figure 8(d), then the tracker cannot distinguish between the output having not yet been set and the output already reset. Note the specifics of this direct mapping approach: only those signals whose switching directly precedes the given output are used in its support.



(a) Redundant places (b) Optimisation (c) Mapping into circuit

Figure 7: Optimisation of the device specification



(a) System (b) Output exposure (c) Correct optimisation (d) Coding conflict

Figure 8: Preventing coding conflicts

It is computationally simpler to detect redundant places by processing the original specification. For this the set of all STG places P is divided into three non-intersecting subsets: P_U, P_R, P_M such that $P_U \cup P_R \cup P_M = P$ and $P_U \cap P_R \cap P_M = \emptyset$. The set P_R consists of redundant places which can be safely removed from the device STG, the set P_M holds the mandatory places which must be preserved in the device model, and the set P_U contains the places which have not been considered yet (undefined places). In the following figures the undefined places are depicted as ordinary circles (○), redundant places are drawn as small circles (∘), and the mandatory places are shown as bold circles (●). Initially all STG places belong to P_U, both sets P_R and P_M are empty. Then, each place in P_U is tested for being redundant. If the place removal does not cause a coding conflict then the place is redundant and is moved into P_R, otherwise it is mandatory and is moved into P_M.

A coding conflict for a place p is detected by intersecting two sets of signals. The first set contains the signals whose transitions are fired in the *forward neighbourhood* of place p limited by places and transitions in $P_R \cup T$. The second set consists of the signals whose transitions are fired in the *backward neighbourhood* of place p limited by places and transitions in $P_R \cup T$.

The *forward neighbourhood* $\varepsilon_f(p,\ X)$ of place p limited by nodes (places and transitions) in X is defined as the minimal (w.r.t.$\subseteq$) set such that:

$$\begin{aligned}\varepsilon_f(p,\ X):\quad & p \in \varepsilon_f(p,\ X);\\ & \forall x \in X\ if\ \exists y \in \varepsilon_f(p,\ X):\ x \in y\bullet,\ then\ x \in \varepsilon_f(p,\ X)\end{aligned} \tag{1}$$

The set of signals $Z_f(p,\ X)$ whose transitions are fired in the forward neighbourhood of place p limited by the set of nodes X are defined using labelling function λ:

$$Z_f(p,\ X) = \{z:\ \exists t \in \varepsilon_f(p,\ X) \cap T:\ \lambda(t) \in \{z+,\ z-\}\} \tag{2}$$

Similarly, the *backward neighbourhood* $\varepsilon_b(p,\ X)$ of place p limited by the set of nodes X is defined as the minimal (w.r.t.$\subseteq$) set such that:

$$\begin{aligned}\varepsilon_b(p,\ X):\quad & p \in \varepsilon_b(p,\ X);\\ & \forall x \in X\ if\ \exists y \in \varepsilon_b(p,\ X):\ x \in \bullet y,\ then\ x \in \varepsilon_b(p,\ X)\end{aligned} \tag{3}$$

$$Z_b(p,\ X) = \{z:\ \exists t \in \varepsilon_b(p,\ X) \cap T:\ \lambda(t) \in \{z+,\ z-\}\} \tag{4}$$

For the detection of a coding conflict the forward and backward neighbourhoods are calculated on a set of transitions and redundant places. If $Z_f(p,\ P_R \cup T) \cap Z_b(p,\ P_R \cup T) = \emptyset$, then the removal of the place p does not cause coding conflicts. However, if there is a signal z whose transitions belong to both the forward and the backward neighbourhoods of place p limited by transitions and redundant places, then the removal of this place causes a coding conflict for signal z. The state of the signal z is the same before its transition in the backward neighbourhood and after its transition in the forward neighbourhood of place p. As the place p is the only undefined place between these transitions (the others are redundant) it must be preserved in order to separate the same state of the signal z in different parts of the system specification.

Consider the detection of coding conflict on the example shown in Figure 9. The place under question is *p07*. Its forward neighbourhood limited by redundant places is *{p07, in2+, p08, p09, out2-, out3+}* and its backward neighbourhood is *{p07, out1+/1, out2+/2, p05, p06, p04, in1+/1, out1+/2, in1+/2, p01, out2+/1}*. Places *p00*, *p02*, *p03*, *p11*, *p12* are not redundant and form a border for the place *p07* neighbourhoods. The signals whose transitions are fired in the forward and backward neighbourhoods are *{in2, out2, out}* and *{in1, out1, out2}* respectively. The intersection of these sets is *{out2}* which means that place *p07* separates the different states of the signal *out2* and removal of this place will cause a coding conflict.

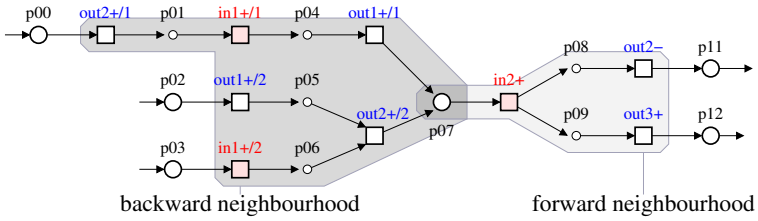
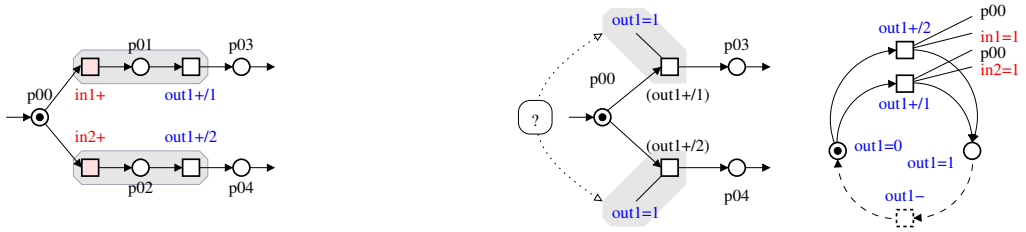


Figure 9: Forward and backward neighbourhoods

4.2 State Machine-specific coding conflicts

The situation becomes more complex if a place under test is a direct successor of a choice transition. For example, places *p01* and *p02* in Figure 10(a) are not redundant. If these places are removed, then the choice between the conflicting branches is controlled by the same condition *out1=1*, which is ambiguous, see Figure 10(b). This is an

State Machine-specific coding conflict. The forward and backward neighbourhoods do not help here because the places whose removal cause a coding conflict are in mutually exclusive branches.



(a) Initial STG

(b) An ambiguouty caused by removal of places p01 and p02

Figure 10: Mandatory places after choice

There are several strategies to avoid the ambiguity in choice branches after choice place p_{choice}. The first extreme is to preserve all post-postset places of the choice place. This approach is computationally simple, however the latency reduction might be sacrificed if there is an input signal transition in the postset of the choice place (which is the case for a free-choice). The other extreme is to traverse each conflicting branch from p_{choice} and find all signals whose transitions are present in more than one branch. At least one mandatory place must precede the first transition of such signals in each branch. This approach helps to reduce the input-output latency, but it may be computationally hard if loops or nested choices are found.

A trade-off between the computation speed and latency optimisation is the following. First, each undefined place p_{succ} in the post-postset of p_{choice} is checked. The removal of place p_{succ} does not improve the input-output latency if all preset transitions of p_{succ} are non-input signal events. Such a place is made mandatory to help reducing the computation complexity of the next step. After that, for each transition $t \in p_{choice}\bullet$ a set of transitions $T_{seq}(t) = \{t\} \cup t\bullet\bullet$ is built. It contains the transitions in a choice branch starting from t for the depth of two transitions counting from p_{choice}. The joint set of transitions in choice branches of place p_{choice} limited by the depth of two transitions is $T_{choice}(p_{choice}) = \bigcup_{t \in p_{choice}\bullet} T_{seq}(t)$. If for a transition $t \in p_{choice}\bullet$ there is a signal whose transition belongs to both $T_{seq}(t)$ and $T_{choice}(p_{choice}) \setminus T_{seq}(t)$ then the places in the postset of t must be preserved. Otherwise all places in the postset of $t\bullet\bullet$ transitions are made mandatory to reduce the depth of traversing. The trade-off approach benefits from the input-output latency reduction and low computation complexity, however the size optimisation might suffer.

Consider the application of the trade-off approach to the example in Figure 10(a). For the choice place $p00$, which has transitions $in1+$ and $in2+$ in its postset, $T_{seq}(in1+) = \{in1+, out1+/1\}$, $T_{seq}(in2+) = \{in2+, out1+/2\}$ and $T_{choice}(p00) = \{in1+, out1+/1, in2+, out1+/2\}$. As a transition of $out1$ belongs to both $T_{seq}(in1+)$ and $T_{choice}(p00) \setminus T_{seq}(in1+) = T_{seq}(in2+)$ the place $p01$is mandatory. Similarly, a transition of $out1$ belongs to both $T_{seq}(in2+)$ and $T_{choice}(p00) \setminus T_{seq}(in2+) = T_{seq}(in1+)$, which makes place $p02$ mandatory.

5 Mapping into circuit

In our direct mapping method the places of the tracker are mapped into DCs and the elementary cycles of the bouncer are mapped into set-reset FFs. Read-arcs connecting the elementary cycles to the tracker and to the environment simulate the behaviour of wires and are directly mapped into wires between DCs and FFs.

The mapping of an elementary cycle into a set-reset FF is illustrated in Figure 11. The set and reset functions of the FF are determined by the structure of the set and reset phases of the elementary cycle. The FF is initially set (reset) if the place corresponding to the low (high) level of the signal is marked.

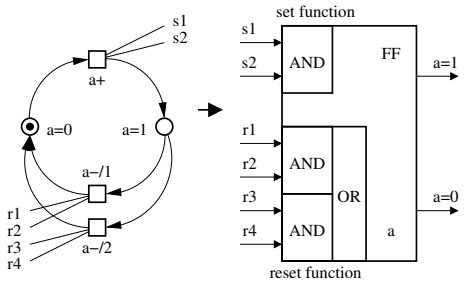


Figure 11: Mapping of elementary cycles into FFs

The mapping of basic tracker structures into DCs is shown in Figure 12. The request and acknowledgement functions of each DC are generated from the structure of the tracker in the preset and postset of the corresponding place. The request function of each DC is shown in its top-left corner and the acknowledgement function in its bottom-right conner.

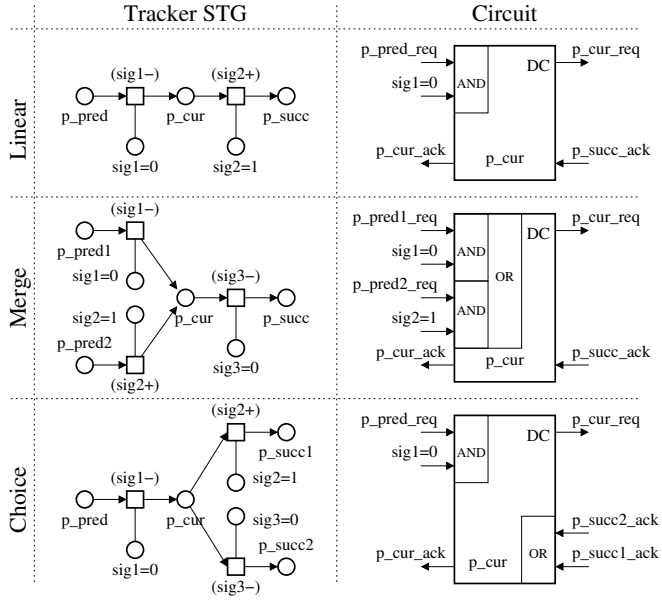
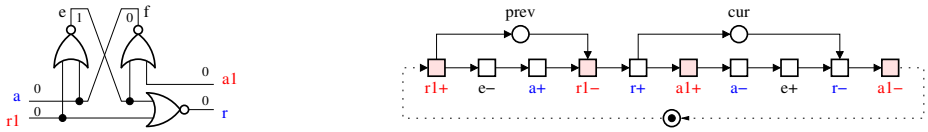


Figure 12: Mapping of tracker places into DCs

A circuit diagram of a traditional DC which was introduced in [9] is shown in Figure 13(a). The input *r1* is the request from the previous stage DC to pass the token. Output *a* acknowledges the receipt of the token to the previous stage DC. Similarly, output *r* requests the next stage DC to accept the token and input *a1* acknowledges its receipt. Signals *e* and *f* represent the 'empty' and 'full' states of the state holding element. Similar to [18], in our method the marking of a tracker place is associated with the state of the output *r* of a corresponding DC. The operation of a DC is illustrated by the STG in Figure 13(b). The transitions of the internal signal *f* are skipped in this STG because this signal is equivalent to the *a* output. The transitive places *prev* and *cur* represent the active levels of signals *r1* (previous stage DC holds a token) and *r* (this DC holds the token) respectively. Note that neither the previous stage DC nor the current one is active while the token moves between transitions *r1-* and *r+*. In most cases this time can be considered as negligible, because it corresponds to a single two input NOR-gate delay.

There is one timing assumption in this DC implementation which is represented by the dotted arcs in Figure 13(b). The assumption is that a new token arrives to the input of a DC only after the token has left the next stage DC. This assumption results in a limitation of the method to have at least three DCs in every loop [18]. For the original

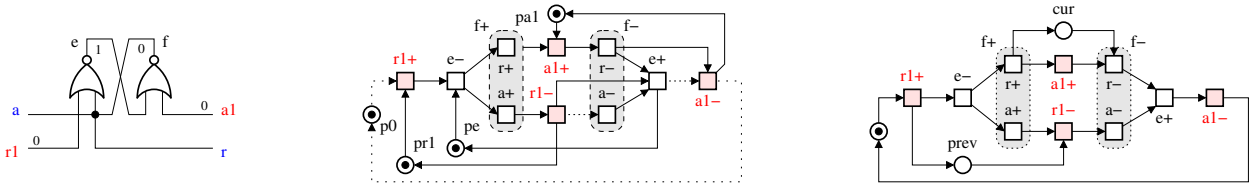


(a) Circuit (b) STG

Figure 13: David cell

specification it means that any loop of the system STG must contain at least three places. Also the number of places in the loops must be kept above two during the optimisation of the tracker STG.

Faster and more compact solutions for a DC implementation, proposed in [3], are called *fast DCs*. A gate-level implementation of a fast DC and its STG are shown in Figure 14(a,b). An interesting feature of the fast DC is that it internally contains a GasP-like interface [34], which uses a single wire to transmit a request in one direction and an acknowledgement in the other. A fast DC has speed advantages over a traditional DC because the reset phase of its state holding element happens concurrently with the token move into the next stage DC. However, fast DCs rely on timing assumptions which are depicted in its STG using dotted arcs, see Figure 14(b).



(a) Circuit (b) STG (c) Timing assumptions

Figure 14: Gate-level fast DC

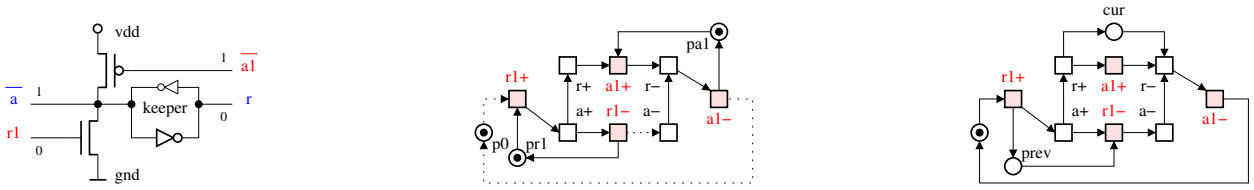
The first timing assumption is captured by the dotted arcs incident to place *p0*. Being the same as for traditional DCs this assumption is satisfied if the number of DC in each loop is grater than two.

The second timing assumption is that the token leaves the DC only after the previous stage DC is empty. It is shown by the dotted arc between transition *r1-* and *a-*. The same timing assumption for the next stage DC is shown by the dotted arc from *e+* to *a1-*. This assumption is easy to meet because the reset of the request *r1-* from the previous stage DC is delayed by a single two-input NOR-gate. The acknowledgement *a1+* from the next stage DC is set with the delay of at least a pair of two-input NOR-gates.

Accepting the above timing assumptions and removing the transitive places the simplified STG of the fast DC is obtained in Figure 14(c). The transitive places *prev* and *cur* represent the high level of signals $r1$ and r respectively. Their state denotes the marking of places associated with the previous stage DC and current DC. One can see that both *prev* and *cur* are marked for the time when *a+* has been executed and *r1-* has not fired yet. This inconsistency between the underlying PN model and fast DCs is called the *token spread.*

Another implementation of a fast DC and its STG are shown in Figure 15(a,b). This implementation uses a *keeper* latch for the state holding element. A *keeper* is a logic level hold circuit which consists of two weak inverters connected back to back. In order to increase the driving ability of the request output, the weak inverter providing the keeper output is replaced by an ordinary inverter. The timing assumptions for the transistor-level implementation are the same as for the gate-level. One can see that the spread of the token is also possible in the transistor-level implementation, see Figure 15(c).

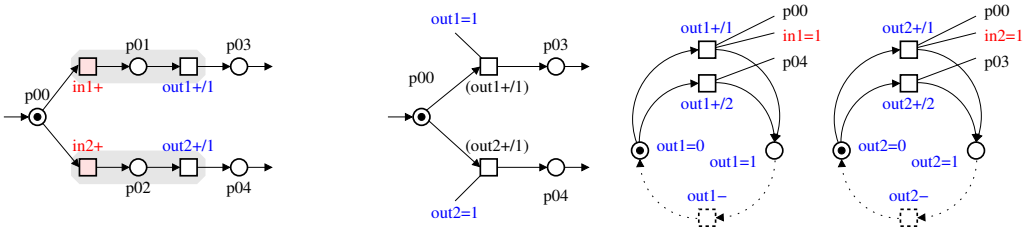
The token spread is not modelled by the underlying PN, which may cause problems in the vicinity of the choice



(a) Circuit (b) STG (c) Timing assumptions

Figure 15: Transistor-level fast DC

place. Consider the example STG shown in Figure 16(a). The transitions that directly succeed places *p01* and *p02* are different signal events, and the removal of these places does not cause any coding conflict. The optimised STG shown in Figure 16(b) can be safely mapped into traditional DCs. However, the direct mapping into fast DCs is problematic due to their token spread feature.



(a) Initial STG (b) Resultant STG

Figure 16: Redundant places after choice

For example, if the STG shown in Figure 16(b) is implemented using fast DCs, then the following scenario is possible: *in1+→out1+/1→(out1+/1)→out2+/1* resulting in the token spread over places *p00* and *p03* for a short time interval. It leads to the incorrect state when transitions *(out2+/1)* and *(out2+/2)* in conflicting branches are enabled simultaneously. The firing of the enabled transition *(out2+/1)* results in the malfunction of the system: both conflicting branches are active at the same time.

A possible solution for the token spread problem is to restrict the propagation of a token in conflicting branches until the token leaves the choice place. It can be done by mapping the first places in the conflicting branches into traditional DCs. Such a DC does not rise its request output until the request of the previous stage DC is low. The application of this approach to the example shown in Figure 16(b) forces places *p03* and *p04* to be mapped into traditional DCs.

The advantages and drawbacks of different DC implementations are summarised in Table 1. Only a traditional DC is free of the token spread problem. The smallest (6 transistors) is the transistor-level fast DC. The fastest (up to 833.3MHz) is the gate-level fast DC.

The maximum frequency of DC operation is measured by SPICE analog simulations using the AMS-0.35μ design kit. For this, traditional DC, gate-level fast DC and transistor-level fast DC have been implemented in AMS-0.35μ library. Then the DCs of each type have been connected in loops of tree DCs and the oscillation of each loop have been captured as shown in Figure 17.

The shortest period is exhibited by gate-level fast DCs. These DCs are the best for the synthesis of fast control circuits. Transistor-level fast DCs are recommended when the circuit size is crucial, however they require extra effort

DC type	min period (ns)	max frequency (MHz)	size (transistor count)	token spread
traditional	3.6	277.8	12	no
fast gate-level	1.2	833.3	8	yes
fast transistor-level	2.1	476.2	6	yes

Table 1: Comparison of DC implementations

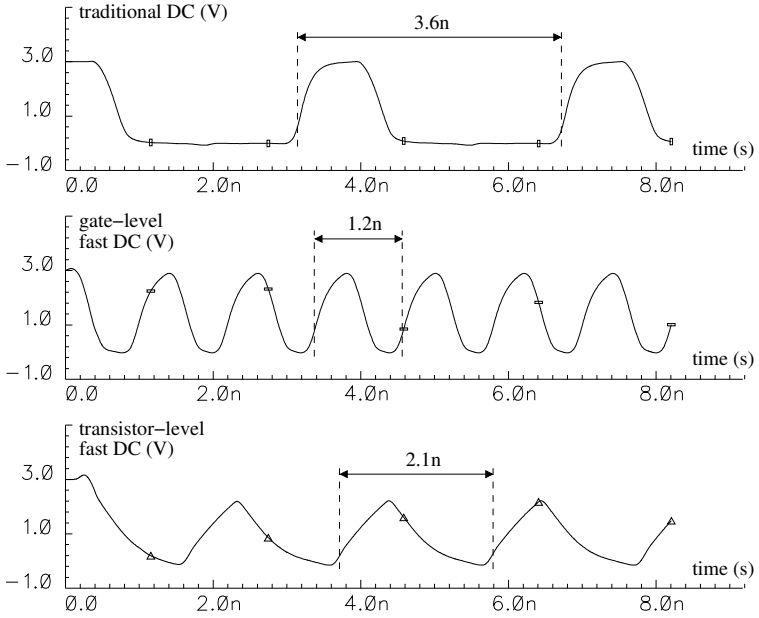


Figure 17: Speed of DC implementations

for the layout of the library of custom cells. Both types of fast DCs rely on timing assumptions and have certain token spread problems. That is why the traditional DCs should be used for the design of speed-independent circuits and to avoid the spread of a token in the vicinity of choice places.

6 Algorithms

This section describes the algorithms employed in the STG optimisation for mapping. The algorithms are implemented in a package of software tools called OptiMist whose design flow is presented in Figure 18.

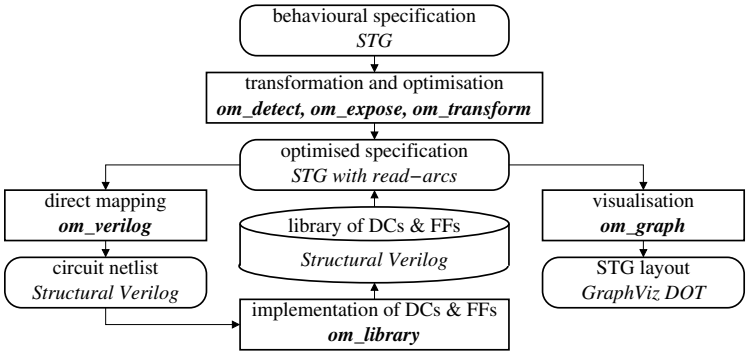


Figure 18: OptiMist design flow

The package consists of separate tools solving the following tasks:

- detection of redundant places;
- exposure of the outputs;
- elimination of redundant places;
- visualisation of an STG with read-arcs extension and tracker-bouncer structure;
- mapping of the optimised specification into a circuit;
- generation of a library of required DCs and FFs either at transistor- or gate-level.

The algorithms for STG optimisation (detection of redundant places, exposure of the outputs and elimination of redundant places) are described in detail in the following subsections. The algorithms for the other three tools are trivial and are not presented here.

6.1 Detection of redundant places

The order in which the redundant places are detected affects the optimisation result. The order is defined by the heuristics presented in Algorithm 1. The *detect_redundant_places* procedure takes *STG* and *optimisation_level* as the input parameters and returns the set P_R of redundant places (lines 01-03). Initially, all STG places are undefined (line 04). The *optimisation_level* parameter defines which optimisation heuristics to apply to the STG (lines 07-12).

Algorithm 1 Detection of redundant places

```
01  procedure detect_redundant_places
02  input: STG = ⟨P, T, F, M0, I, O⟩, optimisation_level ∈ {0, 1, 2, 3}
03  output: PR ⊆ P
04  PR:=∅; PM:=∅; PU:=P
07  if optimisation_level > 0 then
08      optimise_choice(STG, PU, PR, PM)
09      if optimisation_level > 1 then
10          optimise_latency(STG, PU, PR, PM)
11          if optimisation_level > 2 then
12              optimise_size(STG, PU, PR, PM)
```

6.1.1 Choice optimisation

The heuristic *optimise_choice* whose pseudo-code is shown in Algorithm 2 prevents the State Machine-specific coding conflicts. The algorithm implements a trade-off between the computation speed and latency optimisation as described in Section 4. The input of the heuristic is the system STG and initial partitioning of its places into undefined, redundant and mandatory; its output is a new partitioning of the places (lines 01-03).

First, for each choice place p the set T_{choice} containing its postset and post-postset transitions is created (lines 05-07). Then, for each transition t in the postset of choice place p a set T_{conf} is computed (lines 08-09). It contains all transitions in the conflicting branches which are in the postset or post-postset of the choice place p. For each place p_{succ} in the postset of transition t a set T_{seq} containing t and all transitions in the postset of p_{succ} (lines 10-11). If there is a signal whose transition belongs to both T_{seq} and T_{conf} then place p_{succ} is mandatory (lines 12-14). Otherwise place p_{succ} is redundant and all places in its post-postset are made mandatory to reduce the computation complexity (lines 15-18).

Algorithm 2 Choice optimisation

```
01  procedure optimise_choice
02  input: STG = ⟨P, T, F, M0, I, O⟩, PU ⊆ P, PR ⊆ P, PM ⊆ P
03  output: PU ⊆ P, PR ⊆ P, PM ⊆ P
04  for_each p ∈ P : |p•| > 1 do
05      Tchoice := ∅
06      for_each t ∈ p• do
07          Tchoice := Tchoice ∪ {t} ∪ t••
08      for_each t ∈ p• do
09          Tconf := Tchoice \ ({t} ∪ t••)
10          for_each psucc ∈ t• : psucc ∈ PU do
11              Tseq := {t} ∪ psucc•
12              if ∃(z ∈ I ∪ O, tseq ∈ Tseq, tconf ∈ Tconf) :
13                  (tseq ∈ {z+, z−}) ∧ (tconf ∈ {z+, z−}) then
14                  PM := PM ∪ {psucc}
15              else
16                  PR := PR ∪ {psucc}
17                  for_each psucc_succ ∈ psucc•• : psucc_succ ∈ PU do
18                      PM := PM ∪ {psucc_succ}, PU := PU \ {psucc_succ}
19              PU := PU \ {psucc}
```

6.1.2 Latency optimisation

The heuristic *optimise_latency* is aimed at latency reduction. Its basic idea is that a place is redundant if all its direct predecessors are input transitions and all direct successors are non-input transitions. Such a place can be considered redundant even without checking for the possibility of a coding conflict. All its surrounding places are undefined yet which means that the backward neighbourhood includes input transitions only and the forward neighbourhood contains non-input transitions only. Thus, the intersection of the sets of signals in these neighbourhoods is always empty which means the place under question is redundant.

The *optimise_latency* pseudo-code is shown in Algorithm 3. The input of the heuristic is the system STG and initial partitioning of its places into undefined, redundant and mandatory obtained by *optimise_choice* algorithm; its output is a new partitioning of the places (lines 01-03). The algorithm finds all undefined places whose preset transitions are input events and postset transitions are not (lines 04-05). Such places are moved from the set of undefined places into the set of redundant places (lines 06-07).

Algorithm 3 Input-output latency optimisation

```
01  procedure optimise_latency
02  input: STG = ⟨P, T, F, M0, I, O⟩, PU ⊆ P, PR ⊆ P, PM ⊆ P
03  output: PU ⊆ P, PR ⊆ P, PM ⊆ P
04  for_each p ∈ PU do
05      if (∀t ∈ •p, λ(t) ∈ I × {+, −}) ∧ (∀t ∈ p•, λ(t) ∉ I × {+, −}) then
06          PR := PR ∪ {p}
07          PU := PU \ {p}
```

6.1.3 Size optimisation

The *optimise_size* heuristic, whose pseudo-code is presented in Algorithm 4, is aimed at size reduction. The input of the heuristic is the system STG and the partitioning of places into undefined, redundant and mandatory subsets obtained by *optimise_choice* and *optimise_latency* heuristics; its output is the new partitioning of places (lines 01-03). The heuristic is divided into two steps: first, the redundant places are detected in the chains of undefined places (lines 04-10); then, the undefined places left in the STG are checked for redundancy individually (lines 11-17).

At the first step, for each undefined place its backward neighbourhood limited by undefined places and forward neighbourhood limited by non-undefined places are found. If the number of places contained in these neighbourhoods is equal to one (the place under question itself) then this place is a *boundary* place between non-undefined and undefined places (line 05). Such a place is subject for redundancy check (line 06). If for this place there is no signal whose transitions are fired in both forward and backward neighbourhoods limited by redundant places, then the place

Algorithm 4 Size reduction

```
01  procedure optimise_size
02  input: STG = ⟨P, T, F, M0, I, O⟩, PU ⊆ P, PR ⊆ P, PM ⊆ P
03  output: PU ⊆ P, PR ⊆ P, PM ⊆ P
04  // Process sequences of undefined places
05  while ∃p ∈ PU : (εb(p, PR ∪ PM ∪ T) ∩ P = {p}) ∧ (εf(p, PU ∪ T) ∩ P = {p}) do
06      if Af(p, PR) ∩ Ab(p, PR) = ∅ then
07          PR:=PR ∪ {p}
08      else
09          PM:=PM ∪ {p}
10      PU:=PU \ {p}
11  // Process remaining undefined places individually
12  for_each p ∈ PU do
13      if Af(p, PR) ∩ Ab(p, PR) = ∅ then
14          PR:=PR ∪ {p}
15      else
16          PM:=PM ∪ {p}
17      PU:=PU \ {p}
```

is redundant (line 07). Otherwise it is mandatory (line 09). The procedure is repeated until no boundary places left in the STG.

At the second step all undefined places which left in the STG are checked for redundancy without any specific order (lines 11-17). The majority of redundant places are already detected in the previous heuristics. The places left undefined in the STG are usually those preceding the input signal transitions and their removal does not improve the latency, however the size reduction is still possible.

6.2 Exposure of outputs

The conversion of the system STG into a two-level architecture is described by Algorithm 5. The input to the algorithm is a system STG and the initial states S of input and output signals; its output is a modified STG which consists of a tracker and a bouncer (lines 01-03). The initially empty set of read-arcs R connecting the tracker and bouncer is added to the STG (line 04). For each STG signal z a place representing its low level p_z^{low} and a place representing its high level p_z^{high} are created (lines 05-07). The initial marking of these places is chosen according to the initial state S of signal z (lines 08-11). Then, each transition t of signal z is substituted by a dummy transition t_{dummy} in the tracker part (lines 12-18). The signal transition itself is moved into the bouncer part, thus forming the signal elementary cycle (lines 19-23). The tracker and bouncer operation is synchronised by means of read-arcs which are inserted in such way that signal transition t is enabled only when all direct predecessors of the dummy transition t_{dummy} are marked, see read-arcs inserted in lines 24-25. The dummy transition t_{dummy} itself is only enabled when the signal transition t is fired and the marking of signal z elementary cycle is changed, see read-arcs inserted in lines 21 and 23.

6.3 Elimination of redundant places

Algorithm 6 describes the procedure of redundant places elimination. This procedure should be used after detection of redundant places and exposure of outputs. It consists of three steps: initial marking optimisation, trigger signals optimisation and context signal optimisation (lines 04-06).

The procedure of initial marking optimisation is shown in Algorithm 7. It changes the initial marking in such way that no redundant places contain tokens. For this the marking of each redundant place is traversed one transition back assuming that all the places in its postset are marked (lines 04-15). The exception is made for merge places because it is hard to compute which conflicting branch produced the token for the merge place (line 07). The back traversal repeats until either only mandatory places are marked or there is no such transition preceding a marked redundant place whose postset contains places that are all marked. If some redundant places are still marked after the marking recalculation they are made mandatory (lines 16-18). The recalculation of the initial marking for the merge places can

Algorithm 5 Conversion of a system STG into a tracker-bouncer architecture

```
01  procedure convert_tracker_bouncer
02  input: STG = ⟨P, T, F, M0, I, O⟩, S : I ∪ O → {0,1}
03  output: STG = ⟨P, T, F, R, M0, I, O⟩
04  R:=∅
05  for_each z ∈ I ∪ O do
06      create place p_z^low, create place p_z^high
07      P:=P ∪ {p_z^low} ∪ {p_z^high}
08      if S(z) = 0 then
09          M0(p_z^low):=1, M0(p_z^high):=0
10      else
11          M0(p_z^low):=0, M0(p_z^high):=1
12      for_each t ∈ T : λ(t) ∈ {z+, z−} do
13          // Substitute signal z transitions by dummies (in the tracker)
14          create dummy transition t_dummy, T:=T∪{t_dummy}
15          for_each p ∈ •t do
16              F:=F ∪ {(p, t_dummy)}, F:=F \ {(p,t)}
17          for_each p ∈ t• do
18              F:=F ∪ {(t_dummy, p)}, F:=F \ {(t,p)}
19          // Move signal z transitions into elementary cycle (in the bouncer)
20          if λ(t) = z+ then
21              F:=F ∪ {(p_z^low, t)}, F:=F ∪ {(t, p_z^high)}, R:=R ∪ {(p_z^high, t_dummy)}
22          else
23              F:=F ∪ {(p_z^high, t)}, F:=F ∪ {(t, p_z^low)}, R:=R ∪ {(p_z^low, t_dummy)}
24          for_each p ∈ •t_dummy do
25              R:=R ∪ {(p,t)}
```

Algorithm 6 Elimination of redundant places

```
01  procedure optimise
02  input: STG = ⟨P, T, F, R, M0, I, O⟩, P_R ⊆ P
03  output: STG = ⟨P, T, F, R, M0, I, O⟩
04  optimise_marking(STG, P_R)
05  optimise_bouncer(STG, P_R)
06  optimise_tracker(STG, P_R)
```

be improved by employing the reachability analysis algorithms. However, they require either building a finite prefix or a reachability graph which is computationally complex for large specifications.

Algorithm 7 Re-calculation of the initial marking from redundant places to mandatory

```
01  procedure optimise_marking
02  input: STG = ⟨P, T, F, R, M0, I, O⟩, P_R ⊆ P
03  output: STG = ⟨P, T, F, R, M0, I, O⟩, P_R ⊆ P
04  // Try to recalculate the initial marking to mandatory places
05  repeat
06      done:=true
07      for_each p ∈ P_R : (M0(p) ≠ 0) ∧ (|•p| = 1) do
08          for_each t ∈ •p do
09              if ∀p_conc ∈ t• : M0(p_conc) = 1 then
10                  for_each p_pred ∈ •t do
11                      M(p_pred):=1
12                  for_each p_conc ∈ t• do
13                      M(p_conc):=0
14                  done:=false
15  until done
16  // Make all marked places mandatory
17  for_each p ∈ P_R : M(p) ≠ 0 do
18      P_R:=P_R \ {p}, P_M:=P_M ∪ {p}
```

An auxiliary procedure removing an STG node together with its incident arcs is described by Algorithm 8. It is moved to a separate algorithm in order to lighten the *optimise_bouncer* and *optimise_tracker* pseudo-code. The input of the *remove_node* algorithm is an STG and its node which is required to remove. The removal of node *x* starts from the elimination of its producing and consuming arcs (lines 04-07). If *x* is a place, then the read-arcs from this place to all transitions are removed and the node is subtracted from the set of STG places (lines 08-11). If *x* is a transition, then

read-arcs from all places to this transition are removed and x is removed from the set of STG transitions (lines 12-15).

Algorithm 8 Node removal together with its incident arcs

```
01  procedure remove_node
02  input: STG = ⟨P, T, F, R, M0, I, O⟩, PR ⊆ P, x ∈ P ∪ T
03  output: STG = ⟨P, T, F, R, M0, I, O⟩, PR ⊆ P
04  for_each y ∈ •x do
05      F:=F\{(y,x)}
06  for_each y ∈ x• do
07      F:=F\{(x,y)}
08  if x ∈ P then
09      for_each y ∈ x⋆ do
10          R:=R\{(x,y)}
11      P:=P\{x}, PR:=PR\{x}
12  else
13      for_each y ∈ ⋆x do
14          R:=R\{(y,x)}
15      T:=T\{x}
```

The pseudo-code for optimisation of context and trigger signals in the bouncer part is shown in Algorithm 9. It changes the read-arcs connecting the tracker with the bouncer in such a way that only mandatory places control the transitions of elementary cycles. In order to do this for each transition t_{read} which is controlled by a redundant place p its copy t^{dup}_{read} is created and its consuming and producing arcs are duplicated (lines 06-09).

Algorithm 9 Optimisation of the bouncer context and trigger signals

```
01  procedure optimise_bouncer
02  input: STG = ⟨P, T, F, R, M0, I, O⟩, PR ⊆ P
03  output: STG = ⟨P, T, F, R, M0, I, O⟩
04  Ttrig:=∅
05  for_each t_read ∈ p⋆ : p ∈ PR do
06      // Duplicate t_read together with its consuming and producing arcs
07      create transition t^dup_read, T:=T ∪ {t^dup_read}
08      for_each p_read_pred ∈ •t_read do F:=F ∪ {(p_read_pred, t^dup_read)}
09      for_each p_read_succ ∈ t_read• do F:=F ∪ {(t^dup_read, p_read_succ)}
10      for_each t ∈ •p do
11          // Form trigger signals for t^dup_read
12          if t_read ∉ Ttrig then
13              for_each p_read ∈ ⋆t do R:=R ∪ {(p_read, t^dup_read)}
14              Ttrig:=Ttrig ∪ {t^dup_read}
15          // Form context signals for t^dup_read
16          create transition t^dup, T:=T ∪ {t^dup}
17          F:=F ∪ {(t^dup, p)}, F:=F\{(t,p)}
18          for_each p_pred ∈ •t do
19              create place p^dup_pred, P:=P ∪ {p^dup_pred}
20              R:=R ∪ {(p^dup_pred, t^dup_read)}
21              for_each t_pred_succ ∈ p_pred• : t_pred_succ ≠ t do F:=F ∪ {(p^dup_pred, t_pred_succ)}
22              for_each t_pred_pred ∈ •p_pred do F:=F ∪ {(t_pred_pred, p^dup_pred)}
23              for_each t_pred_read ∈ p_pred⋆ do R:=R ∪ {(p^dup_pred, t_pred_read)}
24              F:=F ∪ {(p^dup_pred, t^dup)}
25          // Remove processed transition t and its preset places
26          if |t•| = 0 then
27              for_each p_pred ∈ •t do
28                  remove_node(STG, p_pred)
29              remove_node(STG, t)
30      remove_node(STG, t_read)
```

In our method only those signals whose transitions directly precede an output transition form the set of its triggers. Line 12 checks if transition t_{read} is controlled by a trigger signal yet. If it is not then trigger signals are introduced by means of read-arcs connecting t^{dup}_{read} to each place which controls a transition in the preset of place p (line 13).

Transition t^{dup}_{read} is added to the set T_{trig} containing transitions which are already controlled by trigger signals (line 14).

Recalculation of the context signals involves some change in the tracker structure. Each transition t in the preset of the redundant place p is copied to t^{dup} (line 16). Then producing arc $\left(t^{dup},\, p\right)$ is added and arc $(t,\, p)$ is deleted, thus removing redundant place p from the postset of t (line 17) and each place p_{pred} in the preset of transition t is copied to p^{dup}_{pred} (lines 18-19). This place p^{dup}_{pred} is used by the transition t^{dup}_{read} in the elementary cycle as a new context signal instead of place p, see read-arc $\left(p^{dup}_{pred}, t^{dup}_{read}\right)$ in line 20. All arcs incident to place p_{pred} except consuming arc (p_{pred}, t) are copied to similar arcs connected to place p^{dup}_{pred} (lines 21-23). The consuming arc (p_{pred}, t) is mapped into arc $\left(p^{dup}_{pred}, t^{dup}\right)$ (lines 24).

If the redundant place p was the only place in the postset of transition t then this transition is removed together with its preset places and their incident arcs (lines 26-29). Finally the transition t_{read} is removed together with its consuming, producing and read-arcs. If there are other read-arcs from redundant places to t_{read} they have been copied into the read-arcs to t^{dup}_{read} and are processed in the next iterations of the algorithm. If there are redundant places in the pre-preset of place p these are also processed in the next iterations.

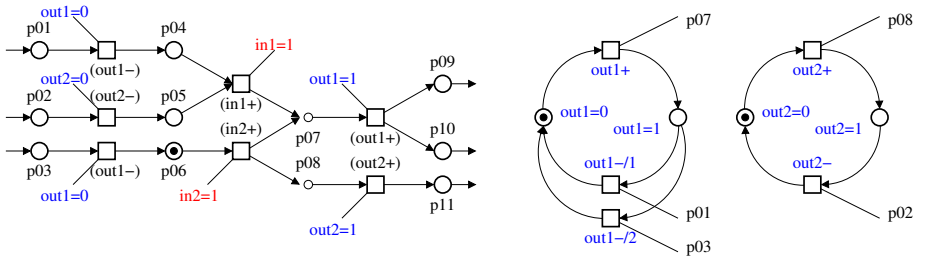
After the application of *optimise_marking* and *optimise_bouncer* algorithms to the device STG its redundant places are not marked with tokens and do not control any transition by means of read-arcs. These places can be removed now by the procedure whose pseudo-code is shown in Algorithm 10. Each redundant place p is removed individually with the required change of the tracker structure. For each transition t in the preset of p a copy t^{dup}_{succ} of the each transition t_{succ} in the postset of p is created (lines 06-10). The copy transition t^{dup}_{succ} is added to the T^{dup} which contains all transitions which are duplicated for t. Incident arcs of t_{succ} except of consuming arc (p, t_{succ}) are also copied (lines 11-13). After that each place in the preset of t is connected by a consuming arc with each transition in T^{dup} (lines 14-17). When all transitions in the preset of p are processed, the transitions in the postset of p and the redundant place p itself are removed (lines 18-25). If redundant place p was the only place in the postset of t then transition t is also removed.

Algorithm 10 Optimisation of the tracker by redundant places removal

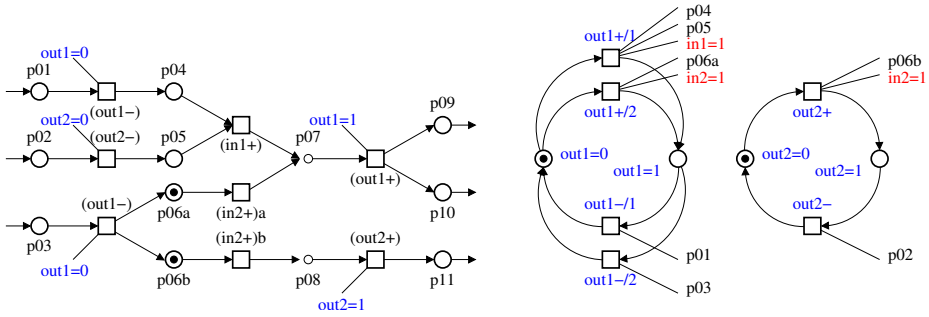
```
01  procedure optimise_tracker
02  input: STG = ⟨P, T, F, R, M0, I, O⟩,  PR ⊆ P
03  output: STG = ⟨P, T, F, R, M0, I, O⟩
04  for_each p ∈ PR do
05      // Duplicate p• transitions and their incident arcs
06      for_each t ∈ •p do
07          T^dup := ∅
08          for_each t_succ ∈ p• do
09              create transition t^dup_succ
10              T := T ∪ t^dup_succ,  T^dup := T^dup ∪ t^dup_succ
11              for_each p_conc ∈ •t_succ : p_conc ≠ p do F := F ∪ {(p_conc, t^dup_succ)}
12              for_each p_succ ∈ t_succ• do F := F ∪ {(t^dup_succ, p_succ)}
13              for_each p_read ∈ ⋆t_succ do R := R ∪ {(p_read, t^dup_succ)}
14          // Connect •t places with all T^dup transitions
15          for_each p_pred ∈ •t do
16              for_each t^dup_succ ∈ T^dup do
17                  F := F ∪ {(p, t^dup_succ)}
18      // Remove redundant place p, p• transitions and processed •p transitions
19      for_each t ∈ •p do
20          F := F \ {(t, p)}
21          if |t•| = 0 then
22              remove_node(STG, t)
23      for_each t_succ ∈ p• do
24          remove_node(STG, t_succ)
25      remove_node(STG, p)
```

The procedure of redundant places removal is described using a simple example whose STG is shown in Figure 19(a). Only places *p07*, *p08* are redundant and the initial marking does not require recalculation. Redundant places *p07* and *p08* control transitions *out1+* and *out2+* respectively. New context and trigger signals for each transi-

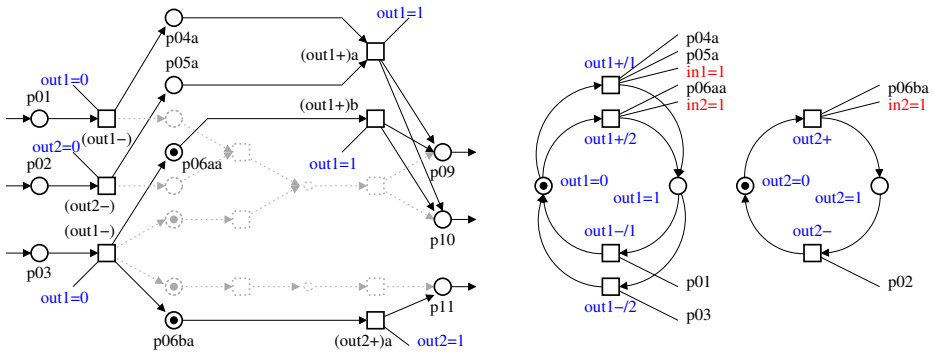
tion are found by *optimise_bouncer* algorithm. Its result is shown in Figure 19(b).



(a) Initial STG with redundant places



(b) Bouncer optimisation



(c) Tracker optimisation

Figure 19: Removal of redundant places

Dummy *(in2+)* is split into *(in2+)a* and *(in2+)b* because it precedes two redundant places. The place *p06* which is in the preset of *(in2+)* is also split into *p06a* and *p06b*, so that *p06a* precedes *(in2+)a* and *p06b* precedes *(in2+)b*.

Only the *(in2+)b* dummy precedes redundant place *p08*. The context and trigger signals of the transition *out2+* are defined by the preset of place *p08*. Its trigger consists of place *in2=1* which controls *(in2+)b* and its context is formed by place *p06b* which is in preset of *(in2+)b*. Read-arcs from *in2=1* to *(in2+)b* and from *p06* to *out2+* are removed.

Two dummies *(in1+)* and *(in2+)a* precede redundant place *p07* which means there are two mutually exclusive sets of triggers/context signals for *out1+*. For this reason the *out1+* transition is duplicated. The trigger of its first copy *out1+/1* is the place *in1=1* which controls *(in1+)*; its context is provided by places *p04* and *p05* which are in preset of *(in1+)*. The trigger of the second copy *out1+/2* is place *in2=1* which controls *(in2+)a*; its context signals places

p06a which is in preset of *(in2+)a*. Read-arcs from *p07* to *out1+*, from *in1=1* to *(in1+)* and from *in2=1* to *(in2+)a* are removed.

Redundant places *p07* and *p08* are then removed from the STG using *optimise_tracker* algorithm whose result is shown in Figure 19(c). Note that dummy *(out1+)* is split into *(out1+)a* and *(out1+)b*. There are two transitions in the preset of *p07*, for each of them a copy of *p07* postset is created.

The algorithms presented in this section are implemented in the OptiMist toolkit. The toolkit automates the mapping of STGs into circuits. At the same time it gives a designer full control on the choice of optimisation heuristics and allows manual adjustment of the solution to specific requirements. OptiMist can be employed in combination with Cadence to allow simulation and technology mapping of circuits. A basic library of DCs and FFs has been created for Cadence. It can be expanded, if necessary, using a tool from the OptiMist package which generates a Verilog netlist for DCs and FFs at transistor-level or gate-level.

The results presented in Section 7 and Section 8 are obtained by OptiMist tools.

7 GCD controller example

Consider the use of the OptiMist tools on the example of the GCD control unit whose STG is shown in Figure 20.

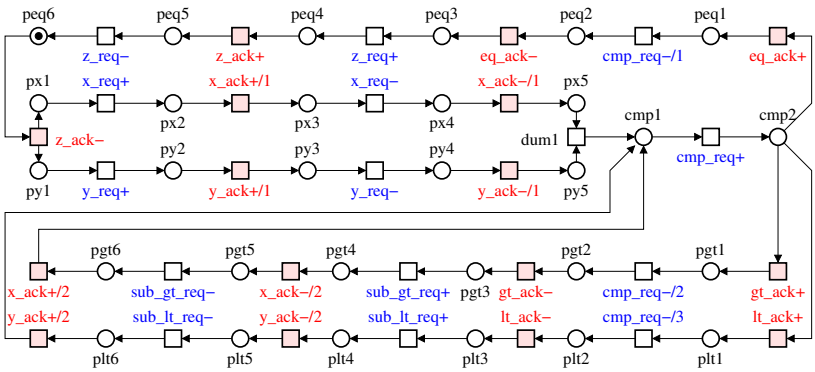


Figure 20: STG of GCD controller

The redundant places are detected in the original STG before the exposure of outputs by the `om_detect` tool. The first heuristic applied to the GCD example is *optimise_choice*. It prevents coding conflicts from occurring in choice conflicting branches without restricting the latency reduction. The result of this heuristic is shown in Figure 21(a). It is obtained by the following command:

```
$ om_detect --level1 --output gcd_1.g gcd.g
```

There are three places *pgt1*, *peq1* and *plt1* which are in the post-postset of the free-choice place *cmp2*. All of them are mandatory because they are preceding the transitions of the same signal *cmp_req*. Making these places mandatory reduces the input-output latency for *eq_ack+*→*cmp_req-/1*, *gt_ack+*→*cmp_req-/2* and *lt_ack+*→*cmp_req-/3* handshakes. However, it is the only way to avoid a coding conflict.

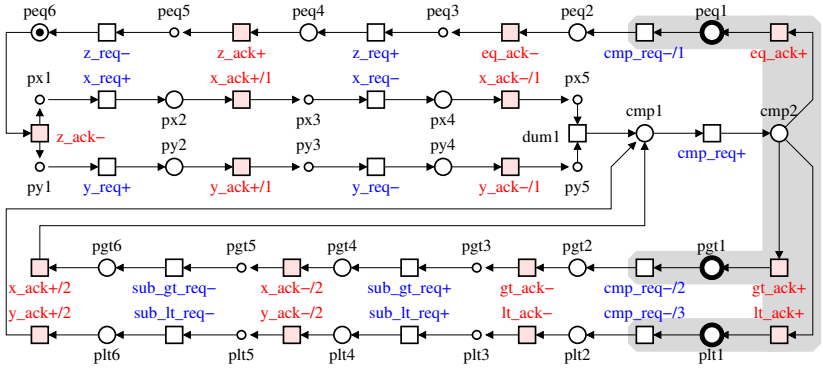
The second heuristic *optimise_latency* reduces both the size and the latency of a circuit. The redundant places detected by this heuristic in the GCD example are *px1*, *py1*, *px3*, *py3*, *px5*, *py5*, *pgt3*, *plt3*, *pgt5*, *plt5*, *peq3* and *peq5*, see Figure 21(b). The preset of each of these places contains transitions of input signals only and the postset contains transitions of non-input signals. The command executed for detecting these redundant places is:

```
$ om_detect --level2 --output gcd_2.g gcd.g
```

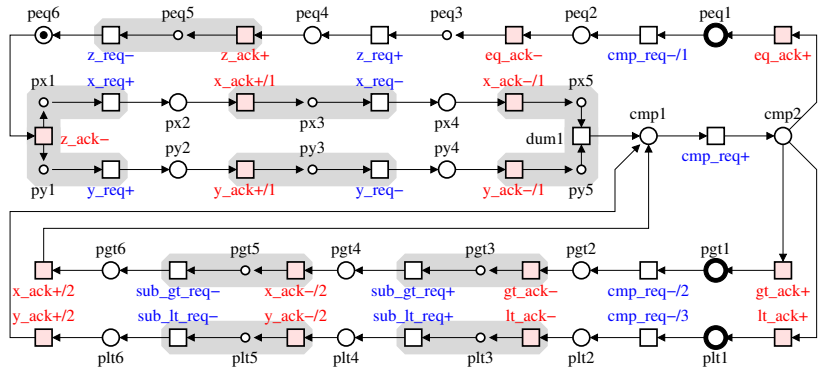
The last heuristic *optimise_size* detects redundant places *cmp2, pgt2*, *plt2* and *peq2* in the GCD example, see Figure 21(c). Removal of place *cmp1* also does not cause a coding conflict, however it is kept by the `om_detect` tool in order to preserve the simplicity of the *cmp_req* elementary cycle. Without this place the positive phase of the *cmp_req* would be controlled by two context signals from the tracker (read-arcs from *px4* and *py4*) and two trigger signals from

the environment (read-arcs from places *x_ack=0* and *y_ack=0*). The trade-off between the complexity of elementary cycles and the number of places in the tracker can be set by command line parameters of the `om_detect` tool:

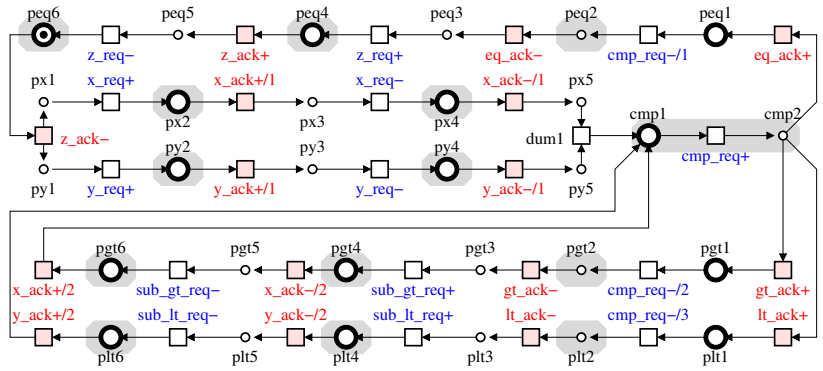
```
$ om_detect --level3 --join2 --output gcd_3.g gcd.g
```



(a) Choice optimisation



(b) Latency optimisation



(c) Size optimisation

Figure 21: Detection of redundant places in STG of GCD controller

After the detection of redundant places the `om_expose` tool partitions the STG of GCD control path into tracker and bouncer parts:

```
$ om_expose --output gcd_3e.g gcd_3.g
```

The resultant STG is shown in Figure 22. The bouncer consists of elementary cycles representing the outputs of GCD controller, one cycle for each output. The elementary cycles for the inputs are not shown as they belong to the environment. The tracker is connected to inputs and outputs of the system by means of read-arcs, as it is described in the algorithm of outputs exposure.

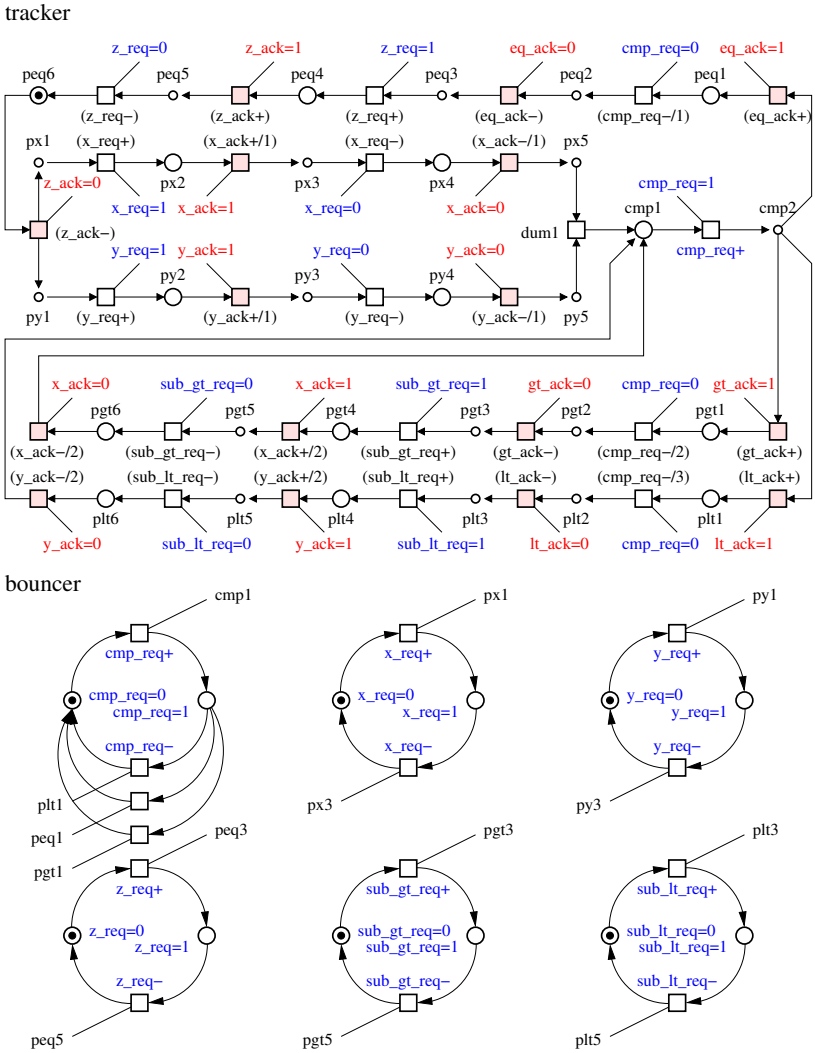


Figure 22: Exposure of outputs in STG of GCD controller

After the redundant places are detected and the outputs are exposed, the STG is optimised by removing the redundant places from the tracker part. The removal of a place involves the change in the STG structure but preserves the behaviour of the system w.r.t. input-output interface. The result of GCD control unit optimisation is presented in Figure 23. This operation is automatically performed by the `om_transform` tool:

```
$ om_transform --level5 --output gcd_3et.g gcd_3e.g
```

This STG can now be used for circuit synthesis. For this each tracker place is mapped into a DC and each elementary cycle is mapped into a FF. The request and acknowledgement functions of a DC are mapped from the structure of the tracker in the vicinity of the corresponding place. The set and reset functions of a FF are mapped from the structure of the set and reset phases of the corresponding elementary cycle. The GCD controller circuit obtained by this technique is presented in Figure 24. The netlist is produced automatically by the following command:

```
$ om_verilog gcd_3et.g --statistics --output gcd.v
```

This circuit consists of 15 DCs and 6 FFs. If the DCs are implemented as transistor-level fast DCs then the maximum number of transistor levels in pull-up and pull-down stacks is 4. This transistor stack appears in the request

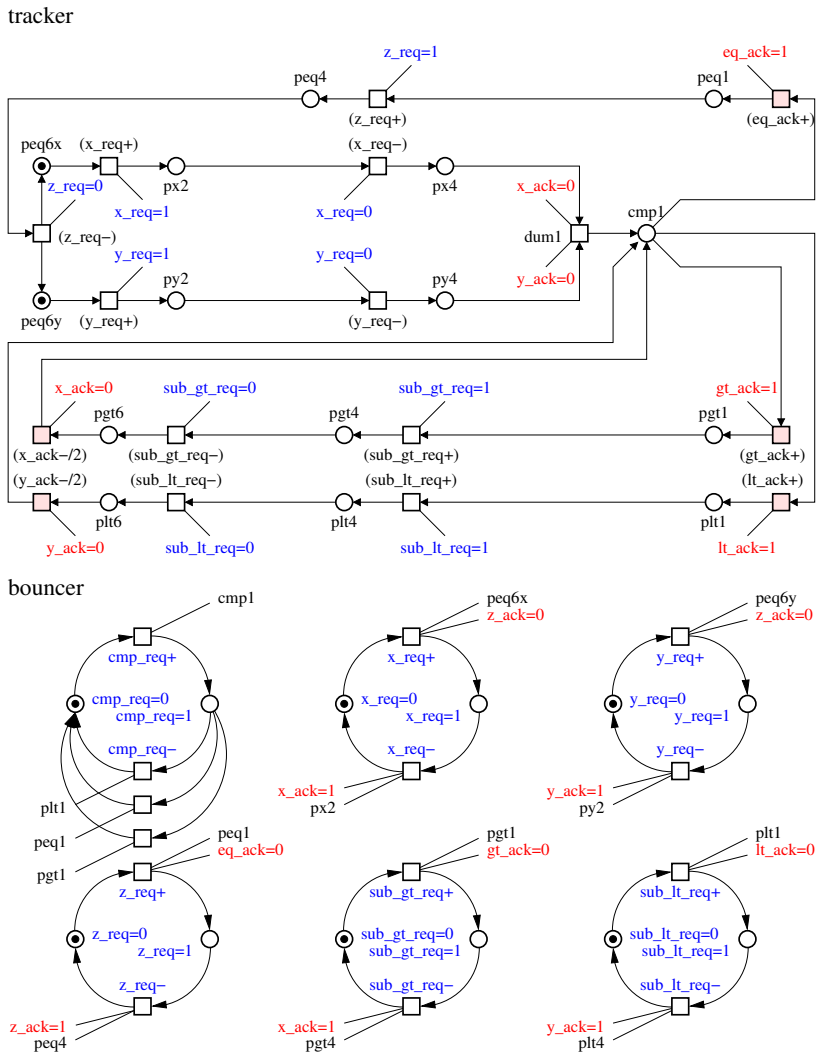


Figure 23: Elimination of redundant places in STG of GCD controller

function of the DC for *cmp1* and is formed by the signals *x_ack=0*, *y_ack=0*, *px4_req* and *py4_req*.

The longest latency, which is the delay between an input change and reaction of the controller by changing some outputs, is exhibited by *cmp_req* signal. The latency of its set and reset phases is equal to the delay of one DC and one FF. The other outputs are triggered directly by input signals which means that their latencies are equal to one FF delay plus the delay of one inverter when the trigger signal requires inversion.

8 Benchmarks

This section highlights the advantages and drawbacks of the direct mapping approach implemented in OptiMist for a set of benchmarks. The direct mapping approach is compared against explicit logic synthesis (implemented in Petrify) in terms of circuit size and speed. The complexity of the underlying algorithms is taken into account by measuring the computation time of OptiMist and Petrify on Pentium 3 1GHz, 1Gb RAM computer. The effect of optimisation heuristics on the direct mapping is also analysed. For this comparison each benchmark STG has been synthesised in three different ways:

- Direct mapping by the OptiMist tools without detection and elimination of redundant places;

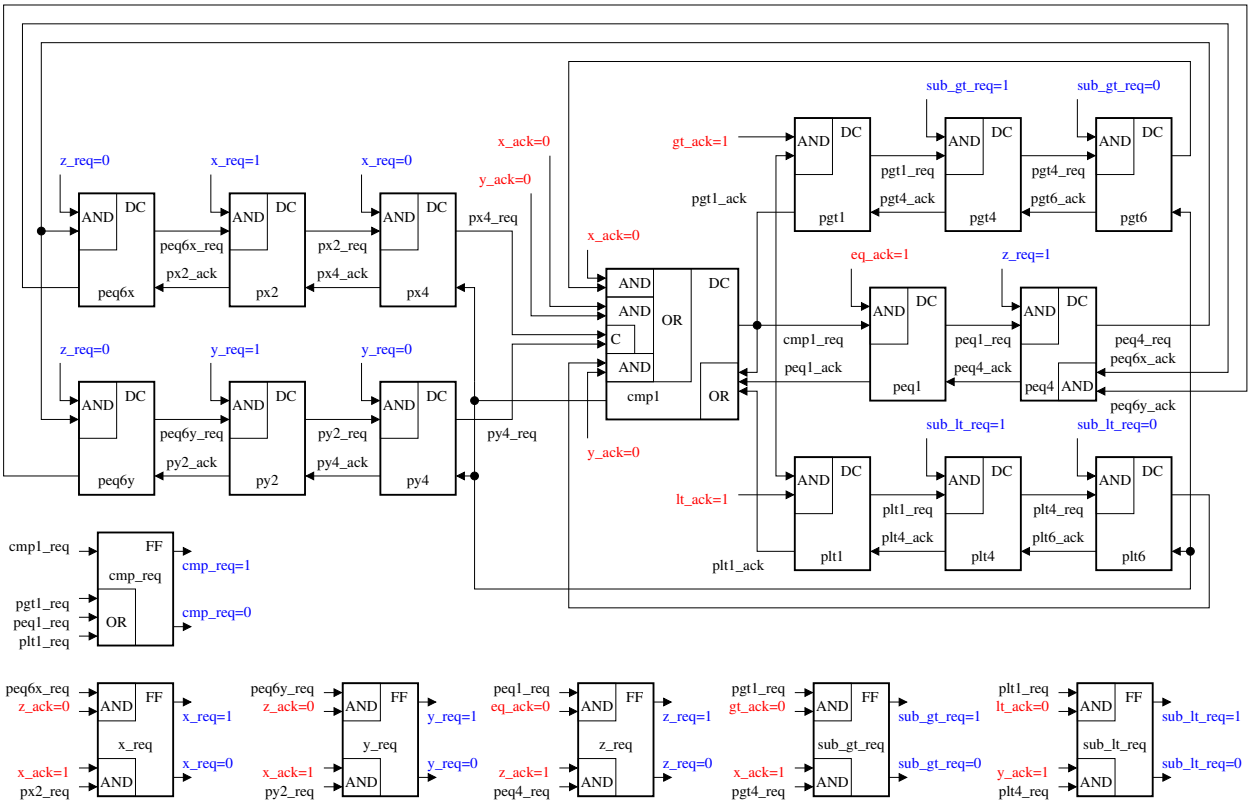


Figure 24: GCD controller circuit obtained by the OptiMist tool

- Direct mapping by the OptiMist tools with latency and size optimisation by removing the redundant places from the STG;
- Logic synthesis by the Petrify tool with automatic resolution of CSC conflicts (unless it is impossible) and logic decomposition into gates with at most four literals.

The result of the experiment is summarised in Table 2.

The number of transistors is counted for the case of places being implemented as fast DCs, request-acknowledgement logic of DCs and set-reset logic of FFs being implemented at transistor level. The condition of having at least three DCs in a loop is met.

In all experiments, the latency is counted as the accumulative delay of negative gates switched between an input and the next output. The following dependency of a negative gate delay on its complexity is used. The latency of an inverter is associated with a unit delay. Gates which have maximum two transistors in their transistor stacks are associated with 1.5 units; 3 transistors - 2.0 units; 4 transistors - 2.5 units. This approximate dependency is derived from the analysis of the gates in AMS 0.35μm library. The method of latency estimation does not claim to be very accurate. However, it takes into account not only the number of gates switched between an input and the next output, but also the complexity of these gates.

All experiments show the high efficiency of direct mapping optimisation heuristics. About 50% of DCs are redundant in the original STG. Their removal results in up to 35% improvement in the circuit size. The latency of the circuits also benefits from the optimisation. In some cases (*gcd*, *vme-bus*) the worst-case latency cannot be improved because of a potential coding conflict in the conflicting branches. However, this latency is only exhibited by the first output signal after the choice place. The latency of the other outputs is reduced.

The comparison of the circuits obtained by OptiMist (with latency&size optimisation) and Petrify shows that the direct mapping solutions are usually larger than logic synthesis solutions. However the circuits obtained by the direct

benchmark name	DC count	max fin	max fout	transistor count	worst-case latency	computation time
gcd						
OptiMist (no optimisation)	30	2	4	255	4.5	0.11s
OptiMist (latency&size optimisation)	14	3	4	174	4.5	0.18s
Petrify				116	11.0	18s
vme-bus						
OptiMist (no optimisation)	17	3	3	155	5.0	0.09s
OptiMist (latency&size optimisation)	10	3	4	121	5.0	0.11s
Petrify				58	8.5	1s
toggle						
OptiMist (no optimisation)	8	2	2	68	4.5	0.06s
OptiMist (latency&size optimisation)	4	2	2	44	3.5	0.07s
Petrify				22	3.5	0.12s
imec-alloc-outbound						
OptiMist (no optimisation)	17	2	2	143	5.0	0.09s
OptiMist (latency&size optimisation)	6	2	5	73	3.0	0.16s
Petrify				46	7.5	6.6s
par3						
OptiMist (no optimisation)	19	3	4	160	6.5	0.07s
OptiMist (latency&size optimisation)	15	4	4	114	4.5	0.09s
Petrify				78	12.5	11s
count						
OptiMist (no optimisation)	19	3	3	150	5.5	0.07s
OptiMist (latency&size optimisation)	11	3	4	98	3.0	0.11s
Petrify (manual CSC resolution)				68	3.0	1.4s

Table 2: Comparison between OptiMist and Petrify

mapping technique exhibit lower output latency.

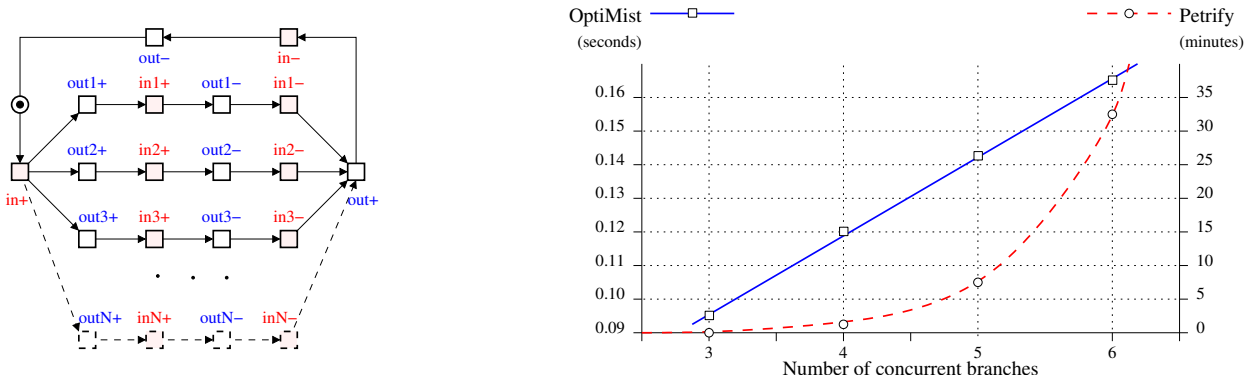
For some benchmarks (e.g *count*) Petrify fails to resolve a CSC conflict even if it is reducible. Manual insertion of additional signals is required in such cases. However, OptiMist completes the job automatically for such benchmarks.

The OptiMist tools can also process large specifications, which are not computable by Petrify in acceptable time. This can be illustrated on the scalable benchmark whose STG is shown in Figure 25(a). Adding the concurrent branches as shown by dashed lines one can increase the complexity of the benchmark. When the concurrency increases, the Petrify computation time grows exponentially, while the OptiMist computation time grows linearly on the same benchmark, see Figure 25(b). Note the different time scale for OptiMist (seconds) and for Petrify (minutes).

9 Summary

A method for direct mapping of STGs into circuit netlist has been presented in this chapter. The method exploits the two-level architecture where a circuit consists of two blocks: the tracker and the block of output flip-flops. The tracker computes context signals for outputs concurrently with the environment operation, thus achieving the latency reduction effect. The output flip-flops generate outputs from context and trigger signals. The adopted architecture allows the minimisation of state-holding elements and reduction of latency. The characteristic feature of the method is that the optimisation is achieved at the specification (Petri net) level as opposed to optimisation of logic circuits after the synthesis stage.

The method is implemented in a package of software tools called OptiMist. The package take an STG as the initial specification of a system, converts the STG in a form convenient for mapping, performs optimisation, and



(a) Scalable benchmark STG

(b) Computation time

Figure 25: Dependency of computation time on STG complexity

produces a Verilog netlist of the circuit. The optimisation of the specification relies on a set of heuristics aimed at circuit latency and size reduction. This package can be employed in combination with Cadence for simulation and technology mapping of circuits.

In the OptiMist tools the optimisation is performed locally and the computation time grows linearly with the size of specification. This allows to process large specifications which are not computable by logic synthesis tools in acceptable time.

The OptiMist tools are fully automated. At the same time a designer can significantly influence the result by choosing one or more optimisation heuristics. In combination with computation speed OptiMist gives the designer an opportunity to synthesise circuits with different optimisation parameters and choose the best solution.

References

[1] Peter A. Beerel and Teresa H. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conference Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, November 1992.

[2] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[3] Alex Bystrov and Alex Yakovlev. Asynchronous circuit synthesis by direct mapping: Interfacing to environment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 127–136, Manchester, UK, April 2002. IEEE Computer Society Press.

[4] Wei-Chun Chou, Peter A. Beerel, and Kenneth Y. Yun. Average-case technology mapping of asynchronous burst-mode circuits. *IEEE Transactions on Computer-Aided Design*, 18(10):1418–1434, October 1999.

[5] Tam-Anh Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, June 1987.

[6] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336. Academic Press, 1967.

[7] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, Spain, November 1996.

[8] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. *Hardware and Petri nets: application to asynchronous circuit design*, volume 1825 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, June 2000.

[9] René David. Modular design of asynchronous circuits defined by graphs. *IEEE Transactions on Computers*, 26(8):727–737, August 1977.

[10] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, University of Utah, September 1997.

[11] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[12] R. M. Fuhrer, Steven M. Nowick, Michael Theobald, N. K. Jha, B. Lin, and Luis Plana. Minimalist: an environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, July 1999.

[13] R.J. van Glabbeek and Peter W. Weijland. Branching time and abstraction in bisimulation semantics. 43(3):555–600, 1996.

[14] Matthew C. B. Hennessy and Robin Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309, Noordweijkerhout, Netherland, July 1980. Springer-Verlag.

[15] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.

[16] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964. Reprinted from J.Franklin Institute, vol. 257, no. 3, pp. 161–190, Mar. 1954, and no. 4, pp. 275–303, Apr. 1954.

[17] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer-Verlag, 1997.

[18] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent hardware: the theory and practice of self-timed design*. Series in Parallel Computing. Wiley-Interscience, John Wiley& Sons, Inc., 1994.

[19] Luciano Lavagno and Alberto Sagiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits*. Kluwer Academic Publishers, 1993.

[20] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[21] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[22] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[23] U. Montanari and F. Rossi. Contextual nets. *Acta Informacia*, 32(6):545–596, 1995.

[24] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[25] Tadao Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[26] Chris J. Myers and Teresa H. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[27] Enric Pastor. *Structural methods for the synthesis of asynchronous circuits from signal transition graphs*. PhD thesis, Universitat Politècnica de Catalunya, February 1996.

[28] Suhas S. Patil and J. B. Dennis. The description and realization of digital systems. In *Proc. IEEE COMPCON*, pages 223–226, 1972.

[29] Carl Adam Petri. *Kommunikation mit automaten (Communicating with automata)*. PhD thesis, University of Bonn, 1962.

[30] Wolfgang Reisig and Grzegorz Rozenberg. Informal introduction to Petri nets. In *Petri nets*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[31] Leonid Rosenblum and Alex Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.

[32] Hiroshi Saito, Alex Kondratyev, Jordi Cortadella, Luciano Lavagno, and Alex Yakovlev. What is the cost of delay insensitivity? In *Proc. International Conference Computer-Aided Design (ICCAD)*, pages 316–323, November 1999.

[33] Charles L. Seitz. System timing. In Carver A. and Mead, editor, *Introduction to VLSI systems*, chapter 7. Addison-Wesley, 1980.

[34] Ivan Sutherland and Scott Fairbanks. GasP: a minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 46–53. IEEE Computer Society Press, March 2001.

[35] Jan Tijmen Udding. *Classification and composition of delay-insensitive circuits*. PhD thesis, Eindhoven University of Technology, 1984.

[36] S. H. Unger. *Asynchronous sequential switching circuits*. Wiley-Interscience, John Wiley& Sons, Inc., New York, USA, 1969.

[37] Victor Varshavsky and V. Marakhovsky. Asynchronous control device design by net model behavior simulation. In J. Billington and Wolfgang Reisig, editors, *Application and Theory of Petri Nets*, volume 1091 of *Lecture Notes in Computer Science*, pages 497–515. Springer-Verlag, June 1996.

[38] Alex Yakovlev, Albert Koelmans, and Luciano Lavagno. High-level modeling and design of asynchronous interface logic. *IEEE Design & Test of Computers*, 12(1):32–40, 1995.